Cairo University

Faculty of Engineering

Electrical Power Department

Graduation Project Spring 2023-2024

Control of Self-balancing Robot

Submitted By:

Tarek Samir Kamel	**9202742**
Tarek Ahmed Saeed	**9202741**
Mohanad Osama Khalil	**9203570**
Adham Nabil Fares	**9202231**

Supervised By:

Prof. Abdel-Latif Elshafei

Cairo University
Faculty of
Engineering
Electrical Power Engineering Department

Graduation Project Spring 2023-2024

Control of Self-balancing Robot

By

Tarek Samir Kamel

Tarek Ahmed Saeed

Mohanad Osama Khalil

Adham Nabil Fares

Supervised by

Prof. Abdel-Latif Elshafei

A Graduation Project Report
Presented To

Electrical Power Engineering Department

Acknowledgment

A special thanks and great appreciation are given to everyone who assisted us in accomplishing this project through valuable help, advice, and guidance. This acknowledgment is dedicated to our graduation project's supervisor ***Dr. Abd-EL Latif***. The team members want also to express their gratitude for the sincere advice and unconditional help from the department's T.A.s and fellow colleagues in the other teams.

Team members,

Abstract

This report details the processes for building a self-balancing robot. The self-balancing robot is based on the principle of an inverted pendulum on a moving cart. The robot is inherently unstable. The system may be balanced using a microcontroller, a gyroscope to calculate angle of inclination, and two stepper motors that operate as actuators via an appropriate control mechanism. The proportional-derivative-integral control mechanism is the most prevalent, and it was used to balance our robot. The chapters of the report go into depth on the stages of building the robot, beginning with ***chapter 1***, which offers an introduction to the challenge and the motivation for picking this project. ***Chapter 2*** outlines the primary components employed in our robot, including their operating principles, benefits, downsides and alternatives and outlines numerous tests performed on the components individually, as well as tests performed when these components are interfaced together, in order to fully understand their capabilities and limits. ***Chapter 3*** covers topics related to the mechanical structure, beginning with comparing different materials to build the robot and selecting the best one, and ending with assembling the parts together using nails, screws, and bolts, as well as designing the mechanical structure on Solid-Works and estimating whether the motor torque will be sufficient to withstand the robot's weight. ***Chapter 4*** covers the design of the electric circuit and the printed circuit board. ***Chapter 5*** describe how to balance a self-balancing robot. ***Chapter 6*** goes into depth on the proportional-integral-derivative method for successful robot control, including tuning approaches and describes the mechanical model that was simulated in SIMULINK. This chapter explains the functionality of each block and how to assemble the model. ***Chapter 7*** provides a detailed explanation of the Arduino code. The Arduino programs presented in this chapter include not just those used in the final project, but also those needed for troubleshooting to ensure that the various components perform as intended. ***Chapter 8***, the last chapter, discusses the project and recommends future enhancements that might be made to the motor and the others component. And finally, the appendix concludes the whole final code.

Table of content

List of Figures

Chapter 1 Introduction

1.1 Overview:

Robotics is the study of designing, building, and employing robots to do jobs that were previously done by humans. These duties are either repetitive, like as in the car sector, or dangerous, putting individuals at risk of injury. Many parts of robotics use artificial intelligence. Robots are equipped with sensors that replicate human senses such as vision, touch, and temperature sensing. Current robotics research is focused on developing robots that are self-sufficient and capable of making basic decisions in unfamiliar environments. Most industrial robots do not resemble humans and are intended for specialized uses and scenarios, such as robotic arms used in the automobile industry to weld and screw together. As a result of modern robotics research, tremendous breakthroughs in microcontrollers, precise sensors, and different controlling methods have been gained in the previous few decades, leading in the creation of several sophisticated applications such as self-balancing robots [1].

A self-balancing robot has two wheels and balances itself to avoid falling. The concept is similar to that of a unicycle, in which the rider balances by moving in the same direction as the incline. Self-balancing robots employ a closed feedback control system in which real-time input from sensors (gyroscopes) is utilized to regulate stepper motors to adjust for tilting and maintain the robot upright. Self-balancing robots can be used in a variety of applications, including autonomous trolleys, ship balancing, and Segways. [2]

1.2 Problem Description

Self-balancing robots are inherently unstable systems with numerous variables and nonlinearity. The self-balancing robot operates on the same concept as an inverted pendulum. Both systems are naturally unbalanced and require an effective control mechanism to stabilize by keeping the position upright. Stabilization can be achieved using a variety of control approaches, including proportional, integral, derivative (PID), linear quadratic control (LQR), and fuzzy logic control, to mention a few. To control the self-balancing robot, angle and robot position information must be precisely and rapidly sent to the microcontroller in order to achieve a quick, right action to restore system stability.

1.3 Inverted Pendulum Theory

The inverted pendulum has its center of mass above the pivot point. The vertical rod is mounted on a rolling cart that can only go back and forth. The inverted pendulum is a typical example used in control system references. Without control, the pendulum becomes unstable and will fall if the cart does not move to balance it. The control system should exert force on the cart in order to maintain the pendulum balanced. [3]

To develop an effective control system, all critical variables, such as the forces operating on the inverted pendulum, must be determined in order to derive the pendulum's transfer function, which will be employed in the controller design.

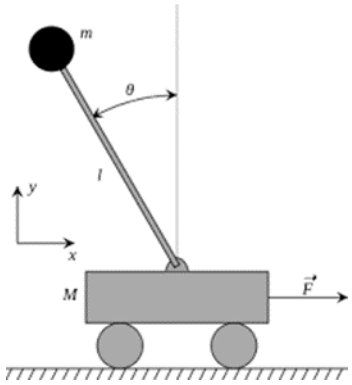

Figure 1: Inverted pendulum free body diagram

The inverted pendulum and self-balancing robots share many similarities. Both systems are unstable and require a control mechanism that can apply force to maintain stability. There are just a few distinctions between the self-balancing robot and the inverted pendulum. To begin, when generating the transfer function of the inverted pendulum, the value of friction torque may be ignored since the friction coefficient is considered to be close to 0. However, friction torque cannot be neglected in self-balancing robots since their wheels will be traveling over surfaces with high friction coefficients. Taking friction torque into consideration results in more precise control movements that aid in the achievement of stability.

Another distinction is the amount of motion. The cart on which the vertical rod is fixed goes only back and forth, but the self-balancing robot moves in all directions since it will be put on a floor plane.

1.4 Project Motivations

As previously stated, self-balancing robots find usage in a variety of automation and industrial applications. They are also utilized in educational and academic settings to provide students with hands-on experience applying control system ideas. It also requires students to deal with hardware issues and constraints, such as building short circuit safety mechanisms and taking into account the rated current limits of certain components. The project also introduces students to mechanical difficulties that must be addressed, including as the maximum weight that the stepper motors can handle and determining the best placement for the center of gravity to provide stability.

1.5 Project Main Idea and Stages

Our primary goal is to balance the robot using an Arduino microcontroller, a gyroscope to determine the angle of inclination, and stepper motors to take necessary action and return the robot to its reference position if any inclination occurs. The self-balancing robot will be conceived and built from scratch.

The project is divided into four stages. The initial stage involves testing each component independently. The second stage involves mechanical design. The third stage is the PCB design. The final stage is determining the best PID settings to ensure stability.

The first stage is to test and familiarize yourself with each component independently. These experiments involved the use of stepper motors, a gyroscope as a level meter, and the integration of the stepper motors and gyroscope. These tests were useful in obtaining an understanding of Arduino programming as well as electrical connections between stepper motors, drivers, gyroscopes, and Arduino microcontrollers.

The second stage is the design and implementation of the printed circuit board (PCB) and testing it.

The third stage is the design of the mechanical structure. This stage includes choosing the appropriate material to build the robot and determining the dimensions of the robot to achieve a robust design. This stage also included choosing suitable wheels that can be coupled easily with the shaft of the stepper motor and that can withstand the weight of the robot as well as tolerate uneven, rough surfaces that the robot might move on.

The fourth and last stage includes assembling the self-balancing robot and acquiring the values of the PID controller. It was decided to use the PID method as a control system. In this stage, a mechanical simulation was modeled on SIMULINK and tested in order to get initial values for the proportional, integral and derivative terms. The mechanical SIMULINK model doesn't accurately represent the real-life robot; however, it provides initial guesses for PID values and then a trial-and-error approach is taken until a desired performance is reached and the robot is self-balanced.

Chapter 2 Main Components

2.1 Stepper Motors

The project's actuator is the first component to be covered. It was found that stepper motors and DC motors are the most popular motor types used in small robotic applications because of their torque-speed characteristics, that best suit for the needs of the self-balancing robot, and their appropriate sizes for a project of this size. A comparison between the DC motor and a stepper motor was necessary to determine which kind of motor would best meet the needs and requirements of the project. A short overview of the main differences between the two types of motors is provided in the following table: [4]

Points of comparison	Stepper motor	Brushed DC motor
Speed	The speed of stepper motor is low, about 200 to 2000 RPM.	DC motors have moderate speed range depending on the type of motor.
Torque-speed characteristics	Stepper motors produce maximum torque at low speeds. The toque decreases as speed increases.	DC motors produce high torque at low speeds
Control mechanism	The control mechanism of stepper motor needs micro- controllers.	The control mechanism of DC motors is simple and no need of extra devices like micro-controllers.
Efficiency	The efficiency of stepper motors is low.	The DC motors have high efficiency around 85%.
Noise	Arcs in the brushes will generate noise	Generates some noise especially at low speeds
Cost	Moderate	Low
Applications	Stepper motors are used in various position control applications such as robotics, printers, hard disc drives, etc.	DC motors are commonly used in toys, computers, cranes, kitchen appliances, lifts, etc.

Stepper motors are chosen for their superior torque at low speeds, accuracy, output characteristics and simple controllability.

Figure 2: NEMA-17 (Stepper motor)

Unlike DC motors which rotate in a continuous manner, stepper motors rotate in increments or steps. Rotating in steps made stepper motors suitable when very precise positioning and high torque are required. That is why stepper motors are used in 3D printers, CNC machines, DVD drives and analog clocks [5].

Stepper motors are classified according to the way of magnetic field creation into variable reluctance, permanent magnet, and hybrid synchronous. The most available type commercial-wise is the hybrid-synchronous which is further classified into bi-polar and unipolar stepper motors.

2.1.1 Theory of Operation:

Stepper motors include a magnetized gear core that will turn once force is applied to it. The core is enclosed by a number of coils that act as electro-magnets. The coils are arranged near the rotor so that magnetic fields within the coil can control the movement of the rotor. By controlling the value of the current entering these coils via a micro-controller, the rotor can be controlled to move discrete steps.

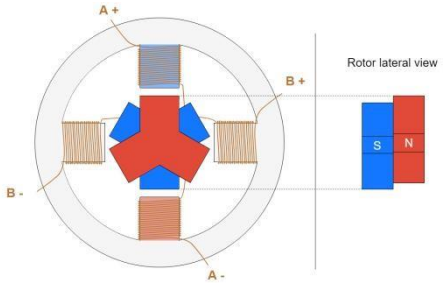

Figure 3: Hybrid stepper motor. [7]

The electromagnetism theory explains the operation of stepper motors. when the current passes through one of the coils, magnetic field is formed which is perpendicular to the direction of the current. Each coil is energized one at a time. The rotor aligns itself with the coil currently energized in a position which gives field lines the path with the least reluctance. Using a microcontroller, the current can be sent as pulses and energize coils one at a time so that rotor moves in an incremental fashion or steps. The number of coils determines the size of each step. The NEMA 17 stepper motor has a step size of 1.8o. [6]

Micro-controllers can be programmed to allow micro-stepping. If equal magnitudes of current are supplied to two coils at the same time, the rotor will align itself at a position half-way between the two coils. By extending this principle, the ratio of current between two coils can be manipulated to allow very precise movement that can reach a resolution of a fraction of a step. The following figure illustrates the idea of micro-stepping further:

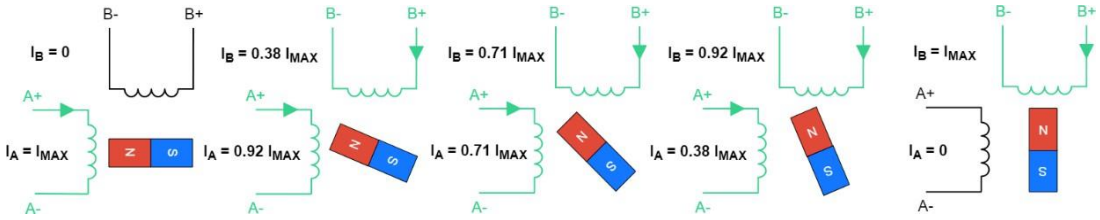

Figure 4: Micro-stepping. [7]

I_{max} is the maximum current of one phase. Starting from left, $I_A = I_{max}$ while $I_B = 0$. In the next step, $I_A = 0.92I_{max}$ while $I_B = 0.38I_{max}$. The magnetic field is rotated by 22.5° compared to the initial state. Using micro-stepping helps in reaching fully accurate positions without the need to add more coils. [7]

2.1.2 Stepper Motor types:

Unipolar stepper motors have one winding for each phase with a center tap at each winding. This gives six connections, however, the center taps of the two windings can be connected to give five connections. In unipolar stepper motors, current flows in one direction hence the name. Fixed direction of current allows simple control as there is no need to reverse the direction of the current. Control of unipolar stepper motor is done by using four MOSFETs in a half-bridge configuration. However, the unipolar motor uses only half of its coil at a time which means less efficiency and less output torque, but they have greater max speed than bi-directional motors. [8]

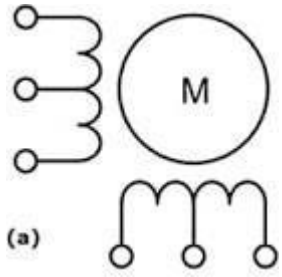

Figure 5: Unipolar stepper motor. [37]

Bi-polar stepper motors also have one winding for each phase but no center taps. This gives four connections, two per phase. In bi-polar stepper motors, current flows bi-directionally. The voltage's polarity is reversed to change the magnetic flux poles. This reversal of polarity requires complex driving circuitry using eight MOSFETs however, modern ICs and power control devices made it easier to control the voltage's polarity and the complexity of controlling bi-directional stepper motors ceases to be a major issue. Bi-directional motors utilize all the windings which mean more efficiency and more output torque. In this project, speed of the self-balancing robot is not much of a concern as the output torque, so, bi-directional stepper motors are used. [8]

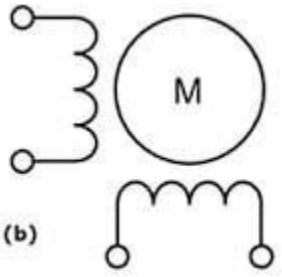

Figure 6: Bi-polar stepper motor. [37]

2.2 MPU-6050:

The MPU-6050 is a micro-electrical-mechanical system, MEMS for short. The MPU contains a 3-axis gyroscope and a 3-axis accelerometer which enables the MPU to measure velocity, orientation, displacement just to name a few. The MPU also contains a digital motion processor (DMP) which is powerful enough to compute complex calculations which can free up some load off the microcontroller. In addition to its low cost, MPU-6050 has many materials, resources and libraries hence very easy to use with Arduino NANO which made it suitable for the self-balancing robot. [9]

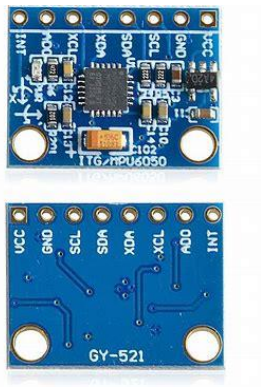

Figure 7: MPU-6050 module. [39]

The MPU-6050 has 8 pins that are listed below:

2.2.1 MEMS Accelerometer

Pin	Name	Function
VCC	Supply voltage	Provides power for the module, Connect to the 5V pin of the Arduino.
GND	Ground	Ground Connected to Ground pin of the Arduino.
SCL	Serial clock	Serial Clock Used for providing clock pulse for I2C Communication.
SDA	Serial data	Serial Data Used for transferring Data through I2C communication.
XDA	Auxiliary serial data	Auxiliary Serial Data - Can be used to interface other I2C modules with MPU6050.
XCL	Auxiliary serial clock	Auxiliary Serial Clock - Can be used to interface other I2C modules with MPU6050.
AD_0	Address	Address select pin if multiple MPU6050 modules are used.
INT	Interrupt	Interrupt pin to indicate that data is available for MCU to read.

MEMS accelerometer measures the linear acceleration of the object they are attached to. Accelerometers work on the principle of the inertia of a mass on a spring. The mass forms a movable plate and there is another fixed plate that forms a capacitor. When the object to which the accelerometer is attached is subjected to linear acceleration, the mass tries to resist the change in its current state which results in inertia. As the mass moves, the distance between the two plates changes thus the capacitance changes. The change in capacitance is proportional to the applied acceleration. The change in capacitance is measured by a high-resolution analog to digital converter and then the acceleration is calculated from the rate of change of capacitance. This is then converted into readable data and sent to the I2C master device (Arduino). [9]

2.2.2 MEMS Gyroscope

The Coriolis effect drives the MEMS gyroscope's operation. The Coriolis effect states that when a mass moving at a specific velocity is exposed to an external angular motion, a force exists that forces the mass to move perpendicularly. The rate of displacement is proportional to the external angular motion. The MEMS gyroscope has four masses that are in constant oscillatory motion. When an angular motion is applied, the Coriolis effect induces a change in capacitance between the masses that is proportional to the axis of the motion. The change in capacitance is detected and turned into a reading.

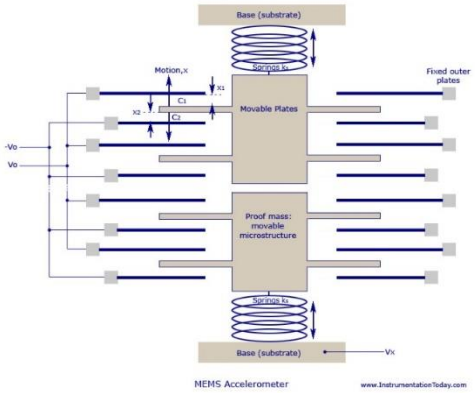

Figure 8: MEMS accelerometer. [38]

2.3 Arduino NANO

Arduino boards are microcontroller boards based on AT-mega microcontrollers, with applications in embedded systems, robotics, and automation. Arduino.cc has produced a variety of Arduino boards, including the UNO, Mega, Micro, and Pro Mini. However, the Arduino NANO will be utilized in this project. The key argument for picking the NANO type is its compact size, which will not take up much space. The Arduino NANO is based on the AT-mega 328P microcontroller. Unlike the Arduino UNO, Arduino NANO doesn't have a DC power jack but is supplied power from a mini- USB port on the board.[10]

To program the Arduino NANO, you must first download an open-source Arduino integrated development environment (IDE) that uses the C++ programming language. Because it is open source, there are several resources available for programming the Arduino NANO, and many programmers have created libraries, which are blocks of code that perform a certain purpose. These libraries were used to program the Arduino NANO and interface the various components, as described in detail in the following sections. [10]

The pin diagram of the Arduino NANO and its functions are described below:[10]

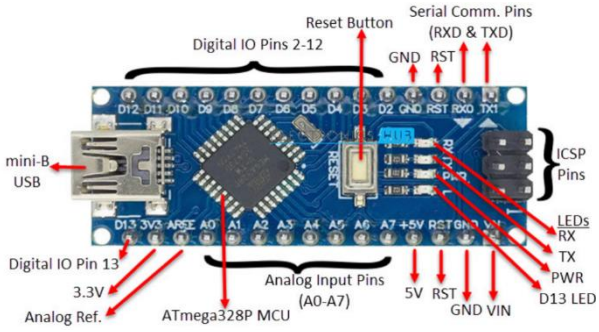

Figure 9: Arduino NANO pin layout. [40]

Pin name	Description	Function
V_{in}	Input volt	Used when an external power source is used from 7V to 12V.
5V	5v	Regulated power supply voltage of the nano board and it is used to give the supply to the board as well as components
3.3V	3.3v	Same function as the 5v
GND	Ground	The ground pin of the board.
RST	Reset	Used to reset the microcontroller.
Pins: A0 to A7	Analog pins	Used to calculate the analog voltage of the board within the range of 0V to 5V.
Pins: D0 to D13	Digital pins	Used as an I/P otherwise o/p pins. 0V & 5V.
T_X and R_X	Receive and transmit	Used to transmit and receive data in serial communication
Pins: 2,3	Interrupts	Used to interrupt the main program and execute certain instructions in case of emergency.
Pins: 3,5,6,9,10,11	PWM	Used to provide 8-bit of PWM output.
Pins: 10,11,12,13	SPI	Used to communicate with sensors and registers that use SPI communication.
Pin: 13	Built in LED pin	Used to activate the LED that indicates that the Arduino is working normally i.e., sending, and receiving data.
A4, A5	I2C protocol	A4 is a bidirectional serial data line (SDA). A5 is Serial clock line SCL
A_{REF}	Reference volt pin	Used to give reference voltage to the input voltage.

2.4 Rechargeable Lithium-ion Battery

2.4.1 Construction

Rechargeable lithium-ion batteries are used in a variety of applications, including mobile phones, laptop computers, MP3 players, and even electric cars. A rechargeable lithium-ion battery consists of one or more compartments called cells. Each cell is made up of a positive and negative electrode, a chemically conductive material known as an electrolyte, and a separator. The positive electrode is often comprised of lithium cobalt oxide or lithium iron phosphate. The negative electrode is often composed of carbon (graphite). The electrolyte substance varies depending on the brand and kind of battery. The separator is a thin sheet of micro-perforated plastic that separates the positive and negative electrodes while still enabling ions to flow through.[11]

Figure 10: Rechargeable lithium-ion battery. [41]

2.4.2 Theory of operation

When the battery charges, the positive electrode releases part of its lithium ions, which travel through the electrolyte to the negative electrode and remain there. During this operation, the battery receives and stores energy. When the battery is discharged, lithium ions pass from the negative electrode to the positive electrode through the electrolyte. During this operation, the battery both discharges and supplies electricity to a load. In both charging and discharging, electrons flow in the opposite direction of ions around the outside circuit rather than through the electrolyte.

2.4.3 Advantages and Disadvantages

Lithium-ion batteries offer various benefits, including the fact that they are more dependable than previous battery technologies such as nickel-cadmium batteries and do not suffer from the memory effect. Memory effect is an issue with nickel-cadmium batteries that requires the battery to be completely depleted before attempting to recharge it again. Lithium-ion batteries are safer for the environment since they do not contain hazardous heavy metals like cadmium. However, lithium-ion batteries have one important disadvantage: thermal runaway. Thermal runway occurs when the battery overheats due to overcharging or when an internal failure results in a short circuit this eventually leads to an explosion or chemical fire, which must be doused with carbon dioxide. However, thermal runaway is an uncommon phenomenon. [11]

2.5 DRV8825 Stepper Motor Driver

There are various motor drivers available commercially. However, it was found that three types of motor drivers in particular are suitable for driving the stepper motor. These drivers are the A4988, DRV8825 and the TMC2208. All these drivers have the same pin diagram. The table below shows the difference between the drivers: [12]

A4988	DRV8825	TMC2208
offers up to 1/16-steps micro-stepping	offers up to 1/32-steps micro-stepping	offers up to 1/256-steps micro-stepping
Has maximum supply voltage of 35 V	has maximum supply voltage of 45 V which make it less affected by LC voltage spikes.	Has maximum supply voltage of 35 V
can deliver 1 A without cooling	can deliver 1.5 A without cooling	can deliver 1.2 A without cooling
Maximum current output of 2A	Maximum current output of 2.5A	Maximum current output of 2A
Cheapest out of the three	Moderate cost	Expensive

Figure 11: Comparison between the three types of drivers

Due to the simplicity of the stepper motor control and the variety of stepping modes provided by the DRV8825 driver, it is the ideal stepper driver for applications that require precise and reliable stepper motor control such as self-balancing robots. The DRV8825 has a drive voltage capacity of 45V. This means that the NEMA17 bi-polar stepper motor can be controlled with up to 2.5A per coil. The output current can be regulated by using a current limiting potentiometer to not exceed the rated current of the stepper motor. The driver contains six modes of stepping which are: full-step, half-step, quarter-step, eighth- step, sixteenth step and thirty-second step. Excessive power dissipation from the driver's IC causes temperature to rise which may lead to irreversible damage to the driver. Therefore, DRV8825 drivers usually come with a heat sink. Although the rated current of the driver is 2.5A per coil, it can only supply 1.5A per coil without overheating. To supply the full rated current, the heat sink must be installed. [12]

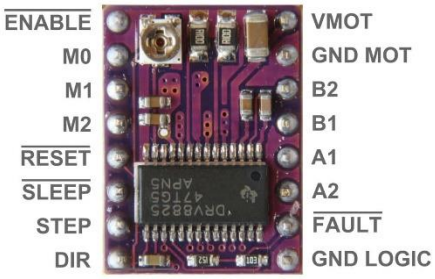

Figure 12: DRV8825 stepper driver pin diagram. [12]

The DRV8825 pin diagram is described below:

P_{in}	Function
V_{MOT}, GND_{MOT}	Supply power to the motor. A capacitor of at least 47µF Should be placed across these two pins to protect against voltage spikes.
GND_{LOGIC}	Connected with the Arduino's ground
M_0, M_1, M_2	Used to pick the one of the six step resolutions.
STEP	Controls the micro-steps of the motor. As the pulse frequency increase, so does the motor's speed.
DIR	Controls the spinning direction of the motor. When it is set HIGH the motor turns clockwise. When it is set LOW, the motor turns anti-clockwise.
EN	Active low input. When it is LOW, the driver is enabled.
SLP	Active low input. When it is LOW, the driver is in sleep mode, reducing the power consumption to a minimum.
RST	Active low input. When it is LOW, all step inputs are ignored.
FAULT	Sets to LOW whenever there is an over-current protection or thermal shutdown.
B_1, B_2, A_1, A_2	Output pins connected to the 4-wires of the stepper motor.

2.6 Components Interfacing

Once we were acquainted with the main components, we proceeded to conduct a series of tests to familiarize ourselves with their operation, ensure proper functionality, and identify any potential limitations or drawbacks. In total, we conducted four tests, which are detailed below:

2.6.1 Stepper motors test

The motors were programmed to rotate one full turn in a clockwise direction at low speed, followed by two full turns in a counterclockwise direction at high speed, with this sequence repeating. To achieve this, we familiarized ourselves with the fundamental programming functions of Arduino, which enabled us to control the speed and direction of the stepper motors. Additionally, we learned how to correctly wire the stepper motors to the Arduino module and motor driver.

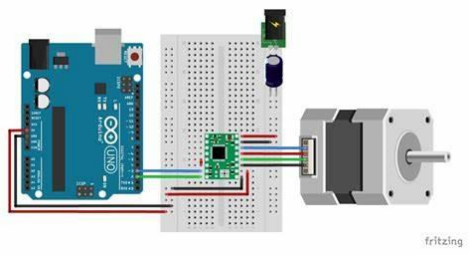

Figure 13: Stepper motor connected to motor driver and Arduino

This is the code of our experiment:

```
// Connections to A4988
const int dirPin = 2;                    // Direction
const int stepPin = 3;                   // Step
const int STEPS_PER_REV = 200;           // Motor steps per rotation
```

Figure 14: Set step and direction pins to pin 3 and pin 2

```
void loop() {
  // Set motor direction clockwise
  digitalWrite(dirPin, HIGH);
  // Spin motor one rotation slowly
  for (int x = 0; x < STEPS_PER_REV; x++) {
    digitalWrite(stepPin, HIGH);
    delayMicroseconds(2000);
    digitalWrite(stepPin, LOW);
    delayMicroseconds(2000);
  }
  // Pause for one second
  delay(1000);
```

Figure 15: Code of clockwise one revolution at low speed

```
  // Set motor direction counterclockwise
  digitalWrite(dirPin, LOW);
  // Spin motor two rotations quickly
  for (int x = 0; x < (STEPS_PER_REV * 2); x++) {
    digitalWrite(stepPin, HIGH);
    delayMicroseconds(1000);
    digitalWrite(stepPin, LOW);
    delayMicroseconds(1000);
  }
  // Pause for one second
  delay(1000);
```

Figure 16: Code of counterclockwise two revolution at high speed

2.6.2 Acquisition of MPU-6050 Readings

During this test, we effectively utilized the mpu-6050 to measure the angles in a three-dimensional plane relative to a set point determined by the gyroscope when positioned on a horizontal plane. However, we initially encountered an issue while attempting to install the mpu-6050 on the breadboard and supplying it with power proved to be challenging. Nevertheless, we were able to address this issue by soldering the pins into the gyroscope.

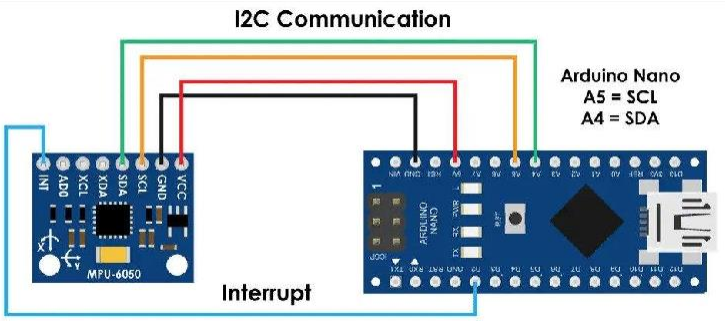

Figure 17: Connection of MPU6050 and Arduino NANO

The code we used in this experiment is in Arduino IDE examples and called MPU_DMP6.

2.6.3 Using MPU-6050 As a Level Meter

In this experiment, we utilized the gyroscope as a level meter, with readings from the MPU- 6050 transmitted to the Arduino. The Arduino was programmed to activate specific LED lights when the MPU-6050 reading entered a particular range. To accomplish this, we employed five LED lights, with the middle LED illuminating when the angle was between -10 to 10 degrees. The LED to the right lit up when the angle was between 10 to 20 degrees, while the far-right LED lit up when the angle exceeded 20 degrees. The same methodology was applied to the left and far-left LEDs, but with negative degree values.

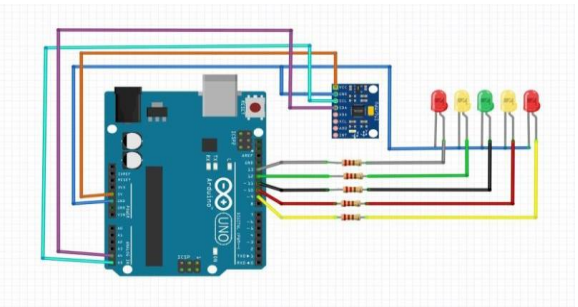

Figure 18: MPU6050 as a level meter

```
// Check Angle for Level LEDs

  if (angle_pitch_output < -6.01) {
  // Turn on Level LED
  digitalWrite(levelLED_neg1, HIGH);
  digitalWrite(levelLED_neg0, LOW);
  digitalWrite(levelLED_level, LOW);
  digitalWrite(levelLED_pos0, LOW);
  digitalWrite(levelLED_pos1, LOW);

  } else if ((angle_pitch_output > -6.00) && (angle_pitch_output < -3.01)) {
  // Turn on Level LED
  digitalWrite(levelLED_neg1, LOW);
  digitalWrite(levelLED_neg0, HIGH);
  digitalWrite(levelLED_level, LOW);
  digitalWrite(levelLED_pos0, LOW);
  digitalWrite(levelLED_pos1, LOW);

  } else if ((angle_pitch_output < 3.00) && (angle_pitch_output > -3.00)) {
  // Turn on Level LED
  digitalWrite(levelLED_neg1, LOW);
  digitalWrite(levelLED_neg0, LOW);
  digitalWrite(levelLED_level, HIGH);
  digitalWrite(levelLED_pos0, LOW);
  digitalWrite(levelLED_pos1, LOW);
```

Figure 19: Code of using MPU-6050 as a level meter

2.6.4 Integrating MPU-6050 With Stepper Motor

Building on the previous experiment, we utilized the MPU-6050 as a level meter and incorporated the stepper motor in addition to the LED lights. The motor responded to different angle values, with no action taken when the angle was between -10 to 10 degrees. When the angle was between 10 and 20 degrees, the motor rotated in a counter-clockwise direction, and when the angle exceeded 20 degrees, the motor rotated counter-clockwise at a higher speed. The same sequence was repeated for negative angle values, but the motor rotated clockwise. This experiment emulated the robot's efforts to maintain balance, with the MPU-6050 functioning as a sensor and the stepper motor as an actuator.

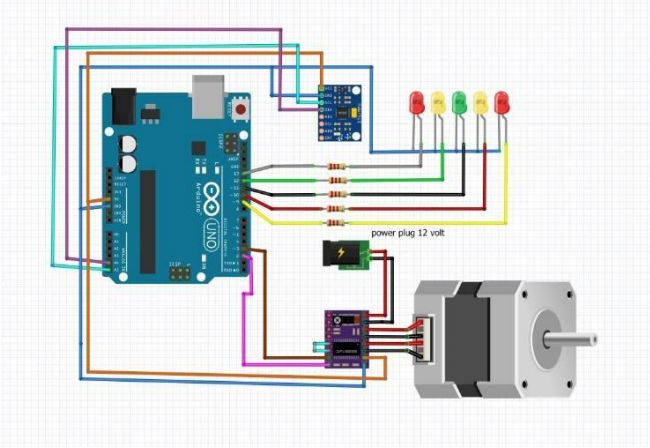

Figure 20: MPU as a level meter with stepper motors

Chapter 3 Mechanical Structure

Self-balancing robots are gaining popularity for their ability to maintain stability on two wheels. The mechanical structure is crucial, as it must support the robot's weight and provide flexibility for balance. The structure includes a chassis, two wheels, a motor, battery, and sensors. Sensors detect changes in orientation, used to adjust the motor for balance. The chassis stabilizes the wheels and motor, while the battery powers them. The mechanical structure is essential to performance, and this overview provides insight into designing, building, and operating self-balancing robots.

3.1 Structure Requirements

Designing the mechanical structure of the self-balancing robot is an important stage in our project, as if it isn't correct, it will affect negatively the balance of the self-balancing robot. So, the design of the mechanical structure has some key requirements will be discussed below:

- **Light Weight:** the material of the mechanical structure should have light weight for two reasons
    - in order not to overload stepper motors.
    - to minimize energy required from batteries so the lifetime of the batteries increased.
- **Weight distribution:** the weight of the self-balancing robot should be evenly distributed across its center to maintain stability of the robot.
- **Robustness:** the structure of the robot should be robust enough to withstand stresses and external forces acting on it.
- **Formability:** the robot isn't formed as one block it consists of parts, so the material of the structure of the robot should be formable to be shaped and cut into parts without losing its stability.
- **Cost:** The cost of the materials is a crucial consideration and should be appropriate for the project's needs and the developer's budget.

3.2 Material Used: Plywood

Choosing the material of the mechanical structure is a crucial factor in the design as it affects the stability, lightness of weight, robustness of the structure. There are many materials used to construct self-balancing robots such as plastic, plywood, acrylic, etc. After searching between the materials used in designing the mechanical structure, we construct this table which shows a brief review of some materials' pros and cons.

Casing materials	Advantages	Disadvantages
Metal [14]	Metals provide extra protection from mechanical shocks and damage.	Metals are heavy and their weight will be a great challenge to overcome as it will negatively affect the stability of the robot.
Plywood [15]	Plywood is stronger than MDF and can support heavier weights.	Harder to moulid.
medium- density fiber board (MDF) [15]	MDFs are easier to moulid and cheaper than plywood	Can't support heavier weights and are easy to damage.
Plastics [14]	Plastics have light weight; therefore, they don't need too much energy to move the robot.	it can be vulnerable or fragile against mechanical strikes or rough terrain. Plastics are not usually used for the outer casing on the robot, rather, they are used as layers where PCBs, components, etc. are placed.

After constructing this table, we found that plywood is the best choice in our project which is a type of wood that is commonly used in the construction of self-balancing robots. There are several reasons why plywood is a popular choice for this application:

- Light bulk density
- high strength
- beautiful texture
- insulation
- small deformation
- cost-effective, Although its cost is higher than MDF

Figure 21: Different thickness of plywood sheets

3.2 System Estimation

Once we had identified the requirements for designing the mechanical structure of the self-balancing robot and selected acrylic as the casing material, our next step was to determine if the torque generated by the motor would be sufficient to support the weight of the robot. To accomplish this, we utilized an equation to estimate the required torque based on estimated values for the robot's parameters.

$$T_{max} = mg*l*\sin(\theta_{max}) \quad [1.0]$$

$$T_{max} = 0.8*9.8*0.05*\sin(5^{o}) = 0.03417 \text{ Nm} \quad [1.1]$$

Where:

- T_{max} is the maximum torque produced by the motor.
- m is the total mass of the robot.
- g is the constant of gravity.
- l is the height of the center of mass.
- θ is the maximum recovery angle, we choose such a small recovery angle since the dynamics of the system will be linearized to calculate the control gains of the system [16].

There are two motors on the robot so, the maximum torque produced by the motor should be 0.0171 Nm (half of the maximum torque required by the system). Based on our estimations and using a stepper motor with a torque of 0.38 Nm, we determined that the motor's torque will be sufficient to support the weight of the robot's body.

3.3 Mechanical Structure Design 'SOLIDWORKS'

Once the PCB has been designed and its required dimensions have been calculated, the mechanical structure is designed using SOLIDWORKS. SOLIDWORKS is a popular software tool used by mechanical engineers and designers to create detailed designs of self-balancing robots.

We used SOLIDWORKS software to design the three main parts of the robot's body which are the back side, the top & bottom part and the two lateral sides combined together so that the body is almost like a one-side opened box.

The figures below show the dimensions of each part:

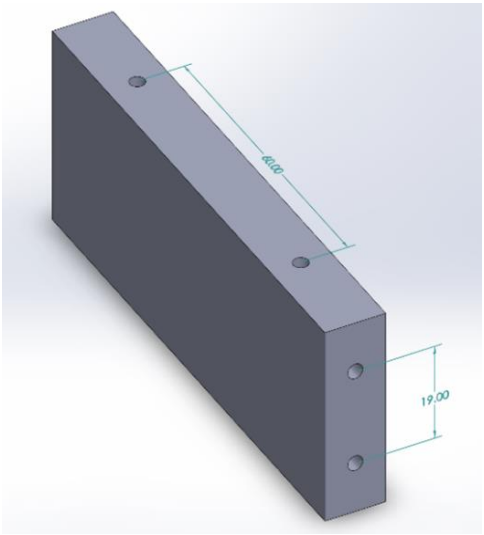

Figure 22: First part of self-balancing robot (Top layer)

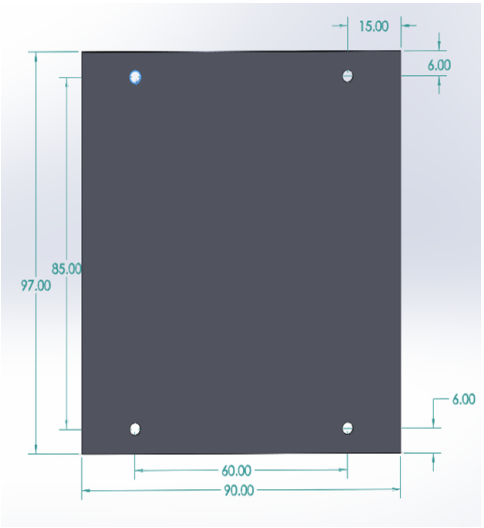

Figure 23: Second part of self-balancing robot (Back layer)

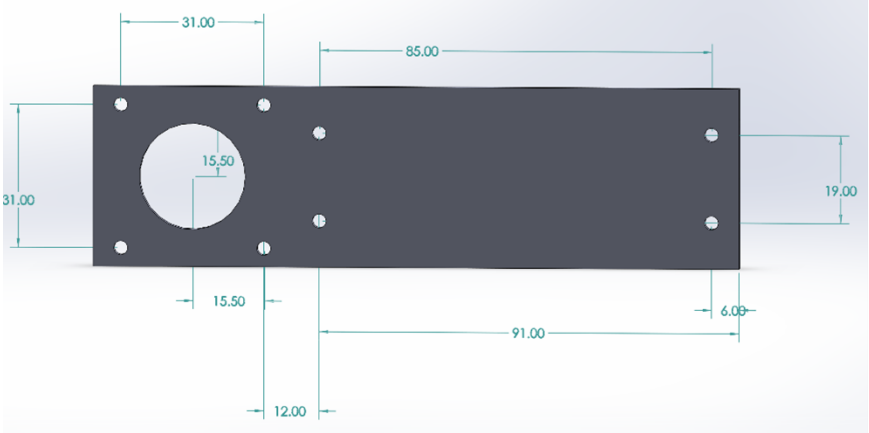

Figure 24: Third part of self-balancing robot (Side layer)

The top and bottom sides have higher thickness than the other sides to be able to endure more mechanical stresses. Also the back side is made only to carry the PCB, so we made sure it is thin enough and not so wide as it is present on one side only and not on the other in order to maintain the uniform distribution of weight keeping the center of gravity of the robot in the center.

The figure below shows the final shape after connection parts together:

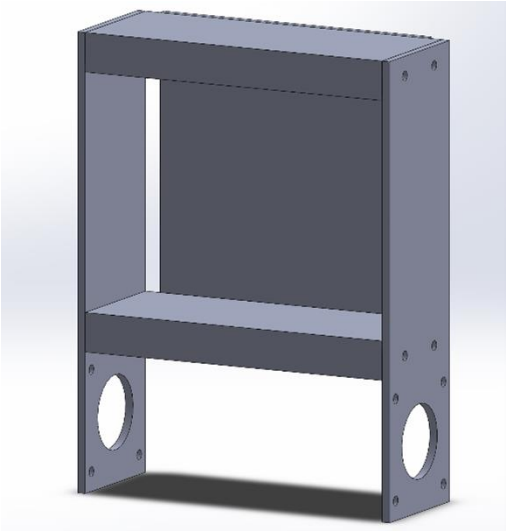

Figure 25: 3-D connected parts

3.4 Mechanical Structure CNC Fabrication

Once we completed the design of the mechanical structure, we began researching the most appropriate fabrication method for our project. We considered two options: 3D printing and laser cutting (CNC). as part of our investigation into the optimal fabrication approach for the robot's mechanical structure casing, we have compiled the subsequent table:

Manufacturing methods for casing:	Advantages	Disadvantages
Laser cutting: [17]	**a.** Laser can cut through all materials. **b.** Works with high accuracy and precision.	**a.** Not fit for complex multidimensional geometry. **b.** Produce harmful fumes during production.
3-D printing: [18]	**a.** Flexible and accurate design. **b.** Doesn't need specific tools or several tools to execute design.	**a.** Materials such as wood cannot be 3D printed because it will burn before they can be melted and extruded through a nozzle.

Based on the information gathered from the table, we concluded that Laser cutting is the optimal choice for our project's fabrication needs. The figures below illustrate the Laser cutting process and the resulting final parts of the robot's body.

Figure 29: Laser cutting process

Figure 27: Final printed ply wood parts

Figure 28: Connected parts (Back view)

Figure 26: Connected parts (Front view)

3.6 Connecting Parts

Figure 30: Main parts

This section will showcase some of the components utilized in our self-balancing robot.

The self-balancing robot parts is connected with stainless steel screws and nuts shown in the figures. stainless steel screws have diameters of 5mm and 1cm in length (the best we found on the current market).

Figure 32: Small stainless steal screws used

Figure 31: Small stainless steal nuts used

3.7 Choosing the Wheels

Wheels are an integral component of self-balancing robots, as they provide the necessary movement and stability for the robot to operate efficiently. In the following sections, we will explore some of the key considerations when designing and constructing wheels for self-balancing robots such as:

- **Light weight:** It is advantageous for the wheels of a self-balancing robot to be lightweight. This is because lighter wheels contribute to the overall reduction of the robot's weight, which is essential for maintaining stability and balance. Additionally, lighter wheels require less power to rotate, which can lead to improved battery life and increased operational efficiency.
- **Stability:** When the wheels are rotating, they should maintain the same speed and direction as the motor shaft without any deviation. It is crucial to ensure stability as the wheels are responsible for transferring control action from the motor shafts to maintain or restore the robot's balance. Therefore, stability is the most critical requirement for wheels.
- **High friction surface**: The outer surface of the wheel should be crafted from a high-friction material, such as rubber, to ensure a firm grip on the surface on which the robot will travel. This will help maintain traction and prevent slipping, ensuring the robot's smooth movement.
- **Uniformity**
- **Good Coupling with the motor shaft**

Wheel type	Advantages	Disadvantages
Normal Type	Most common type. Two degrees of freedom. Wide range of sizes and mounts.	Not fit for Off-Road expeditions. Best fit for smooth terrain.
Off Road wheels	A variation of the tire wheel. Two degrees of freedom. Fit for rough environments.	Not fit for precision movement. Expensive in comparison to the alternatives.
Mecanum wheels	Multidirectional wheel Fit for indoors and outdoors environments. Different left and right wheels.	precision Is a concern. Not that many sources available with such use case (self-balancing robot).
omni wheels	Wheels with rollers at 90 degrees. Four degrees of freedom. Fit for indoors environments.	Inapplicable for a self-balancing robot. Only works for 4 wheels configs. Expensive and hard to find with the current market.

Once we had a clear understanding of the wheel requirements, we conducted market research to identify several types of wheels that would meet our needs. The resulting table outlines the wheels we found:

The figures below show types of wheels illustrated in previous part:

Figure 36: Off road wheels

Figure 35: Mecanum wheels

Figure 34: Normal wheels

Figure 33: Omni wheels

Upon conducting our market research, we have selected a wheel that meets all the requirements (**Normal wheels**) . We have chosen two 65x26mm wheel. The weight of each wheel is 50 GM.

Chapter 4 Electric Circuit Design

4.1 Schematic Design

To design an electric circuit diagram for the robot, a software called fritzing was used due to its simplicity and having a relatively easy to learn curve. It's also the main program we used to design the PCB of the robot.

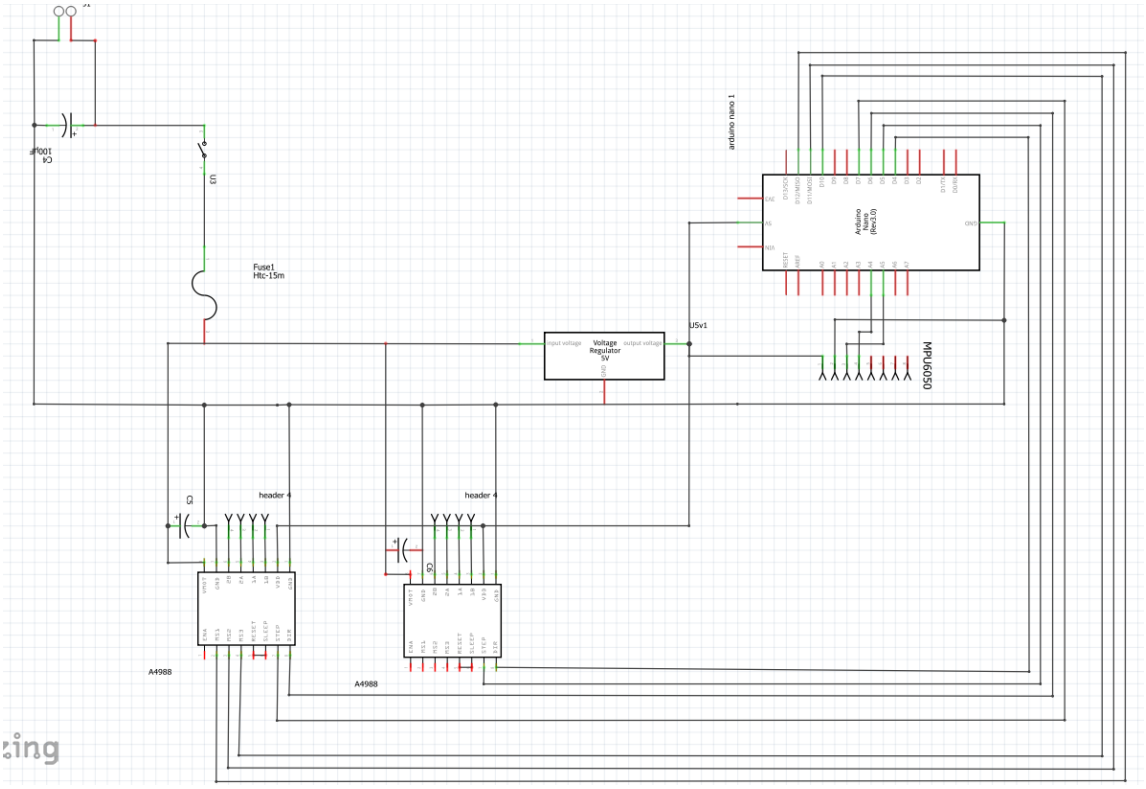

Figure 37: SBR schematic diagram

Our electrical system is a combination of:

- 1x Arduino NANO board.
- 2x DRV8825 stepper motor drivers.
- 1x MPU-6050 Gyroscope.
- 1x 12V-to-5V voltage regulator.
- Multiple protection and regulating elements (diodes, resistors, fuse, capacitors, …etc.).

The step and direction pins of the drivers are connected to the digital pins (D4, D5, D6, D7) preferably, the step pins are connected to the analog out capable digital pins for more smooth and accurate output. However, any other pin can work just fine. The micro-stepping pins on the drivers (MS0, MS1, MS2) were connected to digital pins (D10, D11, D12) together to make sure that both motors are running on the same micro-stepping resolution. The drivers output pins (A2, A1, B1, B2) were connected to 4 female pin headers for easier connection to the stepper motor coils. Finally, the power pins for the logic of the driver VDD and GND are connected from a +5V voltage regulator outputted from the Arduino and the motor power pins VMOT and GND are connected from the external power supply taking into consideration the capacitors between these two pins to eliminate any unwanted voltage spikes.

For the MPU-6050 the connections are similar to earlier chapters where the VCC and GND pins are connected from the voltage regulator while the SCL and SDA pins are connected to the analog pins A5 and A4 in the Arduino, respectively. Only A5 and A4 pins will work as they are designed to be responsible for the I2C protocol.

For the regulator, the input voltage is directly fed from the system 12V supply after the protection layers, the ground is the same as the supply, and the output 5V are connected the MPU-6050 and the Vin pin on the Arduino to insure no overvoltage spikes to the main components.

The system includes five layers of protection for the power circuit to help counter any potential mishaps, these protections include:

- a 2200uF capacitor to counter input supply voltage ripples.
- a Switch to turn on and off the system.
- A diode to counter reverse polarity (in case of reverse polarity of the input supply).
- A DC fuse to protect against over current (up to 3 Amps).
- A voltage divider circuit to check for the current supply voltage output.

4.2 Breadboard Test Setup

The next step was to test out the system output relative to the design schematic to ensure the system reliability to provide the expected outputs.

Using breadboards helps during the testing phase to indicate connections issues and for easier replacement of any broken or unwanted connections before finalizing the system design.

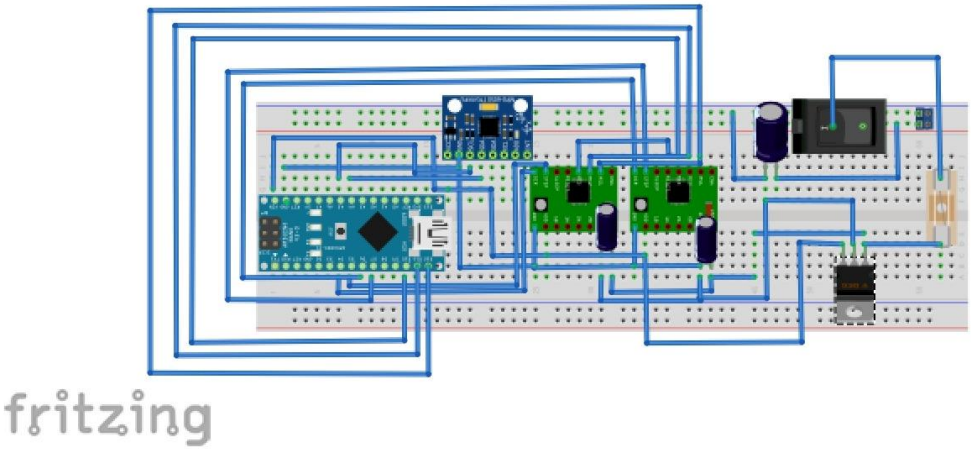

Figure 38: Breadboard schematical representation

However, the breadboard is not the best option for the finished system due to its lack of stable connections and bad mounting to sensitive components like the MPU-6050 as any small tilt in the module will lead to an offset error in the readings that will be translated to an inaccurate decision by the PID controller.

4.3 Printed Circuit Board (PCB) Design

The PCB was designed using fritzing using the finalized schematic from the testing phase in order to eliminate any possible human error (in terms of connections and mounting issues).

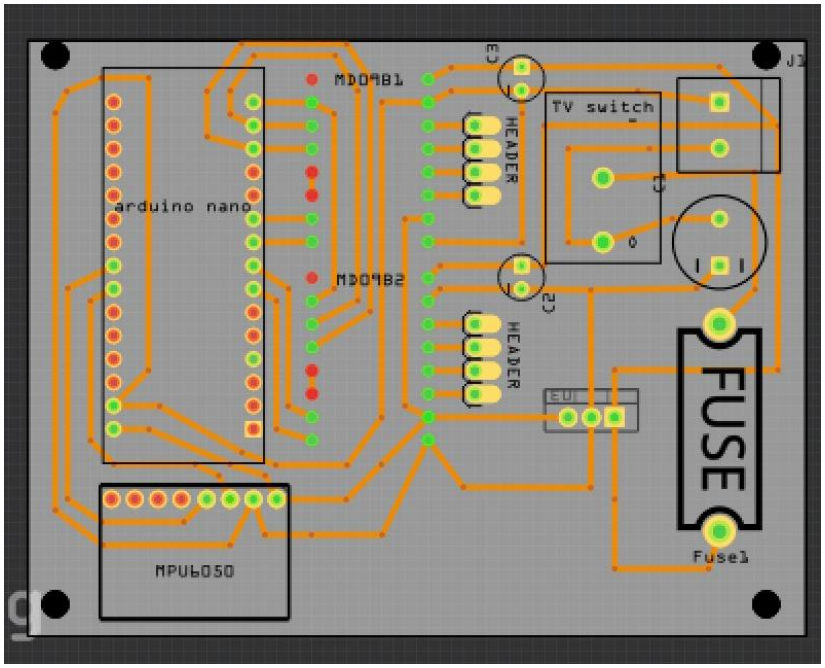

Figure 39: PCB final design inside fritzing software

The material used is FR4 sheets as they are electrical insulators with high dielectric strength [19]. They also feature a high strength-to-weight ratio and are lightweight and resistant to moisture. Add this to their relative temperature resistance, and FR4 material can perform well in most environmental conditions [19].

The PCB dimensions are 85mm by 65mm (55.25cm^2). The dimensions were made relative to the mechanical structure design to ensure a more centered weight distribution.

All control circuit traces are made with trace width of 1.5 mm to withstand up to 2 Amps of current with the copper layer thickness of 2 oz/ft^2. While the power traces width is 2 mm to withstand 3-4 Amps. There is a clearance between each trace of about 0.5mm to insure no shorting between traces during the printing stage.

The PCB was plated with soldering mask to protect the system against possible shorts between close pins during the soldering phase.

The traces routing was made automatically using the fritzing software to find the best copper route, sometimes the auto tracing might fail due to bad positioning of the elements of the circuit, with slight repositioning the auto router can finish routing successfully.

The software is capable of producing 2D and 3D images to preview the expected PCB output from the printing stage.

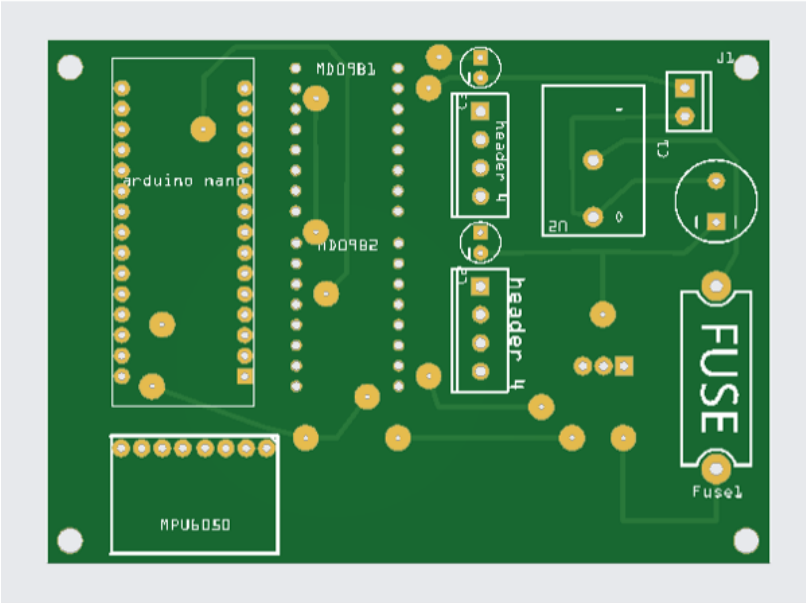

Figure 40: 2d image of the expected PCB output from the printing stage.

After finalizing the design, a GERBER file is generated using the software to send off to a PCB manufacturing company to be printed (usually takes around 2 – 15 business days).

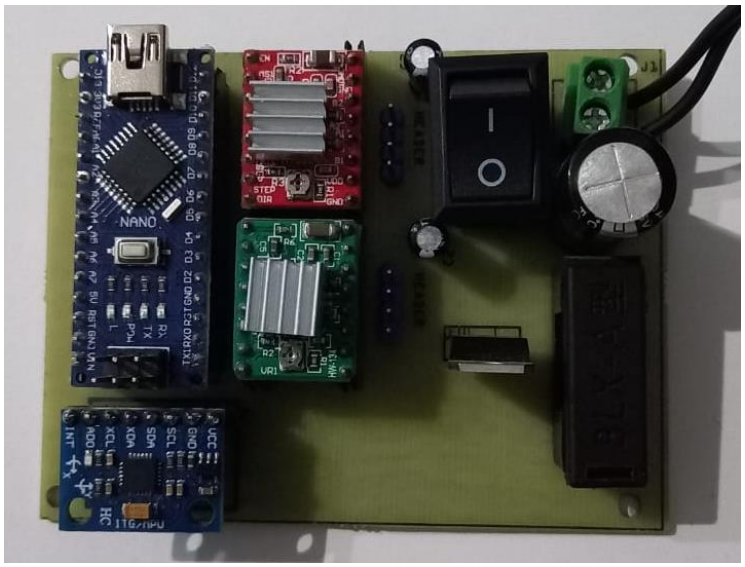

Figure 41: PCB after assembly

Chapter 5 How to Balance a Self-Balancing Robot

5.1 Mechanical Requirements and Limitations

Figure 42: Our self-balancing robot

5.1.1 Weight And Torque Limitations

The weight of the robot should not be too heavy or light. When a control action is used to balance the robot, if it is too light, it may become unstable and respond aggressively. However, if the robot is too heavy, the torque of the steppers may be insufficient to sustain the weight, and it may not stabilize even in the absence of external forces.[20]

Therefore, it is important to calculate the robot's weight before selecting the motors to ensure that the chosen motors can provide sufficient torque to balance the robot [21].
Using Newton's Law:

$$Tinertial = Tgravity + Tapplied$$

$$Tinertial = mgLsin\theta + Tapplied$$

$$Then\ Tapplied = 0$$
$$Tinertial = mgLsin\theta$$

$$t_{max} = mgLsin\theta_{max}$$

$$where\ \theta_{max} = 90°$$

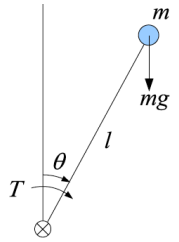

Figure 43 : Effect of robot weigh on torque supplied by motor

When we start the robot, it should normally stabilize with the steppers torque at 0° (vertically) after calibrating the gyro at 90° (horizontal). We should also consider a safety factor in the torque calculation by 1.5 – 2 to make sure that the torque is enough to stabilize the robot under rough circumstances.[22]

5.1.2 Height

The weight of the robot isn't the sole aspect influencing its stability. The robot's body length also affects its stability. If the body length is excessively large, the robot may respond violently to the control action, making it unsuitable for torque calculations [21].

This is because a tall robot has a higher center of mass, making balancing more challenging. Furthermore, a taller robot may have longer legs or wheels, increasing the distance between the center of mass and the point of contact with the ground, making it less stable.

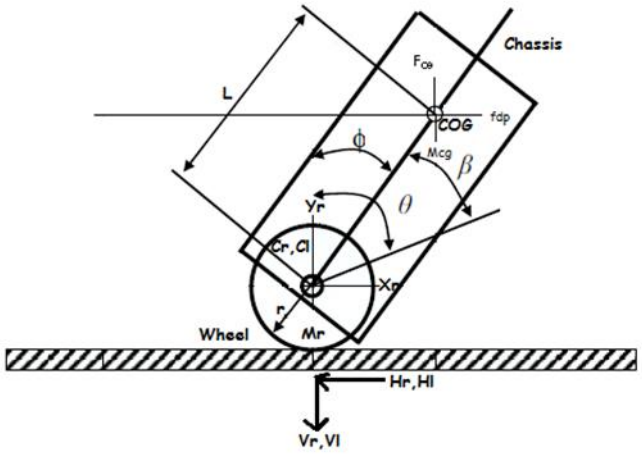

Figure 44: Effect of robot's height on robot's balance

As a result, while constructing a self-balancing robot, it is critical to consider not just the weight but also the robot's dimensions, such as height, width, and length, to guarantee optimal stabilization. A shorter robot, with a lower center of mass and a shorter distance between the center of mass and the contact point with the ground, will be more stable and easier to control. When deciding on the robot's dimensions, it is also vital to consider its intended use. For example, a taller robot with bigger wheels or legs may be more suited to navigating tough terrain or obstacles.

5.1.3 Wheels

The diameter and kind of wheels used in a self-balancing robot can significantly influence its balance and stability [23]. In prior trials, we discovered that utilizing standard wheels with a tiny diameter was ineffective for balancing the robot since they could not even bring the robot to a stop. This is because the wheels' narrow diameter resulted in a low contact area with the ground, making it difficult for the robot to maintain balance.

In order to solve this issue, we switched to larger off-road wheels. These wheels provided a wider contact area with the ground, improving the robot's stability and allowing it to move easily across uneven surfaces. The bigger diameter also enhanced the torque provided by the wheels, making it simpler for the robot to maintain balance.

Therefore, when designing a self-balancing robot, it is important to consider the diameter and type of wheels used to ensure that they can provide sufficient torque and contact area with the ground to stabilize the robot. The choice of wheels will depend on the application of the robot, as well as the terrain it will be operating on.

5.1.4 Center of Gravity

The center of gravity is a crucial factor in stabilizing a self-balancing robot. If the center of gravity is low (near to the wheels), it might produce imbalance, as we saw with our three-racked-body robot. Because of their high weight and low center of gravity, balancing the robot with three racks was problematic at first. However, we were able to enhance the balance by rearranging the racks and putting the battery to the top rack. This centralized the robot's heavy weight at the top, shifting the center of gravity away from the wheels and making weight balance simpler to maintain [21].

To increase the robot's stability, we shortened the space between the top and middle racks. This raised the center of gravity and improved the robot's stability. We were able to create a sturdy and well-balanced robot by optimizing the rack arrangement and center of gravity.

Therefore, when designing a self-balancing robot, it is important to carefully consider the placement of the robot's components and ensure that the center of gravity is positioned in an optimal location for stability. The location of the center of gravity will depend on the weight and distribution of the robot's components and may need to be adjusted during the design process to achieve optimal stability.

5.1.4.1 Center of Gravity Calculations

as the self-balancing robot is based on the concept of an inverted pendulum, there are some important parameters that must be calculated:

- Weight of the cart
- Weight of pendulum
- And the center of gravity (COG) of the pendulum

For the self-balancing robot, we can consider the hinge of the pendulum the wheel axes. This makes the wheels, the cart and everything else the pendulum.

The center of gravity is simply the average location of the weight of the robot's body.

Calculating the center of gravity for a system

The center of gravity of the robot can be calculated by taking the center of gravity of the individual components and doing vector addition of the weighted position vectors. For each of the axis, x, y and z, the center of gravity is found with:

$$COG_x = \frac{m_1 \cdot x_1 + m_2 \cdot x_2 + m_3 \cdot x_3 + \ldots}{m_1 + m_2 + m_3 \ldots}$$

$$COG_y = \frac{m_1 \cdot y_1 + m_2 \cdot y_2 + m_3 \cdot y_3 + \ldots}{m_1 + m_2 + m_3 \ldots}$$

$$COG_z = \frac{m_1 \cdot z_1 + m_2 \cdot z_2 + m_3 \cdot z_3 + \ldots}{m_1 + m_2 + m_3 \ldots}$$

Where:
m_1, m_2, m_3: are the weight of components 1, 2 and 3 making up the system.

x_1, y_1, z_1: are the coordinates of the center of gravity for the first component.

x_2, y_2, z_2: for the second

x_3, y_3, z_3: for the third

Determining the center of gravity of a self-balancing robot can be challenging due to its many components with unknown centers of gravity. However, we can apply the same techniques used for motor vehicles to determine the center of gravity of the robot. We neglect the center of gravity in the x- and y-directions, as they are symmetrical and do not affect the mathematical model. Instead, we focus on the center of gravity in the z-direction and calculate its height.

For motor vehicles, the center of gravity is often determined by weighing the vehicle at each wheel and measuring the wheelbase. We can apply the same techniques to the self- balancing robot to determine its center of gravity. This method involves weighing the robot at two points, one at the rotation axis of the motors and the other as close to the top as possible and measuring the distance between them. The weight and distance measurements can then be used to calculate the height of the center of gravity above the rotation axis [24].

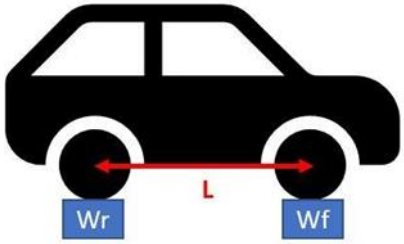

Figure 45: In a motorcar, the weight of the vehicle at each wheel and measuring the wheelbase.

With these measurements we have the car's total weight W which we can model as a point mass at the center of gravity.

$$W = W_r + W_f$$

Since the car is not tipping over, it can be estimated that the center of gravity (COG) is located somewhere between the front and rear axles of the car. Additionally, since the car is not moving, according to Newton's laws, the scales are applying a force equal to the weight to W_r and W_f of the car, acting in the opposite direction. Let us call these forces R_f and R_r

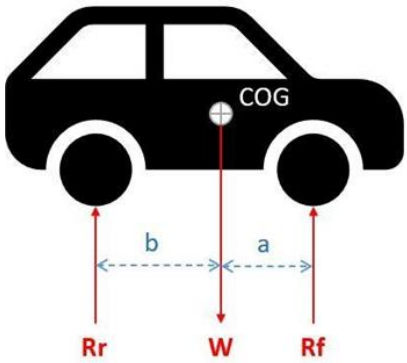

Figure 46: COG is located somewhere between the front and rear axles of the car.

The longitudinal COG of the car can be determined by considering the moment about the rear wheel. Since the car is not rotating around this point, the moments resulting from the weight W at the COG and the force Rf at the front wheels cancel each other out. Mathematically, this can be written as:

$$0 = b \cdot W - L \cdot R_f$$

From here we can make b the subject and thus determine the center of gravity in the longitudinal direction.

$$b = \frac{L \cdot R_f}{W}$$

Applying to the equation to the robot

- When calculating the center of gravity of the robot, we remove the wheels and only weigh the pendulum part to simplify the calculations.
- To weigh the robot, we select two points at the bottom and top of the robot separately. One point is chosen at the rotation axis of the motors to simplify the process, while the other point is chosen as close to the top as possible. The distance between these two points is then measured to determine the distance L.
- We determine the weight of the robot by adding the weights of the two chosen points at the bottom (W_r) and top (Wf) of the robot. This method provides sufficient accuracy for our purposes.
- For our calculations, we define the top of the robot as corresponding to the front of the car and the bottom as corresponding to the rear.
- Using the measurements, we obtained; we can now enter them into the equation to calculate the height of the center of gravity (b) above the rotation axis of the robot.

It's important to note that accurate measurements and calculations are crucial for determining the center of gravity of a robot, as even small errors can have a significant impact on its stability and performance. Therefore, it's important to take care and double-check all measurements and calculations to ensure accurate results.

$$b = \frac{L \cdot R_f}{W}$$

$$W = W_r + W_f$$

Where:

W : total weight of the robot.

W_r : the weight at the bottom of the robot.

W_f : The weight at the top of the robot.

R_f: the force at the front wheels.

Chapter 6 Control Method: PID Controller

The control algorithm used to balance the position of the self-balancing robot is the Proportional Integral Derivative (PID) controller, which is a widely used feedback mechanism in the industry. The PID controller is a three-term controller that attempts to adjust and correct the error between the measured process and the desired process and output the appropriate measure to adjust the process.
The PID controller calculates an error value e(t) as the difference between a desired set point and a measured process variable. It then corrects the error using proportional, integral, and derivative terms. The controller attempts to minimize the error over time by adjusting a control signal. [25]

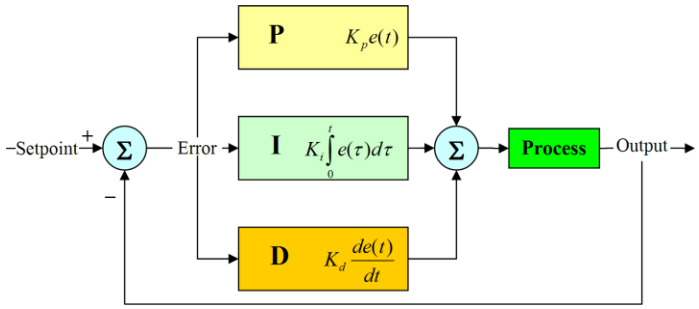

Figure 47: Block diagram of PID controller

6.1 PID Controller in Our Application:

By applying the PID controller terms on our project, in the figure, it is shown self-balancing robot in two main positions: balanced state and tilted position.

The error is calculated by getting the difference between the actual tilting angle sensed by the MPU 6050 and the desired set point tilt angle which will be in our case 0°. The controller generates the suitable control action and sends it to the motors to minimize the error and balance the robot. [26]

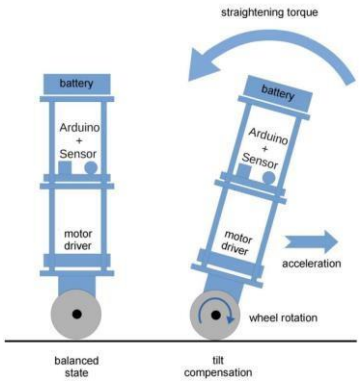

Figure 48: The control mechanism used in self-balancing robot

The PID controller parameters are tuned to find the suitable gains to balance the system.

Step 1: setting the three gains to zero. Then, we increase the K_p value gradually until the system starts to oscillate by then we adjust the K_p value until the system is nearly stable with a steady state error.

Step 2: we increase the value of K_d gradually until the system starts to vibrate when applying a high disturbance in short period of time. Then, we start adjusting the K_d till the system oscillations disappear. However, having a high K_d value may cause overshoot.

Step 3: we increase K_i value gradually until any offset or steady state error is eliminated and corrected. Usually, K_i values are small because when its value increases the system is more prone to instability. [26]

The PID controller use the following equation:

$$u(t) = Kp\, e(t) + Ki \int_0^t e(t)d(t) + Kd \frac{de(t)}{dt}$$

Where:

- K_p is proportional gain.

- K_i is integral gain.

- K_d is derivative gain.

- $u(t)$ is the control action generated by the PID controller.

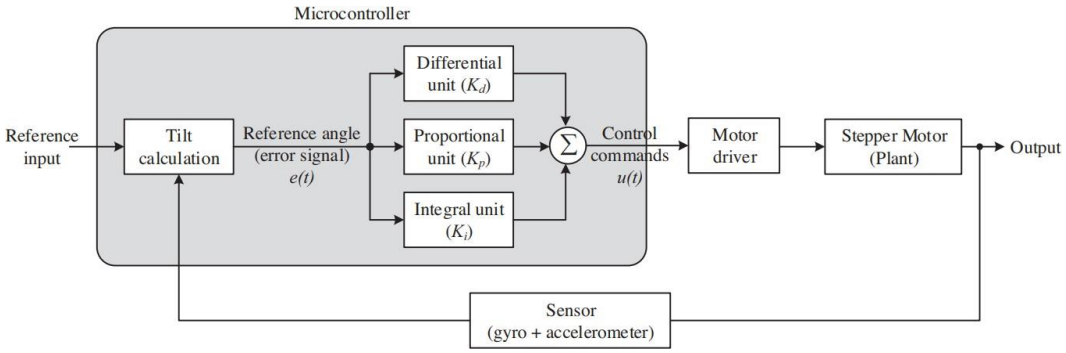

Figure 49: The closed loop control system of the self-balancing robot. [27]

The angle set points where the robot has to maintain its default position and PID values are fed to the Arduino. Kp is the proportional term, which multiplies with the error and speeds-up or slows-down the response. Any change in Kp makes the bot to balance at another set-point.

K_{i} is the integrating term, which reduces the steady-state error and smoothens the motion of the robot. But a slight change in this integrating factor may cause excessive response to the system.

K_{d} is the derivative term. It is used to increase the reaction time of the robot. However, increasing it too much may cause the robot to be unstable. [27] [25]

6.2 Tuning Techniques

There are some tuning techniques that can be used to find the best gain values of K_p, K_i and K_d that gives a stable response of the system.

The tuning technique used was **trial and error** method which is the most common to start with to get an approximate value of the needed gains. [28]

6.2.1 Trial and Error

Parameter	*Rise Time*	*Overshoot*	*Settling Time*	*Steady-State error*	*Stability*
K_p	Decrease	Increase	Small Change	Decrease	Degrade
K_{i}	Decrease	Increase	Increase	Eliminate	Degrade
K_{d}	Minor Change	Decrease	Decrease	No effect in theory	Improve

We also tried an alternative configuration that helped reduce overshoots and oscillations in the system. This was achieved by multiplying the gain controller by the output instead of using the PID in a conventional way, which involves calculating the error and multiplying it by the gain. This method acts as a filter to the system, allowing the robot to balance faster than before and reducing the settling time. Furthermore, this approach reduces oscillations and is particularly effective since the setpoint in our system is not zero, which means that the robot is slightly inclined to move. By reducing the overshoot resulting from the error and the oscillations, this method helped improve the overall performance of the system. [29]

6.3 Mechanical System Simulink Model

To build a 3D mechanical model for the two-wheeled self-balancing robot we use Simulink, to assess the robot and tune it in a safe way before implementing it on hardware also a mechanical model is used to estimate the system's transfer function using the system identification toolbox.

In this model we use the PID controller to get auto-tuned values or trial and error values to balance the system. [30] [31]

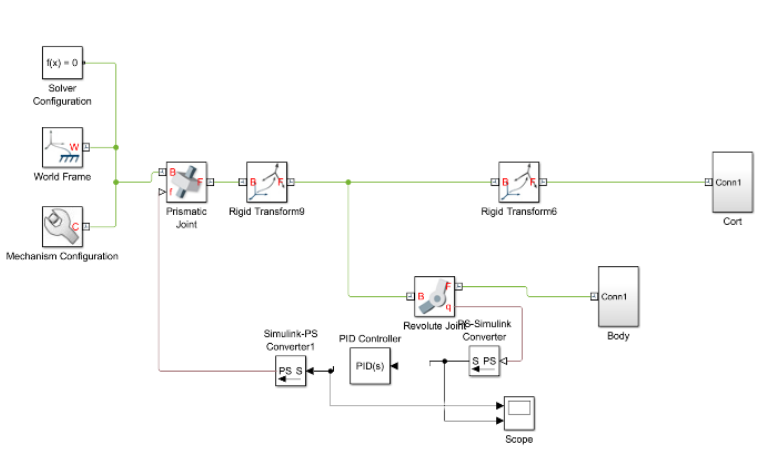

Figure 50: The mechanical system of a two-wheeled self-balancing robot

6.3.1 Blocks Definition

Solver configuration: it is used to perform the calculations of the system.

World frame: This block represents the global reference frame in a model. Directly or indirectly, all other frames are defined with respect to the World frame as it is the ultimate reference frame.

Mechanism configuration: This block specifies the gravity and simulation parameters of the mechanism to which the block connects.

Rigid transform: It is a block used to connect solid blocks with each other's so they can move as a single unit during simulation.

Revolute joint: This block gives a degree of freedom to the motion of the robot's body, and it will allow the chassis to swing like a pendulum.

Prismatic joint: This block will allow the body to move back and forth and stabilizing the body in an upright pose with a suitable control action. [31] [30]

6.3.2 Steps to Build a Model

We start building the model using MATLAB, typing **simnew** in the command window to display a **SIMSCAPE** multibody model template equipped with the needed parts to start building the model.
We start by building the wheels of the robot, we open the library to get the element responsible for wheels:

- Library
- SIMSCAPE
- Multibody
- Body elements.

We get the cylindrical solid block and we adjust its dimensions and mass to build the tires and wheels [32] [30]

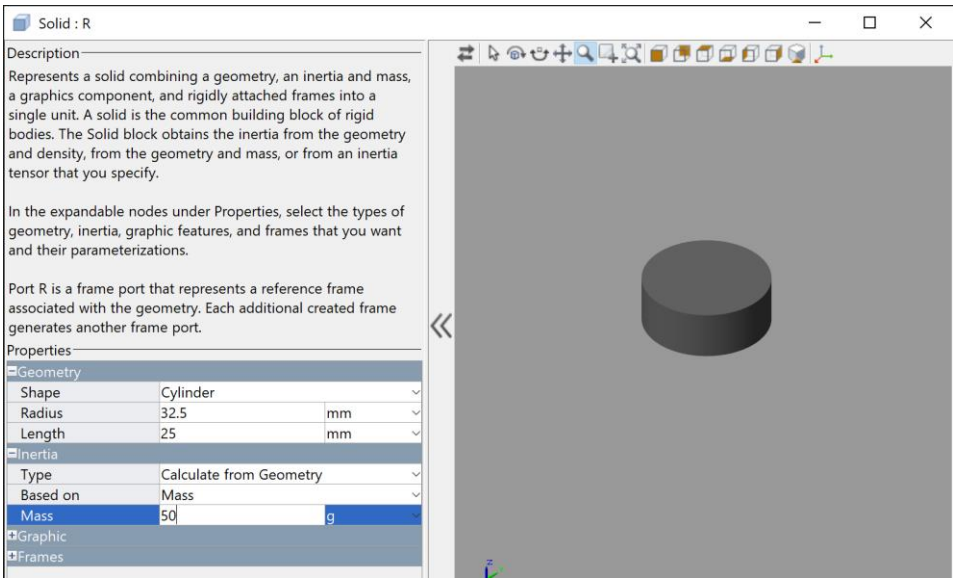

Figure 51: Setting the body element dimensions to simulate the real-life wheel.

To make the right wheel, we repeat the same steps.

Then we connect both wheels with motors in cart subsystem.

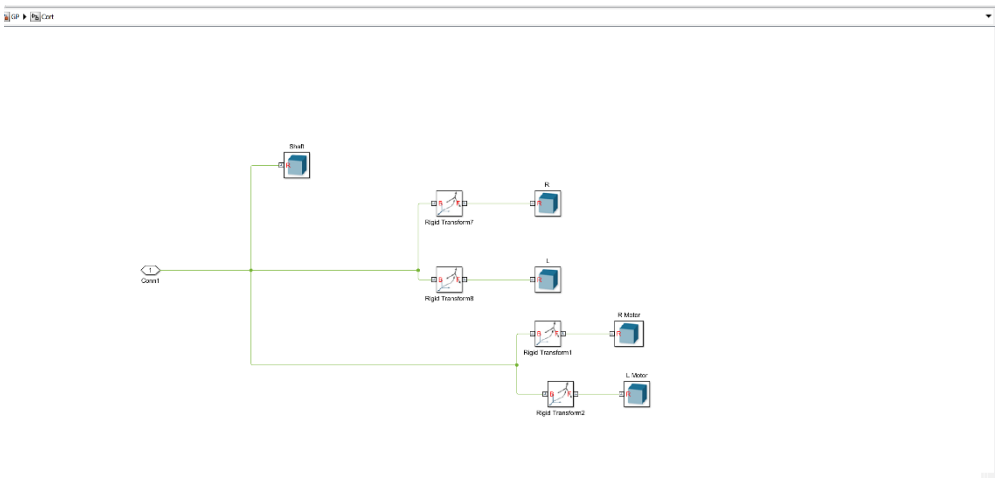

Figure 52: Cart subsystem where there are the two motors and two wheels.

Then, we build the robot's body which consists of five parts attached together, we start building the left , right sides , top , bottom and the back using the brick solid block and adjusting its dimensions properly and their mass.

To link the parts together, we use the rigid transform block by adjusting where the parts will connect together then we put them all in a subsystem to represent the body. [32] [30]

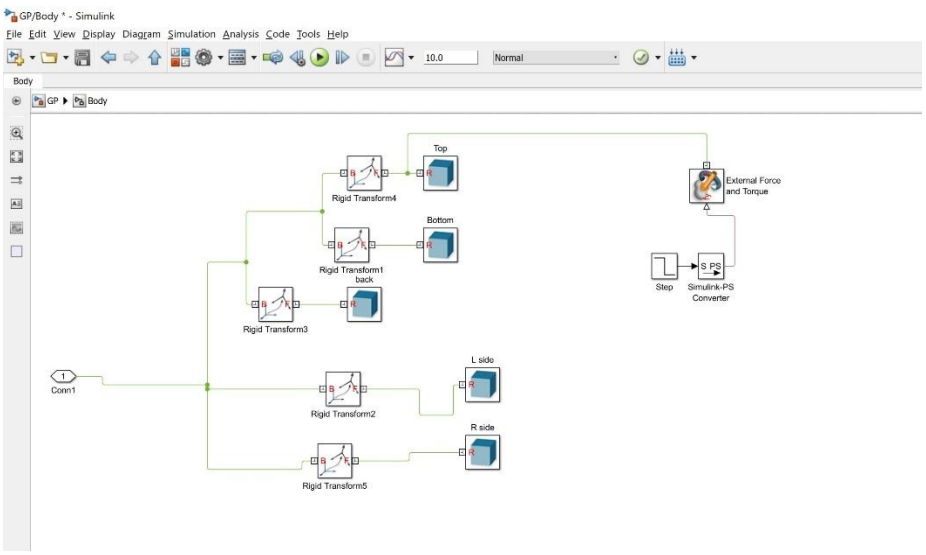

Figure 53: Body subsystem and the connection of parts together using the rigid transform block.

We connect the body and the cart with a revolute joint block to make the robot's body swing like a pendulum and then we connect the prismatic joint block this will make the robot move back and forth and with a proper control action coming from the PID controller, it will eventually stabilize the body.

We read the angle of inclination of the body provided by the position of the revolute joint, so we change its sensing setting to read the position, also, we set the force in the prismatic joint block to be provided as an input and the motion to be automatically computed.
We use a gain to convert the output position angle from radians to degree.

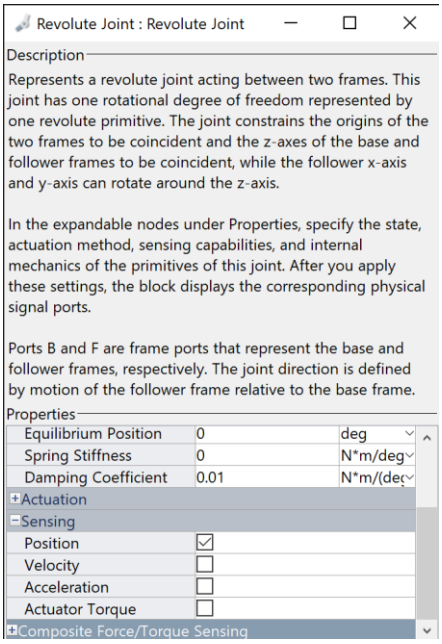

Figure 55: Revolute joint block setting its sensing to position

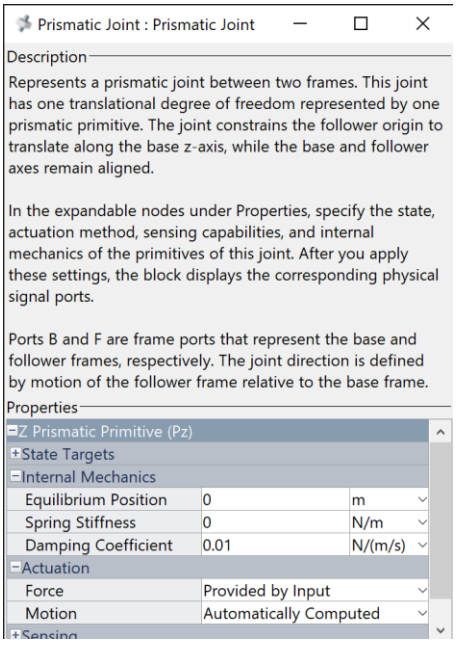

Figure 54: Prismatic joint block setting its actuation force to be provided as an input.

We use a step block to give an initial inclination for the robot instead of applying an external force.

We will then adjust the PID block using an autotune feature to get a suitable PID values to balance the robot in the simulation. [32] [30]

6.4 System parameters estimation

This toolbox is used in order to estimate the parameters of PID controller from the mechanical system created using Simscape.. [31]

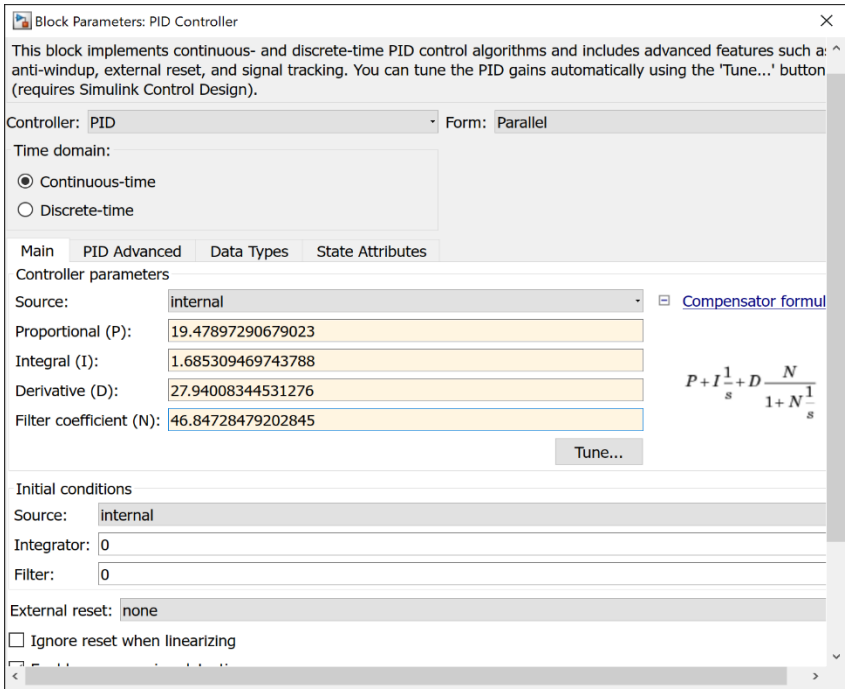

Figure 56: Setting the PID block to a proportional controller.

After using system identification toolbox in MATLAB we obtained initial values of PID controller then we reached the final estimate using trial and error based on our initial values.

The following figure shows the system model we used in MATLAB SIMSCAPE tool:

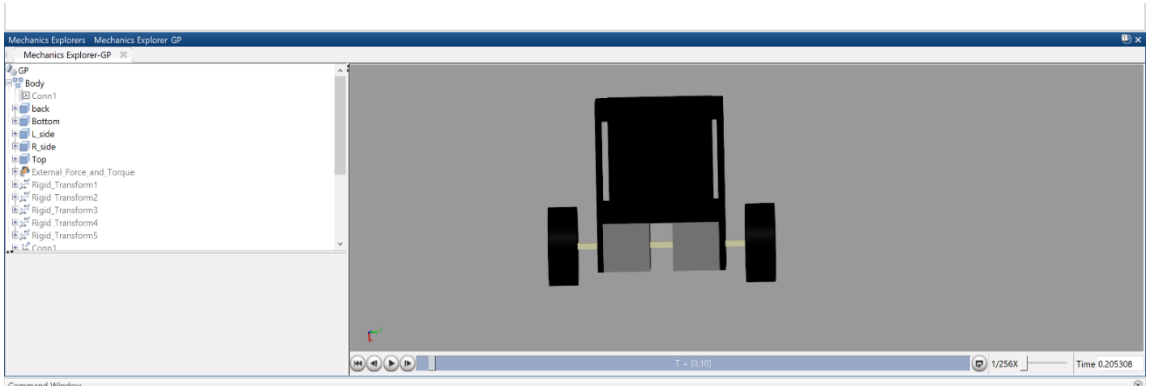

Figure 57: System model by SIMSCAPE.

While this figure shows the response obtained from the system after tunning using MATLAB system identification toolbox

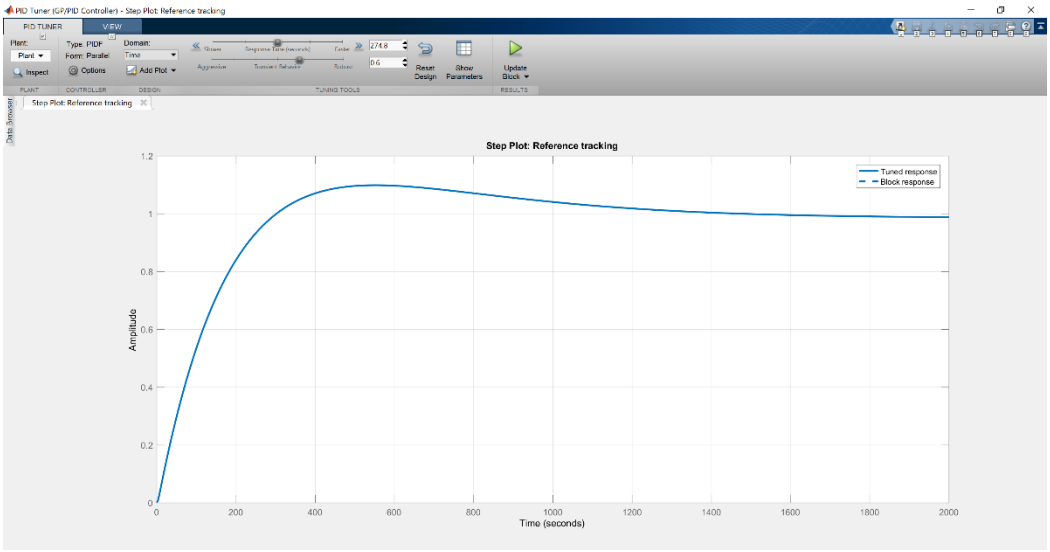

Figure 58: System response after tunning.

Chapter 7 Arduino Codes Used This Project

All the codes used here can be found in the appendix section. This section is mainly focused on explaining the code's main idea and how the main parts work.

7.1 MPU-6050 Calibration

This code is used in order to acquire the approximate gyro values (x, y, z) for the MPU to indicate that the robot is currently in an upright position.

The robot is placed on a stand in an upright position then the hardware test is uploaded. The code checks if there are any I2C devices connected and checks if any of these devices is an MPU-6050.

The program will output several raw gyro values as well as the balance value. The balance value is used in the void loop to be used in angle calculations.

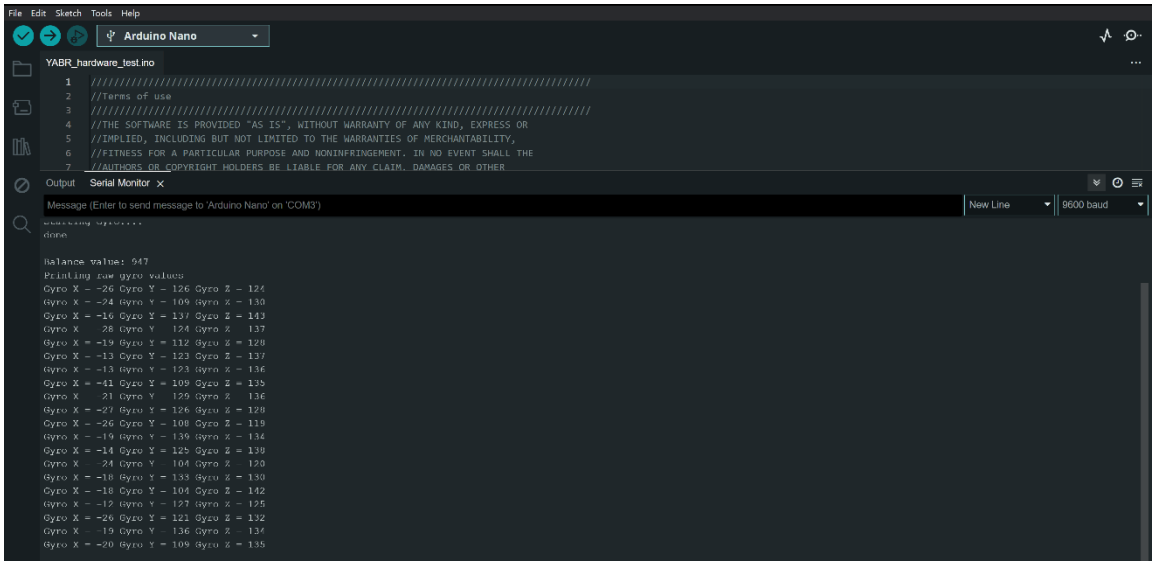

Figure 59: The MPU calibration code serial monitor's output

These values are later combined into a single value (Balance value) that is later used in the main code calculations to set up a balance point for the PID controller to approach.

It's worth noting that the value can exist with a marginal error with the range of 100 points (up or down) and thus it advised to run the code multiple times and average the acquired balance values for a more accurate set point.

7.2 Hardware Troubleshooting Tests Code

The aim for this code was to test each component's individual functionality to help during any troubleshooting phase. This code provides the user with 4 tests for each individual component, which are:

- Battery voltage indicator to help identify the current voltage level of the source.
- Bluetooth module connection test with minor control actions (forward, reverse, turn left, turn right)
- An MPU level meter test to insure the MPU is still working.
- A small forward-reverse cycle to insure the correct operation of the motors.

After uploading the code, the Arduino runs a check to see the current battery voltage of the input through an analog pin. If the input is smaller than 10.5 volts, the robot won't start any other test while generating a pulsating clock that flickers a LED on the Arduino to indicate low input voltage.

After choosing the MPU test (sending an integer value = 5), the system starts to function as a level meter where the angular orientation of the MPU dictates the motor's motion speed and direction. If with changing the position and angle of the MPU no speed or direction change is observed, then there is a problem related to the system gyroscope.

After choosing the motor test (sending an integer value = 6), the system starts cycling in a forward and reverse directions. If the two motors didn't move in the same direction as each other, then there is an issue of reverse connection of one of the motors.

If the motors didn't function throughout the entire tests without being in the off state, then a problem related to the motors, or the drivers is indicated.

While it may not be the most robust code for troubleshooting, it provides basic visual indications of any issue that may appear during the testing phases of the self-balancing robot. A further improvement for the code can be made by adding text indicators on the serial monitor in case of failure in any test or a morse code signal using the built-in led off the Arduino microcontroller.

7.3 Main Code of The Self-Balancing Robot

The main code consists of 3 main parts:

- The initial parameters and PID gains.
- The void setup and interruption sub routine.
- The main void loops.

7.3.1 The Initial Parameters and PID Gains

In this section, the user inputs the main parameters acquired by the user through system identification that defines the system limits. These parameters include:

- Maximum target speed and turning speed.
- Accelerometer calibration value (balance value).
- PID controller gains value.
- The gyro bus address and wire library.

```
#include <Wire.h>  //Include the Wire.h library so we can communicate with the gyro

int gyro_address = 0x68;           //MPU-6050 I2C address (0x68 or 0x69)
int acc_calibration_value = 1044;  //needs calibration                          //Enter the accelerometer calibration value

//Various settings
float pid_p_gain = 25;                                                  //Gain setting for the P-controller (25)
float pid_i_gain = 1.1;                                                 //Gain setting for the I-controller (1.1)
float pid_d_gain = 30;                                                  //Gain setting for the D-controller (30)
float turning_speed = 15;                                               //Turning speed (15)
float max_target_speed = 50;                                            //Max target speed (50)

```

Figure 60: PID controller gains value.

Then the global variables are declared that will be used throughout the code.

```
16  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
17  //Declaring global variables
18  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
19  byte start, received_byte, low_bat;
20
21  int left_motor, throttle_left_motor, throttle_counter_left_motor, throttle_left_motor_memory;
22  int right_motor, throttle_right_motor, throttle_counter_right_motor, throttle_right_motor_memory;
23  int battery_voltage;
24  int receive_counter;
25  int gyro_pitch_data_raw, gyro_yaw_data_raw, accelerometer_data_raw;
26
27  long gyro_yaw_calibration_value, gyro_pitch_calibration_value;
28
29  unsigned long loop_timer;
30
31  float angle_gyro, angle_acc, angle, self_balance_pid_setpoint;
32  float pid_error_temp, pid_i_mem, pid_setpoint, gyro_input, pid_output, pid_last_d_error;
33  float pid_output_left, pid_output_right;
34
35  SoftwareSerial myserial(2, 3);
```

Figure 61: Global variables

7.3.2 The Void Setup and Interruption Sub Routine

Starting the void setup, the serial port is set to 9600 kbps and the I2C bus is set as master and initializing the I2C bus clock at 400kHz.

```
finalBal.ino
40  void setup() {
41    Serial.begin(9600);  //Start the serial port at 96
42    Wire.begin();        //Start the I2C bus as master
43    TWBR = 12;           //Set the I2C clock speed to
44    myserial.begin(9600);
45
```

Figure 62: Starting void setup

Then in order to generate a variable pulsating control signal for the stepper motor a sub- routine is made that needs to be executed every 20ms.

```
45
46  //To create a variable pulse for controlling the stepper motors a timer
47  //This subroutine is called TIMER2_COMPA_vect
48  TCCR2A = 0;               //Make sure that the TCCR2A register is set to
49  TCCR2B = 0;               //Make sure that the TCCR2A register is set to
50  TIMSK2 |= (1 << OCIE2A);  //Set the interupt enable bit OCIE2A in the TI
51  TCCR2B |= (1 << CS21);    //Set the CS21 bit in the TCCRB register to se
52  OCR2A = 39;               //The compare register is set to 39 => 20us /
53  TCCR2A |= (1 << WGM21);   //Set counter 2 to CTC (clear timer on compare
54
```

Figure 63: Variable pulsating control signal generation

Then we set up the MPU parameter ranges to its maximum scale for more accurate readings by modifying the internal registries of the MPU using the wire library.

```
56  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search.
57  Wire.write(0x6B);                      //We want to write to the PWR_MGMT_1 register (6B hex)
58  Wire.write(0x00);                      //Set the register bits as 00000000 to activate the gyro
59  Wire.endTransmission();                //End the transmission with the gyro.
60  //Set the full scale of the gyro to +/- 250 degrees per second
61  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search.
62  Wire.write(0x1B);                      //We want to write to the GYRO_CONFIG register (1B hex)
63  Wire.write(0x00);                      //Set the register bits as 00000000 (250dps full scale)
64  Wire.endTransmission();                //End the transmission with the gyro
65  //Set the full scale of the accelerometer to +/- 4g.
66  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search.
67  Wire.write(0x1C);                      //We want to write to the ACCEL_CONFIG register (1A hex)
68  Wire.write(0x08);                      //Set the register bits as 00001000 (+/- 4g full scale range)
69  Wire.endTransmission();                //End the transmission with the gyro
70  //Set some filtering to improve the raw data.
71  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search
72  Wire.write(0x1A);                      //We want to write to the CONFIG register (1A hex)
73  Wire.write(0x03);                      //Set the register bits as 00000011 (Set Digital Low Pass Filter to ~43Hz)
74  Wire.endTransmission();                //End the transmission with the gyro
```

Figure 64: Setting MPU parameter ranges

The digital pins connected to the stepper motor drivers are then set as output with the pins connected to the micro-stepping pins are set to output a 1/4 micro-stepping degree.

```
76  pinMode(4, OUTPUT);  //Configure digital poort 2 as output
77  pinMode(5, OUTPUT);
78  pinMode(6, OUTPUT);  //Configure digital poort 3 as output
79  pinMode(7, OUTPUT);
80  pinMode(10, OUTPUT);
81  pinMode(11, OUTPUT);  //Configure digital poort 4 as output
82  pinMode(12, OUTPUT);  //Configure digital poort 5 as output
83                        //Configure digital poort 6 as output
84  digitalWrite(10, HIGH);
85  digitalWrite(11, HIGH);
86  digitalWrite(12, HIGH);
```

Figure 65: Setting digital pins connected to the stepper motors as outputs

A for loop is made to run for 500 loops throughout which the MPU takes multiple readings for the Yah and Pitch angles to identify its current position relative to its surrounding area. Then the output calibrated values are divided by 500 to average out the gyro offset. Finally, a loop timer is initialized to start at the end of the next cycle.

```
87
88    for (receive_counter = 0; receive_counter < 500; receive_counter++) {  //Create 500 loops
89                                                                           // if(receive_counter % 15 == 0)digitalWrite(13, !digitalRea
90      Wire.beginTransmission(gyro_address);                                //Start communication with the gyro
91      Wire.write(0x43);                                                    //Start reading the Who_am_I register 75h
92      Wire.endTransmission();                                              //End the transmission
93      Wire.requestFrom(gyro_address, 4);                                   //Request 2 bytes from the gyro
94      gyro_yaw_calibration_value += Wire.read() << 8 | Wire.read();        //Combine the two bytes to make one integer
95      gyro_pitch_calibration_value += Wire.read() << 8 | Wire.read();      //Combine the two bytes to make one integer
96      delayMicroseconds(3700);                                             //Wait for 3700 microseconds to simulate the main program lo
97    }
98    gyro_pitch_calibration_value /= 500;  //Divide the total value by 500 to get the avarage gyro offset
99    gyro_yaw_calibration_value /= 500;    //Divide the total value by 500 to get the avarage gyro offset
100
101   loop_timer = micros() + 4000;  //Set the loop_timer variable at the next end loop time
102 }
103
```

Figure 66: MPU takes multiple readings for the Yah and Pitch angles to identify its current position

The interruption subroutine is used to throttle the stepper motor speed resulted from the PID controller outputs to the left and right motor to minimize the impact of the nonlinear behavior of the stepper motors.

```
289 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
290 //Interrupt routine  TIMER2_COMPA_vect
291 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
292 ISR(TIMER2_COMPA_vect) {
293   //Left motor pulse calculations
294   throttle_counter_left_motor++;                                  //Increase the throttle_counter_left_motor variable by 1 every time this routine is executed
295   if (throttle_counter_left_motor > throttle_left_motor_memory) {  //If the number of loops is larger then the throttle_left_motor_memory variable
296     throttle_counter_left_motor = 0;                              //Reset the throttle_counter_left_motor variable
297     throttle_left_motor_memory = throttle_left_motor;             //Load the next throttle_left_motor variable
298     if (throttle_left_motor_memory < 0) {                         //If the throttle_left_motor_memory is negative
299       PORTD &= 0b11101111;                                        //Set output 3 low to reverse the direction of the stepper controller
300       throttle_left_motor_memory *= -1;                           //Invert the throttle_left_motor_memory variable
301     } else PORTD |= 0b00010000;                                   //Set output 3 high for a forward direction of the stepper motor
302   } else if (throttle_counter_left_motor == 1) PORTD |= 0b00100000;  //Set output 2 high to create a pulse for the stepper controller
303   else if (throttle_counter_left_motor == 2) PORTD &= 0b11011111;    //Set output 2 low because the pulse only has to last for 20us
304
305  //right motor pulse calculations
306   throttle_counter_right_motor ++;                                    //Increase the throttle_counter_right_motor variable by 1 every time the routine is executed
307   if(throttle_counter_right_motor > throttle_right_motor_memory){     //If the number of loops is larger then the throttle_right_motor_memory variable
308     throttle_counter_right_motor = 0;                                 //Reset the throttle_counter_right_motor variable
309     throttle_right_motor_memory = throttle_right_motor;               //Load the next throttle_right_motor variable
310     if(throttle_right_motor_memory < 0){                              //If the throttle_right_motor_memory is negative
311       PORTD |= 0b01000000;                                            //Set output 5 low to reverse the direction of the stepper controller
312       throttle_right_motor_memory *= -1;                              //Invert the throttle_right_motor_memory variable
313     }
314     else PORTD &= 0b10111111;                                         //Set output 5 high for a forward direction of the stepper motor
315   }
316   else if(throttle_counter_right_motor == 1)PORTD |= 0b10000000;      //Set output 4 high to create a pulse for the stepper controller
317   else if(throttle_counter_right_motor == 2)PORTD &= 0b01111111;      //Set output 4 low because the pulse only has to last for 20us
318 }
```

Figure 67: Interruption subroutine to adjust stepper motor speed

7.3.3 The main void loops

The main loop requires at least 4ms to execute each command available in the loop before repeating the execution.

Figure 68: The main loop

After the initial checks, two bytes are then requested from the MPU to set the starting acceleration speed of the loop. Then we add the calibrated value to specify how far are we from the correct position in order to balance. In order to avoid later division by zero, a limiter condition is used.

Then using the resulted values, we calculate the current pitch angle of the robot. If the accelerometer angle is almost zero. The main PID controller loop starts to operate in this loop.

```
//Angle calculations
Wire.beginTransmission(gyro_address);                          //Start communication with the gyro
Wire.write(0x3F);                                              //Start reading at register 3F
Wire.endTransmission();                                        //End the transmission
Wire.requestFrom(gyro_address, 2);                             //Request 2 bytes from the gyro
accelerometer_data_raw = Wire.read() << 8 | Wire.read();       //Combine the two bytes to make one integer
accelerometer_data_raw += acc_calibration_value;               //Add the accelerometer calibration value
if (accelerometer_data_raw > 8200) accelerometer_data_raw = 8200;    //Prevent division by zero by limiting the acc data to +/-8200;
if (accelerometer_data_raw < -8200) accelerometer_data_raw = -8200;  //Prevent division by zero by limiting the acc data to +/-8200;

angle_acc = asin((float)accelerometer_data_raw / 8200.0) * 57.296;  //Calculate the current angle according to the accelerometer

if (start == 0 && angle_acc > -0.5 && angle_acc < 0.5) {  //If the accelerometer angle is almost 0
  angle_gyro = angle_acc;                                 //Load the accelerometer angle in the angle_gyro variable
  start = 1;                                              //Set the start variable to start the PID controller
}
```

Figure 69: Angle calculations

After that a calculation is made to figure out the current traveled distance during the execution of this loop for later use.

```
147
148   Wire.beginTransmission(gyro_address);                //Start communication with the gyro
149   Wire.write(0x43);                                    //Start reading at register 43
150   Wire.endTransmission();                              //End the transmission
151   Wire.requestFrom(gyro_address, 4);                   //Request 4 bytes from the gyro
152   gyro_yaw_data_raw = Wire.read() << 8 | Wire.read();  //Combine the two bytes to make one int
153   gyro_pitch_data_raw = Wire.read() << 8 | Wire.read(); //Combine the two bytes to make one int
154
155   gyro_pitch_data_raw -= gyro_pitch_calibration_value; //Add the gyro calibration value
156   angle_gyro += gyro_pitch_data_raw * 0.000031;        //Calculate the traveled during this loo
157
158   ////////////////////////////////////////////////////////////////////////////////////////////////
```

Figure 70: Calculating the current traveled distance

After completing the gyro angle calculations and corrections, the PID controller calculations start. A variable is used to store the PID error, then a small brake function is used to lower the PID output from the last loop. After which, the Ki controller gain is calculated and then limited to the maximum control output. Then the PID controller output is calculated using the gain values we have from the initial steps while also making sure to avoid overshooting the control signal using a limiter function. A small deadband function is used to stop the motors when the PID controller output approaches the balance state of the robot.

A small safety protocol is made to turn off the motors and reset the PID memory to 0 in case the tips over or the battery is low or the robot fails to start.

```
172   ////////////////////////////////////////////////////////////////////////////////////////////////////
173   //PID controller calculations
174   ////////////////////////////////////////////////////////////////////////////////////////////////////
175   //The balancing robot is angle driven. First the difference between the desired angel (setpoint) and actual angle (process value)
176   //is calculated. The self_balance_pid_setpoint variable is automatically changed to make sure that the robot stays balanced all the time.
177   //The (pid_setpoint - pid_output * 0.015) part functions as a brake function.
178   pid_error_temp = angle_gyro - self_balance_pid_setpoint - pid_setpoint;
179   if (pid_output > 10 || pid_output < -10) pid_error_temp += pid_output * 0.015;
180
181   pid_i_mem += pid_i_gain * pid_error_temp;  //Calculate the I-controller value and add it to the pid_i_mem variable
182   if (pid_i_mem > 400) pid_i_mem = 400;      //Limit the I-controller to the maximum controller output
183   else if (pid_i_mem < -400) pid_i_mem = -400;
184   //Calculate the PID output value
185   pid_output = pid_p_gain * pid_error_temp + pid_i_mem + pid_d_gain * (pid_error_temp - pid_last_d_error);
186   if (pid_output > 400) pid_output = 400;  //Limit the PI-controller to the maximum controller output
187   else if (pid_output < -400) pid_output = -400;
188
189   pid_last_d_error = pid_error_temp;  //Store the error for the next loop
190
191   if (pid_output < 5 && pid_output > -5) pid_output = 0;  //Create a dead-band to stop the motors when the robot is balanced
192
193   if (angle_gyro > 30 || angle_gyro < -30 || start == 0) {  //If the robot tips over or the start variable is zero or the battery is empty
194     pid_output = 0;                                        //Set the PID controller output to 0 so the motors stop moving
195     pid_i_mem = 0;                                         //Reset the I-controller memory
196     start = 0;                                             //Set the start variable to 0
197     self_balance_pid_setpoint = 0;                         //Reset the self_balance_pid_setpoint variable
198   }
199
```

Figure 71: PID controller calculations

When the PID controller outputs are ready, the values are then sent to each motor control variable separately for later controls. Then a series of condition function is made to control the robot movement using the Bluetooth controls. By dissecting the received byte into bits, each bit will represent a variable that can be modified.

In our case we used the first 8 bits represents:

- Forward movement
- Backward movement
- Turning left
- Turning right
- Emergency stopping
- steady speed mode
- fast speed mode
- battery voltage indicator

Each bit modifies the PID controller output for each motor separately to be able to turn and traverse with a much higher degree of freedom than just moving forward and reverse.

A check function is made to lower the PID setpoint till it approaches zero incase no forward or backwards commands are given

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Control calculations
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
pid_output_left = pid_output;   //Copy the controller output to the pid_output_left variable for the left motor
pid_output_right = pid_output;  //Copy the controller output to the pid_output_right variable for the right motor

if (received_byte & B00000001) {      //If the first bit of the receive byte is set change the left and right variable to turn the robot to the left
  pid_output_left += turning_speed;   //Increase the left motor speed
  pid_output_right -= turning_speed;  //Decrease the right motor speed
}
if (received_byte & B00000010) {      //If the second bit of the receive byte is set change the left and right variable to turn the robot to the right
  pid_output_left -= turning_speed;   //Decrease the left motor speed
  pid_output_right += turning_speed;  //Increase the right motor speed
}

if (received_byte & B00000100) {                            //If the third bit of the receive byte is set change the left and right variable to turn the robot to the right
  if (pid_setpoint > -2.5) pid_setpoint -= 0.05;            //Slowly change the setpoint angle so the robot starts leaning forewards
  if (pid_output > max_target_speed * -1) pid_setpoint -= 0.005;  //Slowly change the setpoint angle so the robot starts leaning forewards
}
if (received_byte & B00001000) {                            //If the forth bit of the receive byte is set change the left and right variable to turn the robot to the right
  if (pid_setpoint < 2.5) pid_setpoint += 0.05;             //Slowly change the setpoint angle so the robot starts leaning backwards
  if (pid_output < max_target_speed) pid_setpoint += 0.005;  //Slowly change the setpoint angle so the robot starts leaning backwards
}

if (!(received_byte & B00001100)) {                 //Slowly reduce the setpoint to zero if no foreward or backward command is given
  if (pid_setpoint > 0.5) pid_setpoint -= 0.05;       //If the PID setpoint is larger then 0.5 reduce the setpoint with 0.05 every loop
  else if (pid_setpoint < -0.5) pid_setpoint += 0.05;  //If the PID setpoint is smaller then -0.5 increase the setpoint with 0.05 every loop
  else pid_setpoint = 0;                            //If the PID setpoint is smaller then 0.5 or larger then -0.5 set the setpoint to 0
}
if (received_byte & B00010000) {  //If the fifth bit of the receive byte is set deactivate the robot and reset everything
  pid_output = 0;                 //Set the PID controller output to 0 so the motors stop moving
  pid_i_mem = 0;                  //Reset the I-controller memory
  start = 0;                      //Set the start variable to 0
}
```

Figure 72: Control calculations

```
234
235   if (received_byte & B00100000) {  //If the sixth bit of the receive byte is set lower the target speed to a more efficien
236     float turning_speed = 15;       //this mode is best for battery efficiency
237     float max_target_speed = 50;
238   }
239
240   if (received_byte & B01000000) {  //If the seventh bit of the receive byte is set increase the target speed
241     float turning_speed = 25;       //this mode is best for more torque demanding tasks (ramps and fast movements)
242     float max_target_speed = 100;
243   }
244
245   //may be there is more conditions
246
247
248
249
```

Figure 73: Control calculations cont.

To compensate for the non-linear behavior of the stepper motors, the following calculations are needed to get a linear speed behavior. The resulted pulse signals are copied to be later used in the interruption routine.

Finally, the loop timer is recalibrated to make sure a full loop time of 4 milliseconds (the time needed by the MPU to sense and send its current position reading for the next loop).

```
256   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
257   //Motor pulse calculations
258   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
259   //To compensate for the non-linear behaviour of the stepper motors the folowing calculations are needed to get a linear speed behaviour.
260   if (pid_output_left > 0) pid_output_left = 405 - (1 / (pid_output_left + 9)) * 5500;
261   else if (pid_output_left < 0) pid_output_left = -405 - (1 / (pid_output_left - 9)) * 5500;
262
263   if (pid_output_right > 0) pid_output_right = 405 - (1 / (pid_output_right + 9)) * 5500;
264   else if (pid_output_right < 0) pid_output_right = -405 - (1 / (pid_output_right - 9)) * 5500;
265
266   //Calculate the needed pulse time for the left and right stepper motor controllers
267   if (pid_output_left > 0) left_motor = 400 - pid_output_left;
268   else if (pid_output_left < 0) left_motor = -400 - pid_output_left;
269   else left_motor = 0;
270
271   if (pid_output_right > 0) right_motor = 400 - pid_output_right;
272   else if (pid_output_right < 0) right_motor = -400 - pid_output_right;
273   else right_motor = 0;
274
275   //Copy the pulse time to the throttle variables so the interrupt subroutine can use them
276   throttle_left_motor = left_motor;
277   throttle_right_motor = right_motor;
278
279   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
280   //Loop time timer
281   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
282   //The angle calculations are tuned for a loop time of 4 milliseconds. To make sure every loop is exactly 4 milliseconds a wait loop
283   //is created by setting the loop_timer variable to +4000 microseconds every loop.
284   while (loop_timer > micros())
285     ;
286   loop_timer += 4000;
287 }
```

Figure 74:Motor pulse calculation

Chapter 8 Conclusion and future work

8.1 Conclusion

The self-balancing robot provides a good learning experience to those who hope to learn how to implement PID controllers, learn how to prototype (electric circuit design, PCBs, basic mechanical structures), have a moderate knowledge on stepper motors and drive systems, accomplish full control of a MPU-6050 gyroscope, and have a starting point in the world of embedded systems and robotics. This project achieves its goals through opening its implementors to the practical side of control systems by guiding us through a journey of structure designing, electrical circuits implementation, controller tuning through various means, Interface design and much more. The project fluidity helps in opening the door to much more complex implementations as it does not limit to standard design principles (the sky is the limit).

8.2 Future work

The self-balancing robot does not dictate a certain implementation technique in terms of circuit design, mechanical structure design, different electrical components and more. So, here we will be mentioning some of the possible improvements that can be done to this project and also mention other control algorithms that may provide a much better response than the PID controllers.

8.2.1 Better Alternative for The Used Components

8.2.1.1 STM32 Micro-controller:

STM32 is a family of 32-bit microcontrollers. STM32 microcontrollers are more related to industrial and real-life applications. While the Arduino NANO is fully capable of controlling self-balancing robots, using the STM32 is an upgrade. STM32 is more powerful than Arduino. STM32 blue pill has a clock up to 72MHz while the Arduino NANO has a clock up to 16MHz. This means that the STM32 blue pill can send data or signals to the stepper motor drivers much faster than the Arduino NANO [33]. Programming the STM32 is more challenging than programming the Arduino, however, it gives a hands-on experience with a microcontroller widely used in the industry.

8.2.1.2 TMC2208 Motor driver:

Although the TMC2208 is more expensive than the A4988 and the DRV8825, its main advantage is its micro-stepping resolution. The TMC2208 offers a micro-stepping resolution up to 1/256 step. This high resolution means less noise coming from the motor and more stability for the self-balancing robot. The TMC2208 has the same pin diagram as the A4988 and the DRV8825 so there is no change when it comes to writing the code. [34]

8.2.1.3 ESP8266 WI-FI Module:

The ESP8266 WI-FI module can be used instead of the HC-05 Bluetooth module. Although the WI-FI module is generally more expensive and consumes more power, it has two main advantages. The first advantage is longer range. Bluetooth module has a range of about 10m while the WI-FI module has a range of about 45m. The next advantage is the faster transmission speed. The Bluetooth module has a transmission speed of 2Mbps while the WI-FI module has a transmission speed of 54 Mbps. [35]

8.2.2 Mechanical Structure Improvements

Opting for 3D printing over laser cutting can be advantageous because plastic, which is commonly used in 3D printing, generally has a lighter weight than aplywood. However, to utilize 3D printing effectively, it is necessary to have proficiency with the necessary tools, such as the SolidWorks program.

3D printing offers several advantages over laser cutting. One key advantage is its ability to produce complex geometries that are difficult or impossible to manufacture using laser cutting. Additionally, 3D printing allows for the use of multiple materials in a single part, offering greater design flexibility and the ability to produce parts with varying material properties. Finally, 3D printing allows for rapid prototyping and design iteration, making it an ideal choice for producing parts with complex shapes and varying material properties [36].

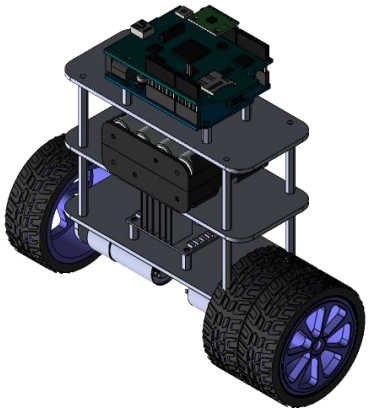

Figure 75: ED printed model of a two-wheeled self-balancing robot

8.2.3 Better Electrical Circuit and PCB Design Techniques

Throughout the project we noticed few details that can be either added or improved throughout our electric design that er decided to mention most of which in this section, this includes:

- Better schematic diagram.
- Better configurations for specific components.
- Recommended circuits to include in future designs.

8.2.3.1 Better Schematic Diagram

An issue we encountered during the assembly is the unavailability of some components in the current market which led to having some missing protection components like the diode. Another issue was the power wiring of some of the components wasn't ideal as the majority of required current by some components had to path through the Arduino first before reaching their target element.
We addressed all wiring related and missing components issues in the following design:

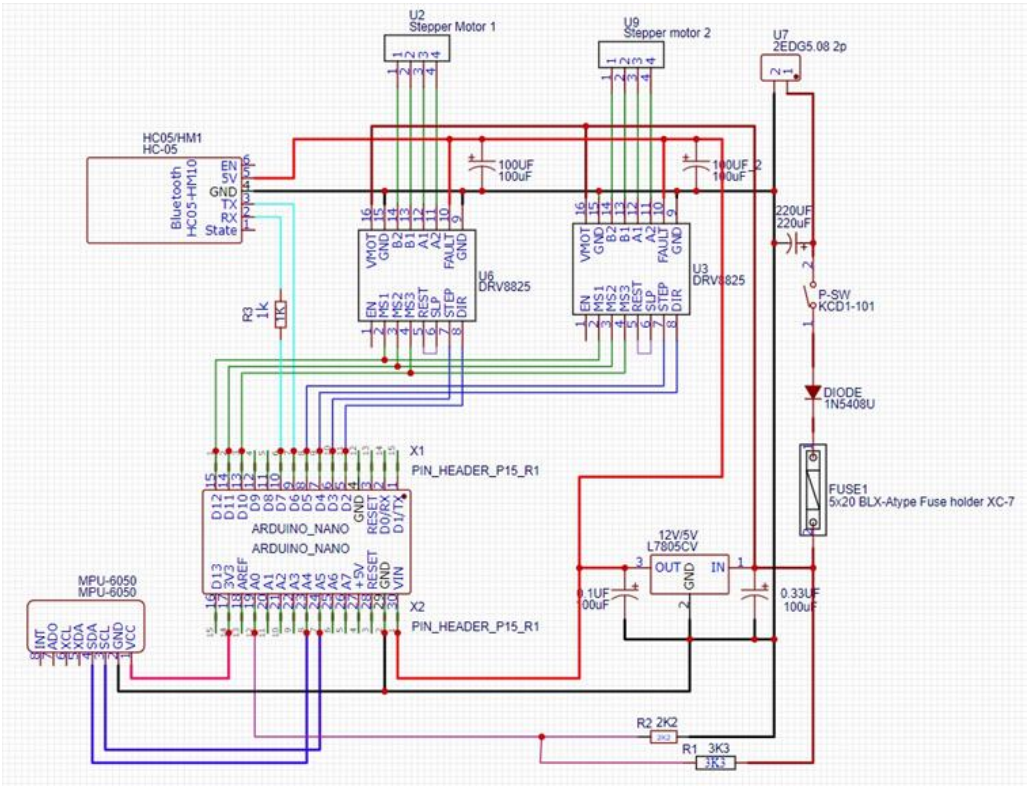

Figure 76: Better schematic diagram

8.2.3.2 Better Configurations for Specific Components

Throughout our practical movement testing we noticed small details that had no documentation on in component placements that result in an unpleasant experience when tunning the robot control output

One example is how to shape the PCB design. It is recommended to have a rounded corner

-aka rounded rectangular- PCB as it's the most ideal design to prevent concentration of stress on the edges of the PCB. Also, the screw holes on the PCB should be designed with the available screws on the market design, the lowest we could find (only in the electronics shops) was 3mm, the most ideal to be found in every possible shop was 5mm.

This board design is 120x60mm and 10mm radius of the rounded corners, with 5mm holes. The board can be wider and longer as you see fits your electric components design.

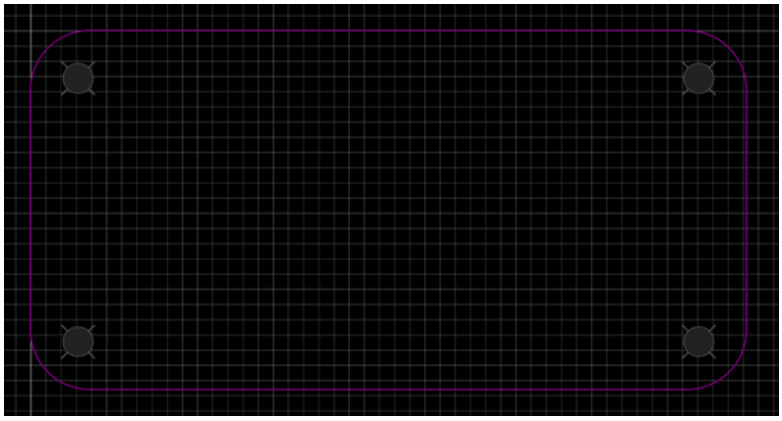

Figure 77: Recommended PCB layout

Another example is the positioning of the Arduino NANO board that had a capacitor in the way of the micro-USB port which prevented us from directly connecting the I2C bus to the Arduino board without needing to disconnect the board from the PCB in order to upload a new code. This can be solved by moving the USB side of the Arduino board to one of the PCB edges, this way we have a better space to reach with the connector.

Another issue we found is the lack of extra connection points with the Arduino board once it is connected to the PCB, So, a solution we found was to add an extra pin header for each pin on the Arduino this way we could easily connect to external test boards without needing to disassemble the PCB.

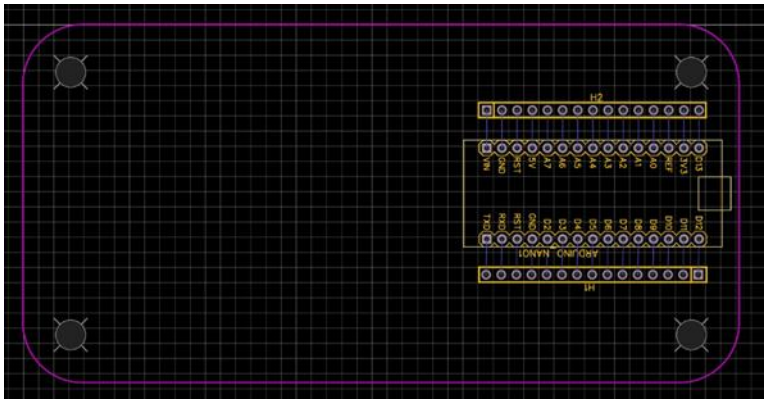

Figure 78: Recommended Arduino configuration on PCB

Another issue was the location of the MPU-6050 gyroscope on the PCB board was not centered to the robot center of motion. This caused the robot to start moving forward when only given an order to rotate around its own axis, a simple is to place the MPU directly in the exact center point of your PCB in case the PCB is centered inside the robot's body.

Also, it is recommended to have two 3mm screw holes in the PCB where the MPU will be to stabilize and minimize the reading error of the MPU.

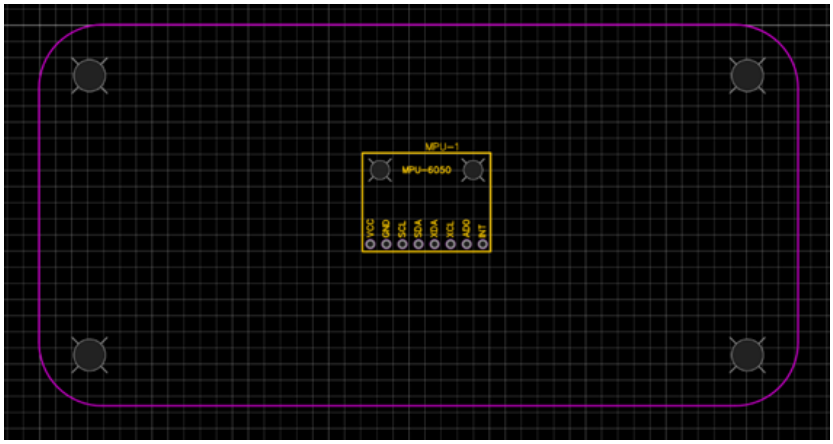

Figure 79: Recommended MPU-6050 position on the PCB

We also recommend using a pluggable terminal header for input power instead of regular pin headers as it proven to be more secure and safe to use for high tensile scenarios (regular pin headers are not durable enough for this project).

The pluggable terminal has proven to be the best terminal block as it works with all possible power sources while securing the connection both ways (holding the wires with nails and mechanically locking with PCB).

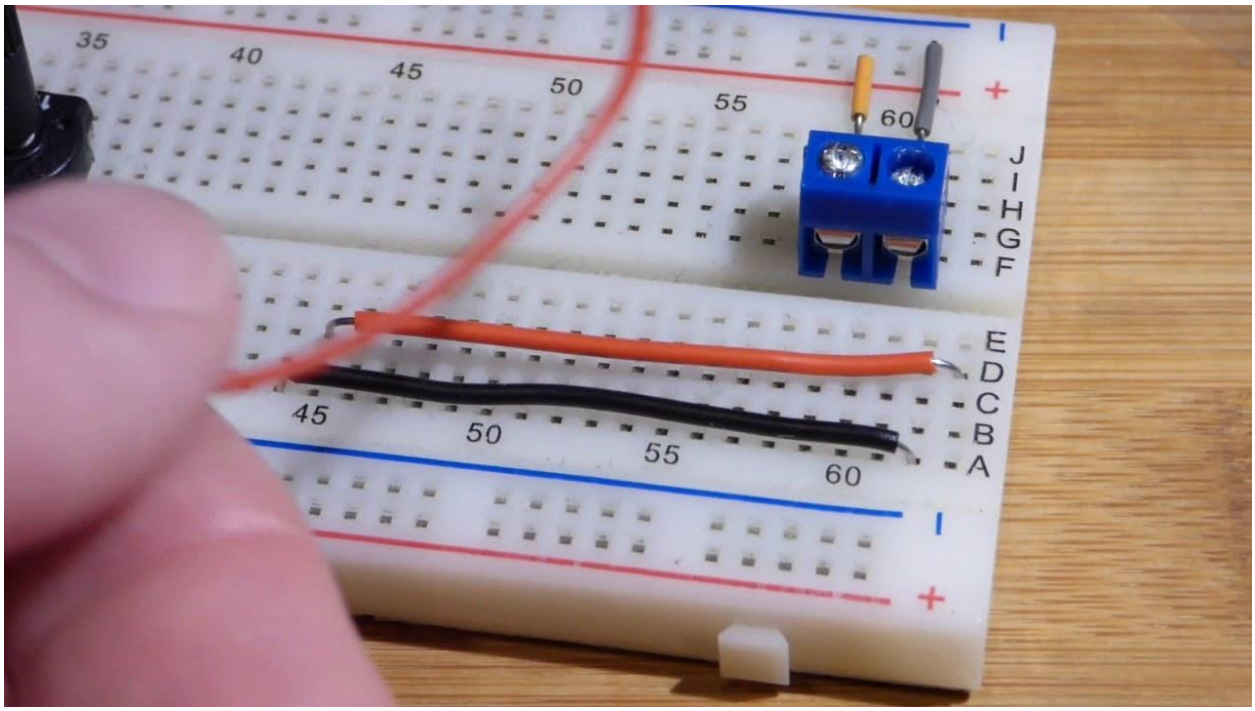

Figure 80: Pluggable terminal block 2 pins connected to an adaptor on a mini breadboard.

8.2.3.3 Recommended circuits to include in future designs

Built in voltage indicator using (The LM3914)

A common problem we faced was having to use external means to identify the current voltage state of the robot as there was no form of level indications on the batteries. A possible fix to this issue Is by using the LM3914 IC board in the following configuration. This IC provides an output to a certain voltage level depending on how the much input voltage is provided by the source in comparison to a set reference value of the IC.

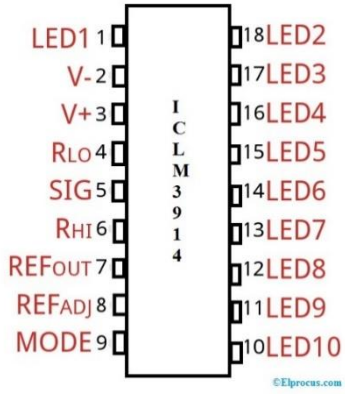

Figure 81:LM3914 recommended circuit from datasheet.

24V to 12V regulation

While using a 12V battery pack is cheaper and more economic, the robot could only last up to 1.5 hours of continuous operation. This can be resolved by adding more batteries, effectively increasing the maximum volage available while only utilizing more battery time. This can be done by using DC-DC step down converter circuit, they can be found for as cheap as 40 EGP and can be easily integrated in the PCB construction. A note worth mentioning is to avoid linear regulator as they result in so much thermal power losses compared to the switching type.

8.2.4 Different Control Algorithms

While the PID controller is an easy learning curve (provided that we already have a good background on the controller) and good system output and response times. The following methods provide a much better system overall output but require more time and effort to implement.

8.2.4.1 Fuzzy Logic Controller

A fuzzy logic controller is a controller that utilizes fuzzy logic to determine a course of action. A fuzzy logic controller is used to supervise conventional proportional and derivative actions such that the conventional gains are adapted online through fuzzy reasoning. The computation of the control action is composed of four steps: input scaling and shifting, fuzzification, fuzzy inference, and de-fuzzification.

For a self-balancing robot system, the robot will be successfully stabilized as required, with good dynamic performance. The robot's position, speed, angle, and angle rate return to the origin point.

8.2.4.2 Linear Quadratic Regulator (LQR) Controller

The Linear Quadratic Regulator (LQR) is a common technique that provides optimal controlled feedback gains to enable the closed-loop poles to be stable and high- performance design of systems. The main idea of operation of the LQR controller is based on finding the present control decision subject to certain constraints so as to minimize some measure of the deviation from the ideal behavior.

This type of controller needs mathematical modeling or the state space model of the system to obtain the feedback gains. For self-balancing robot system, LQR control is capable of rejecting disturbances quite good. It is also good for robot if the robot is moving around while maintaining its balance.

Appendix

- **The MPU Calibration Code**

```
#include <Wire.h>

byte error, MPU_6050_found, nunchuck_found, lowByte, highByte;
int address;
int nDevices;

void setup()
{
 Wire.begin();
 TWBR = 12;
 Serial.begin(9600);
}

void loop()
{
 Serial.println("Scanning I2C bus...");

 nDevices = 0;
 for(address = 1; address < 127; address++ )
 {
  Wire.beginTransmission(address);
  error = Wire.endTransmission();

  if (error == 0)
  {
   Serial.print("I2C device found at address 0x");
   if (address<16)Serial.print("0");
   Serial.println(address,HEX);
```

```
 nDevices++;
 if(address == 0x68 || address == 0x69){
  Serial.println("This could be a MPU-6050");
  Wire.beginTransmission(address);
  Wire.write(0x75);
  Wire.endTransmission();
  Serial.println("Send Who am I request...");
  Wire.requestFrom(address, 1);
  while(Wire.available() < 1);
  lowByte = Wire.read();
  if(lowByte == 0x68){
   Serial.print("Who Am I responce is ok: 0x");
   Serial.println(lowByte, HEX);
  }
  else{
   Serial.print("Wrong Who Am I responce: 0x");
   if (lowByte<16)Serial.print("0");
   Serial.println(lowByte, HEX);
  }
  if(lowByte == 0x68 && address == 0x68){
   MPU_6050_found = 1;
   Serial.println("Starting Gyro....");
   set_gyro_registers();
  }
 }
 if(address == 0x52){
  Serial.println("This could be a Nunchuck");
  Serial.println("Trying to initialise the device...");
  Wire.beginTransmission(0x52);
  Wire.write(0xF0);
  Wire.write(0x55);
  Wire.endTransmission();
  delay(20);
```

```
    Wire.beginTransmission(0x52);
    Wire.write(0xFB);
    Wire.write(0x00);
    Wire.endTransmission();
    delay(20);
    Serial.println("Sending joystick data request...");
    Wire.beginTransmission(0x52);
    Wire.write(0x00);
    Wire.endTransmission();
    Wire.requestFrom(0x52,1);
    while(Wire.available() < 1);
    lowByte = Wire.read();
    if(lowByte > 100 && lowByte < 160){
      Serial.print("Data response normal: ");
      Serial.println(lowByte);
      nunchuck_found = 1;
    }
    else{
      Serial.print("Data response is not normal: ");
      Serial.println(lowByte);
    }
   }
  }
  else if (error==4)
  {
   Serial.print("Unknown error at address 0x");
   if (address<16)
    Serial.print("0");
   Serial.println(address,HEX);
  }
 }
 if (nDevices == 0)
  Serial.println("No I2C devices found\n");
```

```
else
  Serial.println("done\n");
if(MPU_6050_found){
  Serial.print("Balance value: ");
  Wire.beginTransmission(0x68);
  Wire.write(0x3F);
  Wire.endTransmission();
  Wire.requestFrom(0x68,2);
  Serial.println((Wire.read()<<8|Wire.read())*-1);
  delay(20);
  Serial.println("Printing raw gyro values");
  for(address = 0; address < 20; address++ ){
    Wire.beginTransmission(0x68);
    Wire.write(0x43);
    Wire.endTransmission();
    Wire.requestFrom(0x68,6);
    while(Wire.available() < 6);
    Serial.print("Gyro X = ");
    Serial.print(Wire.read()<<8|Wire.read());
    Serial.print(" Gyro Y = ");
    Serial.print(Wire.read()<<8|Wire.read());
    Serial.print(" Gyro Z = ");
    Serial.println(Wire.read()<<8|Wire.read());
  }
  Serial.println("");
}
else Serial.println("No MPU-6050 device found at address 0x68");

if(nunchuck_found){
  Serial.println("Printing raw Nunchuck values");
  for(address = 0; address < 20; address++ ){
    Wire.beginTransmission(0x52);
    Wire.write(0x00);
```

```
    Wire.endTransmission();
    Wire.requestFrom(0x52,2);
    while(Wire.available() < 2);
    Serial.print("Joystick X = ");
    Serial.print(Wire.read());
    Serial.print(" Joystick y = ");
    Serial.println(Wire.read());
    delay(100);
   }
  }
  else Serial.println("No Nunchuck device found at address 0x52");
  while(1);
}

void set_gyro_registers(){
  //Setup the MPU-6050
  Wire.beginTransmission(0x68);                 //Start communication with the address found during search.
  Wire.write(0x6B);                             //We want to write to the PWR_MGMT_1 register (6B hex)
  Wire.write(0x00);                             //Set the register bits as 00000000 to activate the gyro
  Wire.endTransmission();                       //End the transmission with the gyro.

  Wire.beginTransmission(0x68);                 //Start communication with the address found during search.
  Wire.write(0x1B);                             //We want to write to the GYRO_CONFIG register (1B hex)
  Wire.write(0x00);                             //Set the register bits as 00000000 (250dps full scale)
  Wire.endTransmission();                       //End the transmission with the gyro

  Wire.beginTransmission(0x68);                 //Start communication with the address found during search.
  Wire.write(0x1C);                             //We want to write to the ACCEL_CONFIG register (1A hex)
  Wire.write(0x08);                             //Set the register bits as 00001000 (+/- 4g full scale range)
  Wire.endTransmission();                       //End the transmission with the gyro

  Wire.beginTransmission(0x68);                 //Start communication with the address found during search
  Wire.write(0x1A);                             //We want to write to the CONFIG register (1A hex)
```

```
  Wire.write(0x03);                                  //Set the register bits as 00000011 (Set Digital Low Pass Filter to
~43Hz)

  Wire.endTransmission();                            //End the transmission with the gyro

}
```

- **Main Control System Code**

```
#include <Stepper.h>

#include <SoftwareSerial.h>


#include <Wire.h>  //Include the Wire.h library so we can communicate with the gyro


int gyro_address = 0x68;           //MPU-6050 I2C address (0x68 or 0x69)

int acc_calibration_value = 1044;  //needs calibration                          //Enter the accelerometer
calibration value


//Various settings

float pid_p_gain = 25;         //25                                     //Gain setting for the P-controller
(25)

float pid_i_gain = 1.1;       //1.1                                    //Gain setting for the I-controller
(1.1)

float pid_d_gain = 30;         //30                                     //Gain setting for the D-controller
(30)

float turning_speed = 15;     //15                                   //Turning speed (15)

float max_target_speed = 50;  //50                                 //Max target speed (50)


//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

//Declaring global variables

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

byte start, received_byte, low_bat;


int left_motor, throttle_left_motor, throttle_counter_left_motor, throttle_left_motor_memory;

int right_motor, throttle_right_motor, throttle_counter_right_motor, throttle_right_motor_memory;

int battery_voltage;

int receive_counter;

int gyro_pitch_data_raw, gyro_yaw_data_raw, accelerometer_data_raw;
```

```
long gyro_yaw_calibration_value, gyro_pitch_calibration_value;


unsigned long loop_timer;


float angle_gyro, angle_acc, angle, self_balance_pid_setpoint;

float pid_error_temp, pid_i_mem, pid_setpoint, gyro_input, pid_output, pid_last_d_error;

float pid_output_left, pid_output_right;


SoftwareSerial myserial(2, 3);


////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

//Setup basic functions

////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

void setup() {

  Serial.begin(9600);  //Start the serial port at 9600 kbps

  Wire.begin();        //Start the I2C bus as master

  TWBR = 12;           //Set the I2C clock speed to 400kHz

  myserial.begin(9600);


  //To create a variable pulse for controlling the stepper motors a timer is created that will execute a piece
of code (subroutine) every 20us

  //This subroutine is called TIMER2_COMPA_vect

  TCCR2A = 0;               //Make sure that the TCCR2A register is set to zero

  TCCR2B = 0;               //Make sure that the TCCR2A register is set to zero

  TIMSK2 |= (1 << OCIE2A);  //Set the interupt enable bit OCIE2A in the TIMSK2 register

  TCCR2B |= (1 << CS21);    //Set the CS21 bit in the TCCRB register to set the prescaler to 8

  OCR2A = 39;               //The compare register is set to 39 => 20us / (1s / (16.000.000MHz / 8)) - 1

  TCCR2A |= (1 << WGM21);   //Set counter 2 to CTC (clear timer on compare) mode


  //By default the MPU-6050 sleeps. So we have to wake it up.

  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search.

  Wire.write(0x6B);                      //We want to write to the PWR_MGMT_1 register (6B hex)

  Wire.write(0x00);                      //Set the register bits as 00000000 to activate the gyro

  Wire.endTransmission();                //End the transmission with the gyro.

  //Set the full scale of the gyro to +/- 250 degrees per second

  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search.

  Wire.write(0x1B);                      //We want to write to the GYRO_CONFIG register (1B hex)

  Wire.write(0x00);                      //Set the register bits as 00000000 (250dps full scale)
```

```
  Wire.endTransmission();                //End the transmission with the gyro

  //Set the full scale of the accelerometer to +/- 4g.

  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search.

  Wire.write(0x1C);                      //We want to write to the ACCEL_CONFIG register (1A hex)

  Wire.write(0x08);                      //Set the register bits as 00001000 (+/- 4g full scale range)

  Wire.endTransmission();                //End the transmission with the gyro

  //Set some filtering to improve the raw data.

  Wire.beginTransmission(gyro_address);  //Start communication with the address found during search

  Wire.write(0x1A);                      //We want to write to the CONFIG register (1A hex)

  Wire.write(0x03);                      //Set the register bits as 00000011 (Set Digital Low Pass Filter to
~43Hz)

  Wire.endTransmission();                //End the transmission with the gyro


  pinMode(4, OUTPUT);  //Configure digital poort 2 as output

  pinMode(5, OUTPUT);

  pinMode(6, OUTPUT);  //Configure digital poort 3 as output

  pinMode(7, OUTPUT);

  pinMode(10, OUTPUT);

  pinMode(11, OUTPUT);  //Configure digital poort 4 as output

  pinMode(12, OUTPUT);  //Configure digital poort 5 as output

                        //Configure digital poort 6 as output

  digitalWrite(10, HIGH);

  digitalWrite(11, HIGH);

  digitalWrite(12, HIGH);


  for (receive_counter = 0; receive_counter < 500; receive_counter++) {  //Create 500 loops

                                                                         // if(receive_counter % 15 ==
0)digitalWrite(13, !digitalRead(13));        //Change the state of the LED every 15 loops to make the LED blink
fast

    Wire.beginTransmission(gyro_address);                                //Start communication with the gyro

    Wire.write(0x43);                                                    //Start reading the Who_am_I register
75h

    Wire.endTransmission();                                              //End the transmission

    Wire.requestFrom(gyro_address, 4);                                   //Request 2 bytes from the gyro

    gyro_yaw_calibration_value += Wire.read() << 8 | Wire.read();        //Combine the two bytes to make one
integer

    gyro_pitch_calibration_value += Wire.read() << 8 | Wire.read();      //Combine the two bytes to make one
integer

    delayMicroseconds(3700);                                             //Wait for 3700 microseconds to
simulate the main program loop time

  }

  gyro_pitch_calibration_value /= 500;  //Divide the total value by 500 to get the avarage gyro offset
```

```
  gyro_yaw_calibration_value /= 500;    //Divide the total value by 500 to get the avarage gyro offset


  loop_timer = micros() + 4000;  //Set the loop_timer variable at the next end loop time

}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

//Main program loop

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

void loop() {

  if (myserial.available()) {         //If there is serial data available

    received_byte = myserial.read();  //Load the received serial data in the received_byte variable

    receive_counter = 0;              //Reset the receive_counter variable

  }

  if (receive_counter <= 25) receive_counter++;  //The received byte will be valid for 25 program loops (100
milliseconds)

  else received_byte = 0x00;                     //After 100 milliseconds the received byte is deleted


  //Load the battery voltage to the battery_voltage variable.

  //85 is the voltage compensation for the diode.

  //Resistor voltage divider => (3.3k + 3.3k)/2.2k = 2.5

  //12.5V equals ~5V @ Analog 0.

  //12.5V equals 1023 analogRead(0).

  //1250 / 1023 = 1.222.

  //The variable battery_voltage holds 1050 if the battery voltage is 10.5V.

  //battery_voltage = (analogRead(0) * 1.222) + 85;


  /*if(battery_voltage < 1050 && battery_voltage > 800){                     //If batteryvoltage is below 10.5V
and higher than 8.0V

    digitalWrite(13, HIGH);                                               //Turn on the led if battery voltage
is to low

    low_bat = 1;                                                          //Set the low_bat variable to 1

  }*/



////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //Angle calculations


////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  Wire.beginTransmission(gyro_address);                                   //Start communication with the gyro
```

```
  Wire.write(0x3F);                                                    //Start reading at register 3F
  Wire.endTransmission();                                              //End the transmission
  Wire.requestFrom(gyro_address, 2);                                   //Request 2 bytes from the gyro
  accelerometer_data_raw = Wire.read() << 8 | Wire.read();             //Combine the two bytes to make one
integer
  accelerometer_data_raw += acc_calibration_value;                     //Add the accelerometer calibration value
  if (accelerometer_data_raw > 8200) accelerometer_data_raw = 8200;    //Prevent division by zero by limiting
the acc data to +/-8200;
  if (accelerometer_data_raw < -8200) accelerometer_data_raw = -8200;  //Prevent division by zero by limiting
the acc data to +/-8200;

  angle_acc = asin((float)accelerometer_data_raw / 8200.0) * 57.296;  //Calculate the current angle according to
the accelerometer

  if (start == 0 && angle_acc > -0.5 && angle_acc < 0.5) {  //If the accelerometer angle is almost 0
    angle_gyro = angle_acc;                                 //Load the accelerometer angle in the angle_gyro
variable
    start = 1;                                              //Set the start variable to start the PID controller
  }

  Wire.beginTransmission(gyro_address);                  //Start communication with the gyro
  Wire.write(0x43);                                      //Start reading at register 43
  Wire.endTransmission();                                //End the transmission
  Wire.requestFrom(gyro_address, 4);                     //Request 4 bytes from the gyro
  gyro_yaw_data_raw = Wire.read() << 8 | Wire.read();    //Combine the two bytes to make one integer
  gyro_pitch_data_raw = Wire.read() << 8 | Wire.read();  //Combine the two bytes to make one integer

  gyro_pitch_data_raw -= gyro_pitch_calibration_value;  //Add the gyro calibration value
  angle_gyro += gyro_pitch_data_raw * 0.000031;         //Calculate the traveled during this loop angle and add
this to the angle_gyro variable

////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //MPU-6050 offset compensation

////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //Not every gyro is mounted 100% level with the axis of the robot. This can be cause by misalignments during
manufacturing of the breakout board.
  //As a result the robot will not rotate at the exact same spot and start to make larger and larger circles.
  //To compensate for this behavior a VERY SMALL angle compensation is needed when the robot is rotating.
  //Try 0.0000003 or -0.0000003 first to see if there is any improvement.
```

```
  gyro_yaw_data_raw -= gyro_yaw_calibration_value;  //Add the gyro calibration value

  //Uncomment the following line to make the compensation active

  angle_gyro -= gyro_yaw_data_raw * 0.0000003;  //may be negative                              //Compensate the
gyro offset when the robot is rotating


  angle_gyro = angle_gyro * 0.9996 + angle_acc * 0.0004;  //Correct the drift of the gyro angle with the
accelerometer angle



////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //PID controller calculations


////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //The balancing robot is angle driven. First the difference between the desired angel (setpoint) and actual
angle (process value)

  //is calculated. The self_balance_pid_setpoint variable is automatically changed to make sure that the robot
stays balanced all the time.

  //The (pid_setpoint - pid_output * 0.015) part functions as a brake function.

  pid_error_temp = angle_gyro - self_balance_pid_setpoint - pid_setpoint;

  if (pid_output > 10 || pid_output < -10) pid_error_temp += pid_output * 0.015;


  pid_i_mem += pid_i_gain * pid_error_temp;  //Calculate the I-controller value and add it to the pid_i_mem
variable

  if (pid_i_mem > 400) pid_i_mem = 400;      //Limit the I-controller to the maximum controller output

  else if (pid_i_mem < -400) pid_i_mem = -400;

  //Calculate the PID output value

  pid_output = pid_p_gain * pid_error_temp + pid_i_mem + pid_d_gain * (pid_error_temp - pid_last_d_error);

  if (pid_output > 400) pid_output = 400;  //Limit the PI-controller to the maximum controller output

  else if (pid_output < -400) pid_output = -400;


  pid_last_d_error = pid_error_temp;  //Store the error for the next loop


  if (pid_output < 5 && pid_output > -5) pid_output = 0;  //Create a dead-band to stop the motors when the robot
is balanced


  if (angle_gyro > 30 || angle_gyro < -30 || start == 0) {  //If the robot tips over or the start variable is
zero or the battery is empty

    pid_output = 0;                                         //Set the PID controller output to 0 so the motors
stop moving

    pid_i_mem = 0;                                          //Reset the I-controller memory

    start = 0;                                              //Set the start variable to 0

    self_balance_pid_setpoint = 0;                          //Reset the self_balance_pid_setpoint variable
```

```
  }



////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //Control calculations


////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  pid_output_left = pid_output;   //Copy the controller output to the pid_output_left variable for the left
motor

  pid_output_right = pid_output;  //Copy the controller output to the pid_output_right variable for the right
motor



  if (received_byte & B00000001) {      //If the first bit of the receive byte is set change the left and right
variable to turn the robot to the left

    pid_output_left += turning_speed;   //Increase the left motor speed

    pid_output_right -= turning_speed;  //Decrease the right motor speed

  }

  if (received_byte & B00000010) {      //If the second bit of the receive byte is set change the left and right
variable to turn the robot to the right

    pid_output_left -= turning_speed;   //Decrease the left motor speed

    pid_output_right += turning_speed;  //Increase the right motor speed

  }



  if (received_byte & B00000100) {                                  //If the third bit of the receive byte is
set change the left and right variable to turn the robot to the right

    if (pid_setpoint > -2.5) pid_setpoint -= 0.05;                  //Slowly change the setpoint angle so the
robot starts leaning forewards

    if (pid_output > max_target_speed * -1) pid_setpoint -= 0.005;  //Slowly change the setpoint angle so the
robot starts leaning forewards

  }

  if (received_byte & B00001000) {                              //If the forth bit of the receive byte is set
change the left and right variable to turn the robot to the right

    if (pid_setpoint < 2.5) pid_setpoint += 0.05;               //Slowly change the setpoint angle so the robot
starts leaning backwards

    if (pid_output < max_target_speed) pid_setpoint += 0.005;  //Slowly change the setpoint angle so the robot
starts leaning backwards

  }



  if (!(received_byte & B00001100)) {                   //Slowly reduce the setpoint to zero if no foreward or
backward command is given

    if (pid_setpoint > 0.5) pid_setpoint -= 0.05;       //If the PID setpoint is larger then 0.5 reduce the
setpoint with 0.05 every loop

    else if (pid_setpoint < -0.5) pid_setpoint += 0.05;  //If the PID setpoint is smaller then -0.5 increase the
setpoint with 0.05 every loop
```

```
    else pid_setpoint = 0;                                //If the PID setpoint is smaller then 0.5 or larger
then -0.5 set the setpoint to 0

  }

  if (received_byte & B00010000) {  //If the fifth bit of the receive byte is set deactivate the robot and reset
everything

    pid_output = 0;                 //Set the PID controller output to 0 so the motors stop moving

    pid_i_mem = 0;                  //Reset the I-controller memory

    start = 0;                      //Set the start variable to 0

  }


  if (received_byte & B00100000) {  //If the sixth bit of the receive byte is set lower the target speed to a
more efficient values

    float turning_speed = 15;       //this mode is best for battery efficiency

    float max_target_speed = 50;

  }


  if (received_byte & B01000000) {  //If the seventh bit of the receive byte is set increase the target speed

    float turning_speed = 25;       //this mode is best for more torque demanding tasks (ramps and fast
movements)

    float max_target_speed = 100;

  }


  //may be there is more conditions







  //The self balancing point is adjusted when there is not forward or backwards movement from the transmitter.
This way the robot will always find it's balancing point

  if (pid_setpoint == 0) {                                    //If the setpoint is zero degrees

    if (pid_output < 0) self_balance_pid_setpoint += 0.0015;  //Increase the self_balance_pid_setpoint if the
robot is still moving forewards

    if (pid_output > 0) self_balance_pid_setpoint -= 0.0015;  //Decrease the self_balance_pid_setpoint if the
robot is still moving backwards

  }



////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //Motor pulse calculations


////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
```

```
  //To compensate for the non-linear behaviour of the stepper motors the folowing calculations are needed to get
a linear speed behaviour.
  if (pid_output_left > 0) pid_output_left = 405 - (1 / (pid_output_left + 9)) * 5500;
  else if (pid_output_left < 0) pid_output_left = -405 - (1 / (pid_output_left - 9)) * 5500;

  if (pid_output_right > 0) pid_output_right = 405 - (1 / (pid_output_right + 9)) * 5500;
  else if (pid_output_right < 0) pid_output_right = -405 - (1 / (pid_output_right - 9)) * 5500;

  //Calculate the needed pulse time for the left and right stepper motor controllers
  if (pid_output_left > 0) left_motor = 400 - pid_output_left;
  else if (pid_output_left < 0) left_motor = -400 - pid_output_left;
  else left_motor = 0;

  if (pid_output_right > 0) right_motor = 400 - pid_output_right;
  else if (pid_output_right < 0) right_motor = -400 - pid_output_right;
  else right_motor = 0;

  //Copy the pulse time to the throttle variables so the interrupt subroutine can use them
  throttle_left_motor = left_motor;
  throttle_right_motor = right_motor;


////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //Loop time timer

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

  //The angle calculations are tuned for a loop time of 4 milliseconds. To make sure every loop is exactly 4
milliseconds a wait loop
  //is created by setting the loop_timer variable to +4000 microseconds every loop.
  while (loop_timer > micros())
    ;
  loop_timer += 4000;
}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

//Interrupt routine  TIMER2_COMPA_vect

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

ISR(TIMER2_COMPA_vect) {
```

```
  //Left motor pulse calculations

  throttle_counter_left_motor++;                                    //Increase the throttle_counter_left_motor
variable by 1 every time this routine is executed

  if (throttle_counter_left_motor > throttle_left_motor_memory) {    //If the number of loops is larger then the
throttle_left_motor_memory variable

    throttle_counter_left_motor = 0;                                //Reset the throttle_counter_left_motor
variable

    throttle_left_motor_memory = throttle_left_motor;               //Load the next throttle_left_motor
variable

    if (throttle_left_motor_memory < 0) {                           //If the throttle_left_motor_memory is
negative

      PORTD &= 0b11101111;                                          //Set output 3 low to reverse the direction
of the stepper controller

      throttle_left_motor_memory *= -1;                             //Invert the throttle_left_motor_memory
variable

    } else PORTD |= 0b00010000;                                     //Set output 3 high for a forward direction
of the stepper motor

  } else if (throttle_counter_left_motor == 1) PORTD |= 0b00100000;  //Set output 2 high to create a pulse for
the stepper controller

  else if (throttle_counter_left_motor == 2) PORTD &= 0b11011111;    //Set output 2 low because the pulse only
has to last for 20us


 //right motor pulse calculations

  throttle_counter_right_motor ++;                                          //Increase the
throttle_counter_right_motor variable by 1 every time the routine is executed

  if(throttle_counter_right_motor > throttle_right_motor_memory){           //If the number of loops is larger
then the throttle_right_motor_memory variable

    throttle_counter_right_motor = 0;                                       //Reset the
throttle_counter_right_motor variable

    throttle_right_motor_memory = throttle_right_motor;                     //Load the next throttle_right_motor
variable

    if(throttle_right_motor_memory < 0){                                    //If the throttle_right_motor_memory
is negative

      PORTD |= 0b01000000;                                                  //Set output 5 low to reverse the
direction of the stepper controller

      throttle_right_motor_memory *= -1;                                    //Invert the
throttle_right_motor_memory variable

    }

    else PORTD &= 0b10111111;                                               //Set output 5 high for a forward
direction of the stepper motor

  }

  else if(throttle_counter_right_motor == 1)PORTD |= 0b10000000;            //Set output 4 high to create a
pulse for the stepper controller

  else if(throttle_counter_right_motor == 2)PORTD &= 0b01111111;            //Set output 4 low because the pulse
only has to last for 20us

}
```

References

[1] B. A. Schreiber., "robotics," [Online]. Available: https://www.britannica.com/technology/robotics. [Accessed 18 April 2023].

[2] "Balancing robot tutorial," 7 September 2014. [Online]. Available: https://wired.chillibasket.com/2014/09/balancing-robot-intro/. [Accessed 19 April 2023].

[3] "Inverted Pendulum: System Modeling," [Online]. Available: https://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling. [Accessed 20 April 2023].

[4] M. K. Saini, "Difference between Stepper Motor and DC Motor," 22 August 2022. [Online]. Available: https://www.tutorialspoint.com/difference-between-stepper-motor-and-dc-motor#:~:text=DC%20motors%20have%20continuous%20motion.&text=Stepper%20motors%20give%20slow%20response,response%20than%20a%20stepper%20motor.&text=The%20stepper%20motors%20are%20not,po. [Accessed 22 April 2023].

[5] C. Cavalo, "Stepper Motors vs. DC Motors - What's the Difference?," [Online]. Available: https://www.thomasnet.com/articles/machinery-tools-supplies/stepper-motors-vs-dc-motors/. [Accessed 22 April 2023].

[6] "What is a Stepper Motor : Types & Its Working," [Online]. Available: https://www.elprocus.com/stepper-motor-types-advantages-applications/. [Accessed 22 April 2023].

[7] C. Fiore, "Stepper Motors Basics: Types, Uses, and Working Principles," [Online]. Available: https://www.monolithicpower.com/en/stepper-motors-basics-types-uses#:~:text=The%20basic%20working%20principle%20of,rotor%20aligns%20with%20this%20field.. [Accessed 22 April 2023].

[8] B. SCHWEBER, "Unipolar vs. Bipolar drive for stepper motors, Part 1: principles," 6 January 2022. [Online]. Available: https://www.powerelectronictips.com/unipolar-vs-bipolar-drive-for-stepper-motors-part-1-principles-faq/. [Accessed 23 April 2023].

[9] J. Joseph, "How Does the MPU6050 Accelerometer & Gyroscope Sensor Work and Interfacing It With Arduino," 16 May 2022. [Online]. Available: https://circuitdigest.com/microcontroller-projects/interfacing-mpu6050-module-with-arduino#:~:text=How%20does%20MPU6050%20Module%20Work,of%20a%20system%20or%20object.. [Accessed 24 April 2023].

[10] "An Overview of Arduino Nano Board," [Online]. Available: https://www.elprocus.com/an-overview-of-arduino-nano-board/. [Accessed 24 April 2023].

[11] C. Woodford, "Lithium-ion batteries," 10 April 2022. [Online]. Available: https://www.explainthatstuff.com/how-lithium-ion-batteries-work.html. [Accessed 24 April 2023].

[12] "Control Stepper Motor with DRV8825 Driver Module & Arduino," [Online]. Available: https://lastminuteengineers.com/drv8825-stepper-motor-driver-arduino-tutorial/. [Accessed 25 April 2023].

[13] "HC-05 - Bluetooth Module," 16 July 2021. [Online]. Available: https://components101.com/wireless/hc-05-bluetooth-module. [Accessed 25 April 2023].

[14] P. Miller, "Building a two wheeled balancing robot," University of southern Queensland, Queensland, 2008.

[15] M. Ullman, "What's the difference? MDF vs. plywood," 11 February 2020. [Online]. Available: https://www.bobvila.com/articles/mdf-vs-plywood/. [Accessed 11 December 2022].

[16] J. G. M. K. Ye Ding1, "Modeling, Simulation and Fabrication of a," 12-18-2012.

[17] "Laser cutting: Examinig advantages and disadvantages of laser technology," 26 January 2022. [Online]. Available: https://www.rapiddirect.com/blog/advantages-and-disadvantages-of-laser-cutting/. [Accessed 11 December 2022].

[18] M. Rangaiah, "3D printing technology: Advantages and disadvantages," 27 May 2021. [Online]. Available: https://www.analyticssteps.com/blogs/3d-printing-technology-advantages-and-disadvantages. [Accessed 11 December 2022].

[19] Millennium Circuits Limited, "FR4: WHEN CAN YOU USE IT AND WHEN CAN YOU NOT," Millennium Circuits Limited | 7703 Derry St. Harrisburg, PA 17111-5205, [Online]. Available: https://www.mclpcb.com/blog/fr4-guide/. [Accessed 17 4 2023].

[20] R. S. Fearing, "Design of biologically inspired robots," in *2003 IEEE/RSJ International Conference on Intelligent Robots and Systems i(IROS 2003)*, pp. 4224-4229, doi: 10.1109/IROS.2003.1250932., 2003.

[21] S. B. a. K. M. M. Abderrahim, "Design and implementation of a fuzzy controller for a two-wheeled self-balancing robot,," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 5, pp. 101-106, 2012.

[22] J. J. Craig, Introduction to Robotics: Mechanics and Control , 3rd ed, Upper Saddle River, NJ: USA: Pearson Prentice Hall, 2005.

[23] M. A. a. K. M. S. Bououden, "Design of a self-balancing robot with a new control strategy,," *International Journal of Advanced Computer Science and Applications*, vol. 4, no. 12, pp. 188-192, 2013.

[24] R. S. K. a. J. K. Gupta, A Textbook of Machine Design, 14th ed, Delhi: India: S. Chand & Company Ltd, 2018.

[25] M. A. H. a. M. A. R. M. A. Hannan, "Design and Implementation of a Self-Balancing Robot Using PID Controller".

[26] J. A. A.-G. a. A. A. Al-Raweshidy, "Design and Implementation of a Self-Balancing Two-Wheeled Robot Using PID Control".

[27] Y. R. Anmol Singh Shekhawat, "Design and Control of Two-wheeled Self-Balancing Robot using Arduino".

[28] J. D. P. a. A. E.-N. Gene F. Franklin, "Feedback Control of Dynamic Systems".

[29] S. C. D. Y. L. G. a. K. N. Liuping Wang, "PID and Predictive Control of Electrical Drives and Power Converters Using MATLAB/Simulink," pp. 45-47.

[30] "Self balancing robot | Simulink basics series," [Online]. Available: https://www.youtube.com/watch?v=QtmVFlZi5T8&ab_channel=Algobotics.

[31] MathWorks, "MathWorks," [Online]. Available: https://www.mathworks.com/.

[32] MathWorks, "MathWorks," [Online]. Available: https://www.mathworks.com/help/ident/.

[33] Y. Sanghvi, "Arduino Uno vs STM32duino (Blue Pill)," 31 July 2021. [Online]. Available: https://www.tutorialspoint.com/arduino-uno-vs-stm32duino-blue-pill. [Accessed 2 July 2023].

[34] "TMC2208," [Online]. Available: https://wiki.fysetc.com/TMC2208/. [Accessed 2 July 2023].

[35] AKB, "Are Bluetooth and Wifi Modules the Same?," 19 November 2022. [Online]. Available: https://www.campuscomponent.com/blogs/post/are-bluetooth-and-wifi-modules-the-same#:~:text=Difference%20Between%20Bluetooth%20And%20WiFi,your%20gadgets%20to%20the%20internet.. [Accessed 3 July 2023].

[36] A. S. a. L. S. S. K. Khanna, "3D Printing: A state-of-the-art review," in *Materials Today:*

Proceedings, vol. 32, pp. 103-110, 2020, doi: 10.1016/j.matpr.2020.03.081., 2020.

[37] B. Schweber, "Stepper Motors Make the Right Moves with Precision, Ease and Smarter Drivers," [Online]. Available: https://www.mouser.in/publicrelations_techarticle_steppermotors_rightmoves_2015final/. [Accessed 23 April 2023].

[38] "MEMS Accelerometer," [Online]. Available: https://www.leveldevelopments.com/2020/10/what-are-inclinometers/mems-accelerometer/. [Accessed 24 April 2023].

[39] "GY-521 MPU6050 Triple 3-Axis Accelerometer Gyroscope I2C," [Online]. Available: https://electra.store/product/gy-521-mpu6050-triple-3-axis-accelerometer-gyroscope-i2c/. [Accessed 24 April 2023].

[40] "A000005 - Arduino Nano Board," [Online]. Available: https://www.distrelec.ch/en/arduino-nano-board-arduino-a000005/p/11096733. [Accessed 24 April 2023].

[41] "18650 Li-Ion Rechargeable Battery 1C (1200 MAh)," [Online]. Available: https://quartzcomponents.com/products/18650-li-ion-rechargeable-battery. [Accessed 24 April 2023].

[42] [Online]. Available: https://en.wikipedia.org/wiki/Laser_cutting.

[43] [Online]. Available: What is 3D Printing? - Technology Definition and Types. (n.d.). https://www.twi-global.com/technical-knowledge/faqs/what-is-3d-printing.

[44] "Karooza.net," [Online]. Available: https://karooza.net/dertermining-the-centre-of-gravity-for-a-self-balancing-robot. [Accessed 9 May 2023].

[45] H. Chin, "2.004 Lab 8 Intro: Self-Balancing Robot Control Part I: Stabilization Using PID Control," 2019.

[46] J. G. M. K. Ye Ding, "Modeling, Simulation and Fabrication of a Balancing Robot," 2012 .

[47] "Inverted pendulum," January 2010. [Online]. Available: https://en.wikipedia.org/wiki/Inverted_pendulum. [Accessed 20 April 2023].