# Self-Balancing Robot

## Introduction

The self-balancing robot is similar to an upside-down pendulum. Unlike a normal pendulum which keeps on swinging once given a nudge, this inverted pendulum cannot stay balanced on its own. It will simply fall over. Then how do we balance it? Consider balancing a broomstick on our index finger which is a classic example of balancing an inverted pendulum. We move our finger in the direction in which the stick is falling. Similar is the case with a self-balancing robot, only that the robot will fall either forward or backward. Just like how we balance a stick on our finger, we balance the robot by driving its wheels in the direction in which it is falling. What we are trying to do here is to keep the center of gravity of the robot exactly above the pivot point.

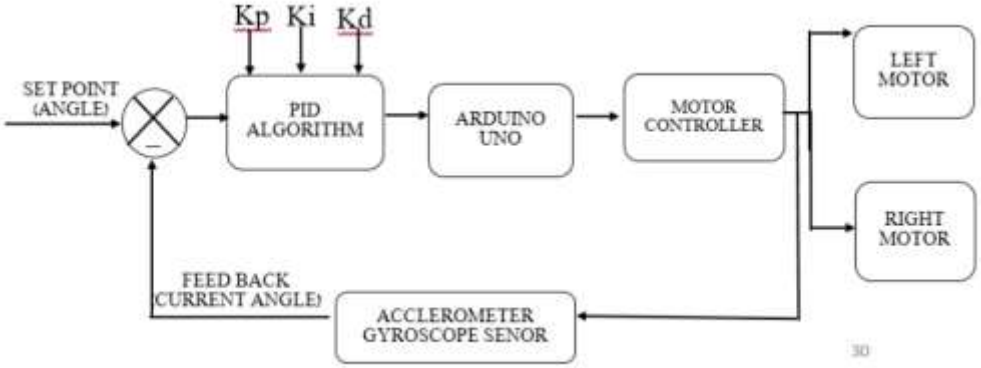
## Design and theory of operation

- ### Working Principle

The IMU sensor mounted on top of the vehicle measured the acceleration and angular acceleration in three axes namely x, y, and z. These values were processed by a Digital Motion Processor (DMP) which transformed these values into a more convenient set of variables i.e., yaw, pitch, and roll. Here only pitch was necessary as it produced the value for tilt in the axis under consideration. This value was fed to the controller; which acted as feedback to the microcontroller.

The microcontroller processed the values obtained from DMP according to program-specified algorithms.

The controller compared the pitch value obtained from feedback with the pre-set value, if there was a deviation then the error value was sent to the PID controller which via its algorithm produced a proportional force to be applied to the motors in order to bring it to back to the original vertical position.
The control signal was produced and sent to the motor controller L298N. The motor controller derived the motor at the specified speed, torque, and direction.




- ## Design Process

The self-balancing robot gets balanced on a pair of wheels having the required grip providing sufficient friction. For maintaining the vertical axis two things must be done, one is measuring the inclination angle and the other is controlling of motors to move forward or backward to maintain a 0 angle with the vertical axis. For measuring the angle, two sensors, an accelerometer, and a gyroscope were used. The accelerometer can sense either static or dynamic forces of acceleration and a gyroscope measure the
angular velocity.

The outputs of the sensors are fused using a Complementary filter. Sensors measure the process; The output gets subtracted from the reference set-point value to produce an error. Error is then fed into the PID where the error gets managed in three ways.

After the PID algorithm processes the error, The controller produces a control signal u. PID control signal then gets fed into the process under control. The process under PID control is a two-wheeled robot. PID control signal will try to drive the process to the desired set-point value that is O' in a vertical position by driving the motors in such a way that the robot is balanced.

- ## PID Control System

The control algorithm that was used to maintain It balance of the autonomous self-balancing two-wheel robot was the PID controller. The proportional. integral, and derivative Pid controller is well known as a three-term roller.

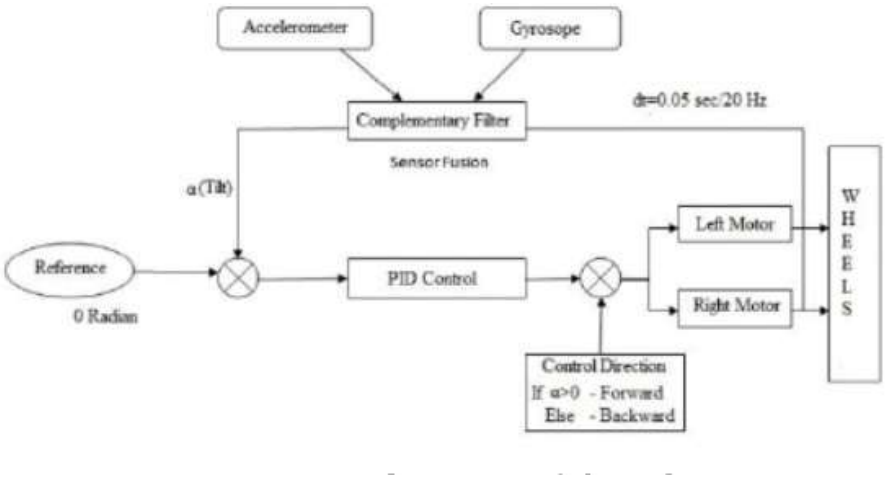
The input to the controller is the error from the system The Kp, Ki, and Kd are referred to as the proportional, integral, and derivative constants (the three terms get multiplied by these constants respectively the Closed loop control system 1s also referred to as a negative feedback system. The basic idea of a negative feedback system is that it measures the process output y from a sensor.

The measured process output gets subtracted from the reference set-point value to Produce an error. The error is then fed into the PID controller, where the error gets managed in three ways, The error will be used on the PID controller to execute the proportional term and integral term for the reduction of steady-state errors, and the derivative term handles overshooting.

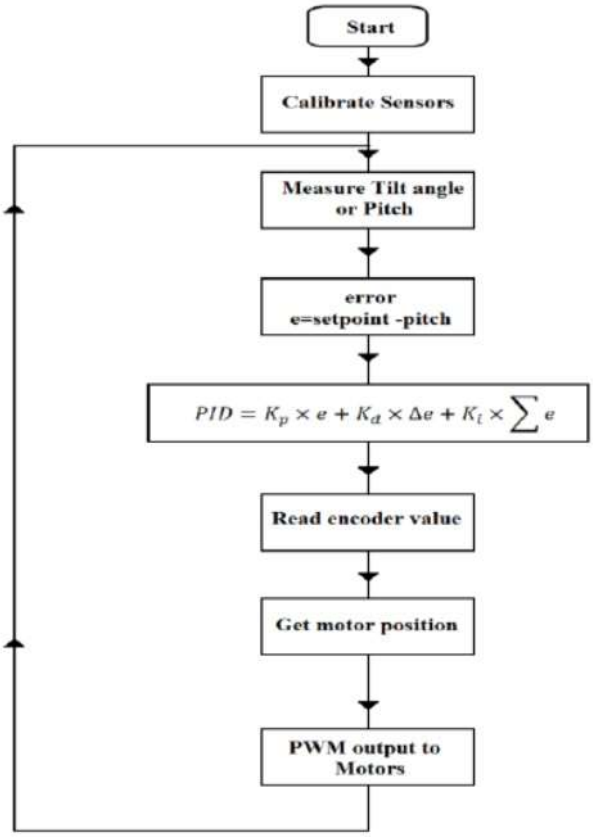
After the PID algorithm processes, the error the controller produces a control signal, The PID control signal then gets fed into the process under control.

- ## CONTROL SYSTEM AND ALGORITHM

The control system of the two wheels balancing robot in this project is illustrated by the block diagram



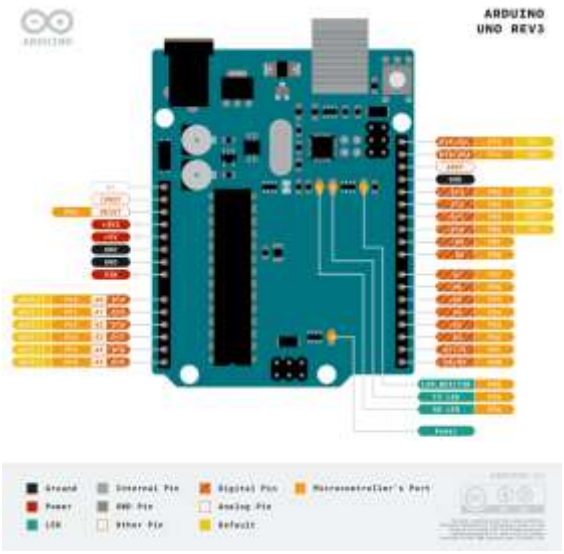
The robot control software is inside the microcontroller module and the program flowchart is shown




## Components

- ### Arduino UNO

Arduino UNO is a microcontroller board based on the ATmega328P (datasheet). It has 14 digital input/output pins (of which 6 can be as PWM outputs), 6 analog inputs, a MHz ceramic resonator (CSTCE16M0V53-R0)



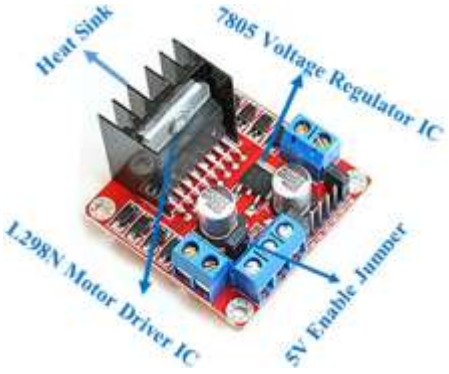

used 16

- ## L298N Motor Driver Module

- This module consists of an L298 motor driver IC and a 78M05 5V regulator. L298N Module can control up to 4 DC motors, or 2 DC motors with directional and speed control.

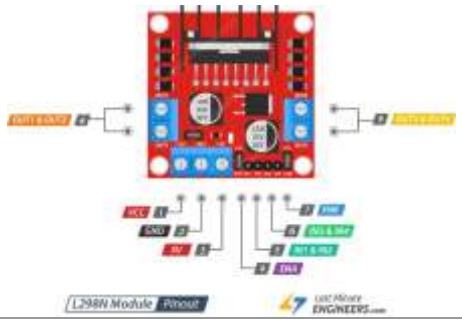
- The L298N Motor Driver module consists of an L298 Motor Driver IC, 78M05 Voltage Regulator, resistors, capacitor, Power LED, 5V jumper in an integrated circuit.

- Close look at the module:
  It has a heat sink to cool down the motor driver IC.
  It has a regulator to convert voltage from 12V to 5V so we can use it to power the Arduino.




- L298N Module can control up to 4 DC motors, or 2 DC motors with directional and speed control

- Features & Specifications:
  - Driver Chip: Double H Bridge L298N
  - Motor Supply Voltage (Maximum): 46V
  - Motor Supply Current (Maximum): 2A
  - Logic Voltage: 5V
  - Driver Voltage: 5-35V
  - Driver Current: 2A
  - Logical Current: 0-36mA
  - Maximum Power (W): 25W
  - Current Sense for each motor
  - Heatsink for better performance
  - Power-On LED indicator




| Pin Name | Description |
|---|---|
| **IN1 & IN2** | Motor A input pins. Used to control the spinning direction of Motor A |
| **IN3 & IN4** | Motor B input pins. Used to control the spinning direction of Motor B |
| **ENA** | Enables PWM signal for Motor A |
| **ENB** | Enables PWM signal for Motor B |
| **OUT1 & OUT2** | Output pins of Motor A |
| **OUT3 & OUT4** | Output pins of Motor B |
| **VCC-12V** | 12V input from DC power source |
| **5V** | Supplies power for the switching logic circuitry inside L298N IC |
| **GND** | Ground pin |

- ## DC motor

That DC Geared Motor with Wheel we used 2 of it on our robot to let the robot move.

Engine parameters:

- Supply voltage: 5 V
- Current consumption: approx. 180 mA
- Integrated gearbox: 48:1
- Bilateral shaft
- Rotation speed after gearbox: approx. 80 rpm.
- Torque after gearbox: approx. 0.5 kg. cm (0.049 Nm)

The parameters of the wheel:

- Tire diameter: 65mm
- Tire width: 26 mm

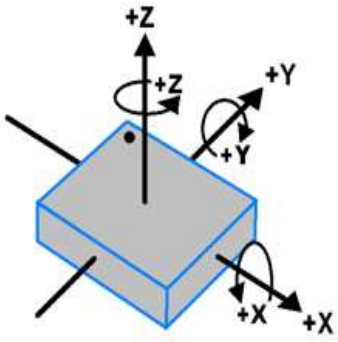
- ## MPU6050 (Gyroscope + Accelerometer) Sensor Module

The MPU6050 sensor module is a complete 6-axis Motion Tracking Device. It combines 3-axis Gyroscope, 3-axis Accelerometer, and Digital Motion Processor all in n small packages. Also, it has an additional feature of the on-chip Temperature sensor. It has an I2C bus interface to communicate with the microcontrollers.
It has an Auxiliary I2C bus to communicate with other sensor devices like 3-axis Magnetometer, Pressure sensor, etc.
If 3-axis Magnetometer is connected to the auxiliary I2C bus, then MPU6050 can provide complete 9-axis Motion Fusion output.

## 3-Axis Gyroscope

The MPU6050 consist of a 3-axis Gyroscope with Micro Electro Mechanical System (MEMS) technology. It is used to detect rotational velocity along the X, Y, and Z axes as shown in the below figure

When the gyros are rotated about any of the sense axes, the Coriolis Effect causes a vibration that is detected by a MEM inside MPU6050.
- The resulting signal is amplified, demodulated, and filtered to produce a voltage that is proportional to the angular rate.
- This voltage is digitized using 16-bit ADC to sample each axis.
- The full-scale range of output are +/- 250, +/- 500, +/- 1000, +/- 2000.
- It measures the angular velocity along each axis in degrees per second unit.

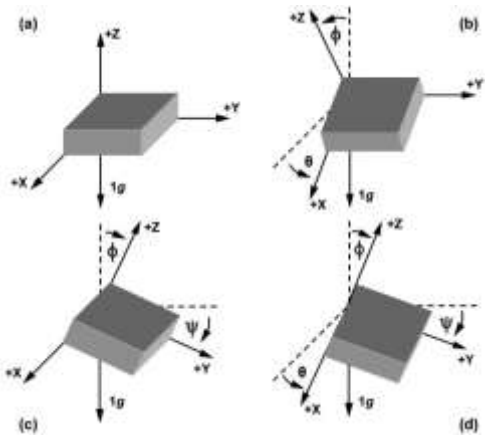



MPU-6050
Orientation & Polarity of Rotation

Gyroscope values in °/s (degree per second)
Angular velocity along the X axis = (Gyroscope X axis raw data/131) °/s.
Angular velocity along the Y axis = (Gyroscope Y axis raw data/131) °/s. Angular velocity along the Z axis = (Gyroscope Z axis raw data/131) °/s.

## 3-Axis Accelerometer

The MPU6050 consists 3-axis Accelerometer with Micro Electro Mechanical (MEMs) technology. It is used to detect the angle of tilt or inclination along the X, Y, and Z axes as shown below figure.

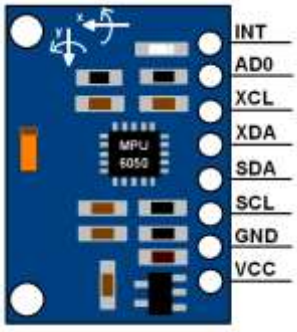



Accelerometer values in g (g force)

Acceleration along the X axis = (Accelerometer X axis raw data/16384) g.
Acceleration along the Y axis = (Accelerometer Y axis raw data/16384) g.
Acceleration along the Z axis = (Accelerometer Z axis raw data/16384) g.

Acceleration along the axes deflects the movable mass.

- This displacement of the moving plate (mass) unbalances the differential capacitor which results in sensor output. Output amplitude is proportional to acceleration.
- 16-bit ADC is used to get digitized output.
- The full-scale range of acceleration are +/- 2g, +/- 4g, +/- 8g, +/- 16g.
- It is measured in g (gravity force) units.
- When the device is placed on a flat surface it will measure 0g on the X and Y axis and +1g on the Z axis.




The MPU-6050 module has 8 pins,

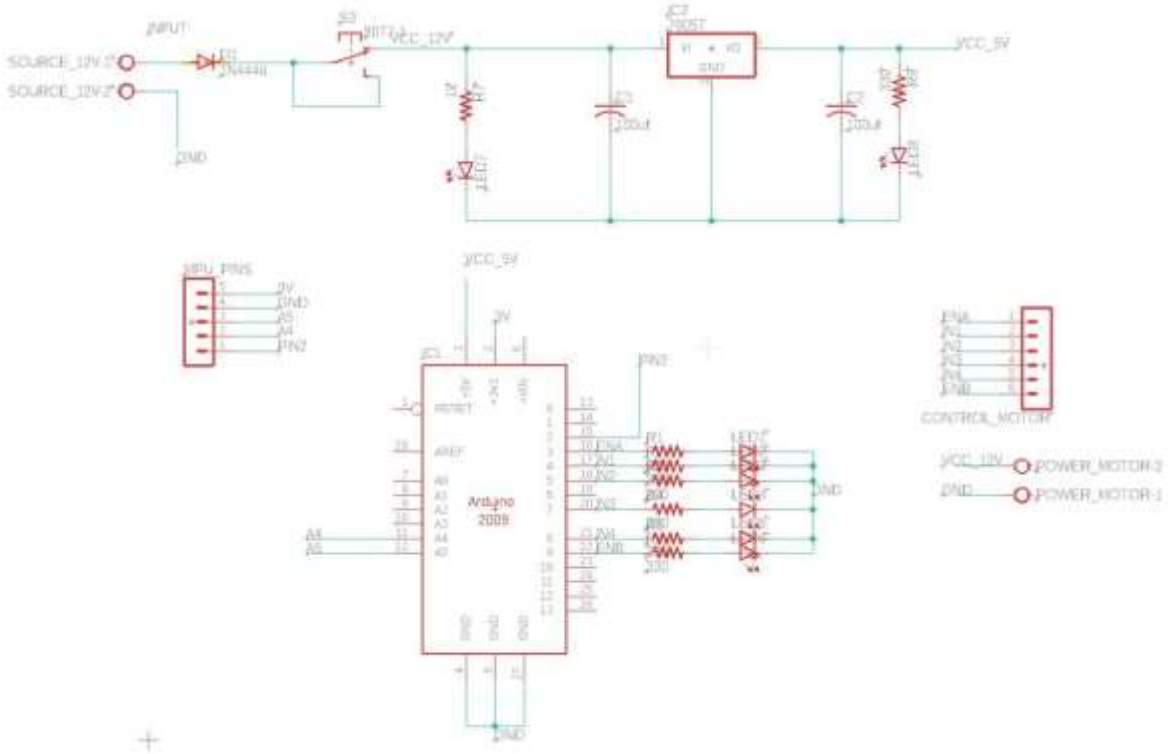
| Pin Name | Description |
|---|---|
| **INT** | Interrupt digital output pin |
| **AD0** | I2C Slave Address LSB pin. This is the 0th bit in the 7-bit slave address of the device. If connected to VCC then it is read as logic one and the slave address changes. |
| **XCL** | Auxiliary Serial Clock pin. This pin is used to connect other I2C interface-enabled sensor SCL pin to MPU-6050. |
| **XDA** | Auxiliary Serial Data pin. This pin is used to connect other I2C interface-enabled sensor SDA pin to MPU-6050. |
| **SCL** | Serial Clock pin. Connect this pin to the microcontroller's SCL pin. |
| **SDA** | Serial Data pin. Connect this pin to the microcontroller's SDA pin. |
| **VCC** | Power supply pin. Connect this pin to +5V DC supply. |
| **GND** | Ground pin |

## Connections

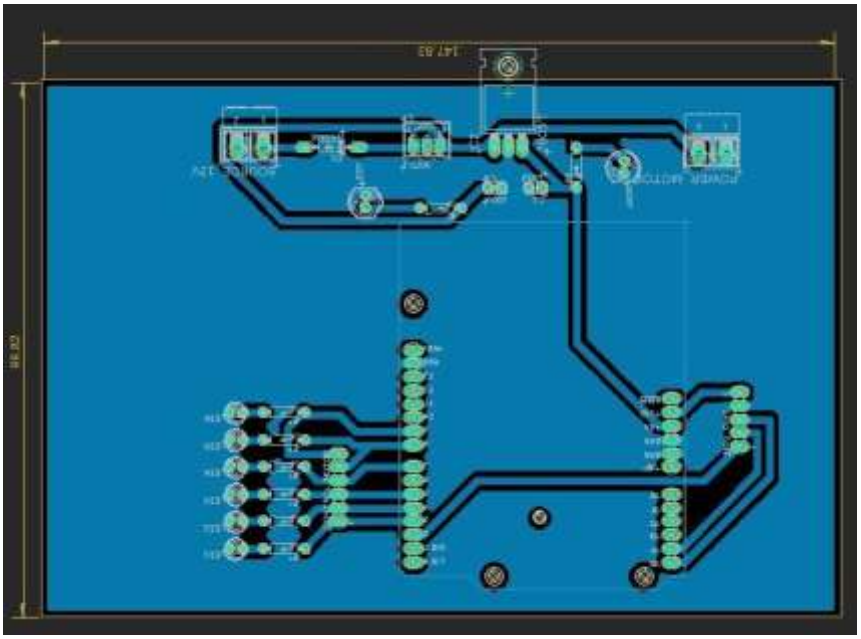
| **MPU-6050** | |
|---|---|
| **MPU-6050 Pin** | **Arduino Pin** |
| GND | GND |
| VCC | 5V |
| SCL | A5 |
| SDA | A4 |
| INT | D2 |

| L298N Motor Driver | |
|---|---|
| **L298N Motor Driver Pin** | **Arduino Pin** |
| 12V | Positive Battery Lead |
| GND | GND + Negative Battery Lead |
| 5V | Vin |
| ENA | D3 |
| IN1 | D4 |
| IN2 | D5 |
| IN3 | D7 |
| IN4 | D8 |
| ENB | D9 |

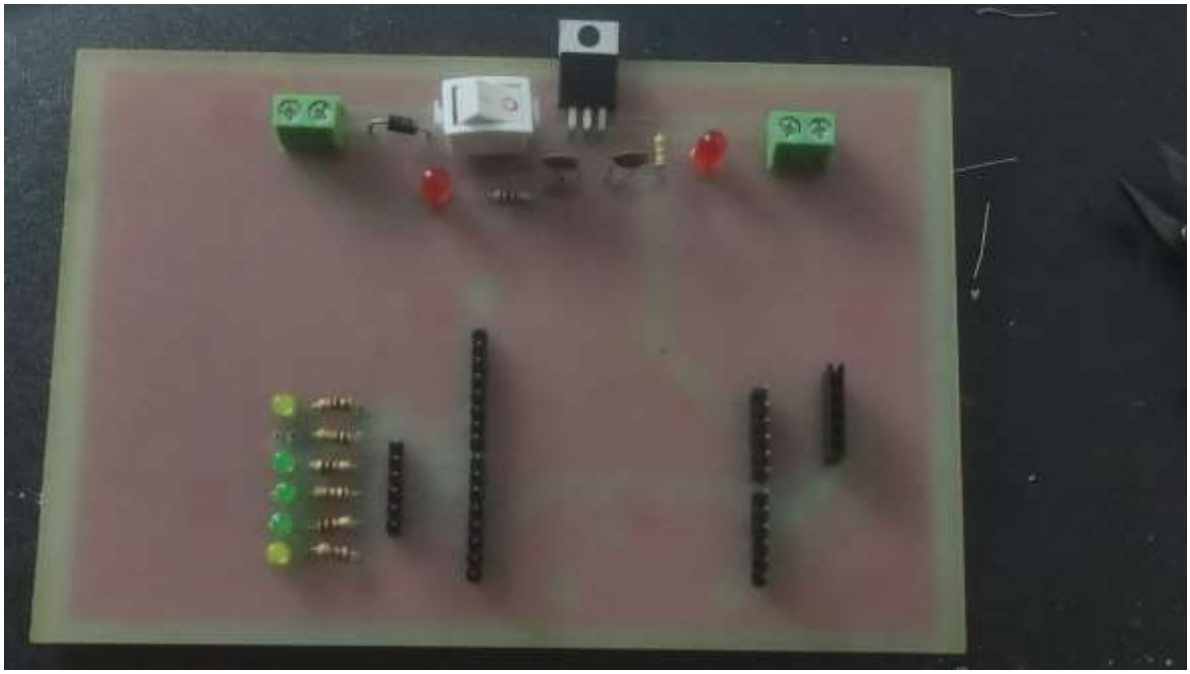
## The schematic circuit:



## PCB design:



## Final PCB:



# The Code

Press here to visit GitHub repository

## Overview

- Before diving into the code explanation, it's good to mention that the code is divided into several layers and not a one chunk piece of code as a (.ino) file. Each layer is divided into its own cpp (.cpp) and header (.h) files. The header file contains all constants' definitions in the form of (#define) statements as well as the functions' prototypes, while the cpp file contains functions' declarations.

- The layers in our robot are:
  - Application: App.cpp – App.h
  - Motors: motors.cpp – motors.h
  - MPU-6050: mpu6050.cpp – mpu6050.h
  - PID: pid.cpp – pid.h
  - Test: test.cpp – test.h

- All besides the (.ino) file which calls the App_start() and App_init() functions present in the application layer.

- First, the application layer (upper most layer) and it has the following functions:
  - App_init() calls the initialization functions for the motors, PID controller, and MPU-6050.
  - App_start() where the application logic for the whole system runs.
  - App_test() where we call each layer's test function at a time to test it when needed.

- Second, the motors layer and it has the following functions:
  - motors_setup() to set pins direction(input or output)
  - move(int speed) and it takes speed as an input parameter holding both the magnitude and sign to move the robot either forward or backward with desired speed.
  - moveForward() to move the wheels in one direction with full speed.
  - moveBackward() to move the wheels in the other direction with full speed.
  - stop() to stop both wheels.

- Third, the MPU-6050 layer and it has the following functions:
  - mpu_setup() to calibrate the sensor and initialize its internal processor(DMP).
  - dmpDataReady() an ISR that's triggered when new data is available to be read.

- mpu_update() to constantly update the sensor's readings.
- return_pitch() to return inclination angle around the y-axis.

- Fourth, the PID layer and it has the following functions:
  - PID_setup() to apply initial configurations as sampling time and mode either automatic or manual for the PID controller.
  - PID_run() to run the PID algorithm to do its calculations.
  - get_pid_output() to return the output calculated by PID which in our case is motor speed ranging from (-255, 255).

- Last but not least, the Test layer to test each of the above layers separately and check if they function correctly. It contains the following functions:
  - test_motors() where we call the move(int speed) function of motors and send different speeds using a Serial connection. Speeds are either positive or negative to move in the corresponding direction.
  - test_mpu6050() and it basically returns the pitch angle to be printed to the serial monitor to check that the readings are correct.
  - test_pid(int input) that takes the inclination angle as input and returns the output speed that should be sent to motors. This is to check that the algorithm is running correctly and its output is reasonable relative to the input angle fed to it.

- That was all for the different layers in the code, what functions are there in each, and a brief summary of each function and its usage.

## • Explanation

- In order to make the robot self-balance using the PID algorithm we have to specify the input and output of the system as well as tune our parameters to get optimum results, and before even this, you should understand how the PID works under the hood. For the PID, we'll be using Brett Beauregard's library.

### MPU-6050

- For the input it's the inclination angle of the robot which is fed from the MPU-6050's readings, depending on how you mount the MPU-6050 relative to your robot structure/plates, you specify the desired axis you're rotating around, in our case it's the pitch angle or angle around Y-axis.

- For readings to be the most accurate -by time this project is out-, we used the MPU-6050's internal processor (DMP) manipulated using Jeff Rowberg's library, mpu6050.h as well as MPU6050_6Axis_MotionApps20.h and the I2Cdev.h for the

I2C communication protocol used by the device to send and receive data with the Arduino. You'll find the link for all these libraries down below.

## PID Controller

- For the output from the system, it's the motors' speed ranging from (-255,255) and the negative sign indicates the direction that's manipulated inside the code. PID is where the magic happens to decide which speed and direction is it to give as output, let's dive a bit into it, check the Design and theory of operation section for more.
- You've to set the initial set point in order for the PID to know where to maintain the system, as we're feeding it with an inclination angle it's around zero degrees (or 180 if you're adding +180 to all angles fed to the system, a matter of adding an offset and doesn't affect how it works), to know what your set point is, make your robot stand still and get this reading from the MPU6050. The PID keeps comparing the input fed to it with this setpoint in order to compensate for the error, then start tuning your parameters.
- The Kp parameter acts as a driving force to make the robot's speed proportional to the direction it falls towards, taking into consideration that a very large number leads to overshooting.
- The Ki parameter is responsible for compensating the offset error that's making the robot away from its setpoint, it's the integral term that looks into the past to overcome this error but also a large value leads to overshooting.
- The Kd value acts as the resistive force, it looks into the future to predict the upcoming angle then damps the overshoots and makes the system quickly reach a steady state. Yet a large value would make it oscillate badly.
- Knowing all this you've to find your perfect match of all three parameters to get satisfactory results, it's basically based on trial and error until you get it.
- In our case this wasn't easy, it took us over a month of work to know what best works for our design.
- The approach we took was as follows:
    - Start tuning the Kp while all others are left zero, let it be a really low value and then double this value each time you test until you find your robot oscillating around its set point, then you can take this value and cut it into half and let it be your Kp. If for example, your best output was at Kp = 8, then let it be 4.
    - Next start tuning your Kd with Ki still zero and Kp with the value you've reached, the same approach, start with a really low value like 0.1 and start doubling it every time until your robot starts balancing with an offset from the setpoint, reaching this, again cut the Kd value into half and leave it untouched.

- Next is tuning the Ki with the same approach, doubling each time and then cutting it in half.
- By this you should have reached your parameters or at least you're around what values they should be for your system.

### Wrapping up

- So, this is the whole system, you get readings from the MPU6050 you pass them to the PID controller, it does its calculations and gives you the output speed which is fed to the motors.
- This is exactly what's found in the infinite loop of the program.

## • Important Milestones throughout the testing process

- Initially the microcontroller used was the ESP-32, why especially? Because it has built-in WI-FI and Bluetooth modules on its board as well as its small compact size relative to the Arduino UNO we had, however, it showed severe issues. It constantly had problems uploading Arduino sketches to it, you had to always hold the reset button pressed when the code was uploading until even this method stopped working. Another solution to the problem was to connect a 1 microfarad capacitor between its reset and ground pins and this also failed, so we had to switch to Arduino.

- It was also decided to have two modes, the automatic self-balancing mode that just balances the robot in place, and another manual one where you can move the robot in whatever direction you want using an ultrasonic sensor, you just adjust the distance and wave your hand in front of it for the robot to follow in the same direction. Due to problems with tuning and the fact that it barely reaches perfect balance without oscillations, the manual mode was omitted and to be added to its next version/update.

- It was also decided to have a lifting arm on top of it but for the same reason mentioned in the previous point, it was omitted and to be added to version 3 after the ultrasonic update.

- Better motors would lead to better results as DC motors used here aren't good enough, also doing stress analysis for the whole system will give better insights on where to place each component, especially the batteries.

- Batteries were placed on a 3rd top plate on the robot but due to severe overshoots and instant falls it was removed to the bottom layer and that led to instant better results for our design.

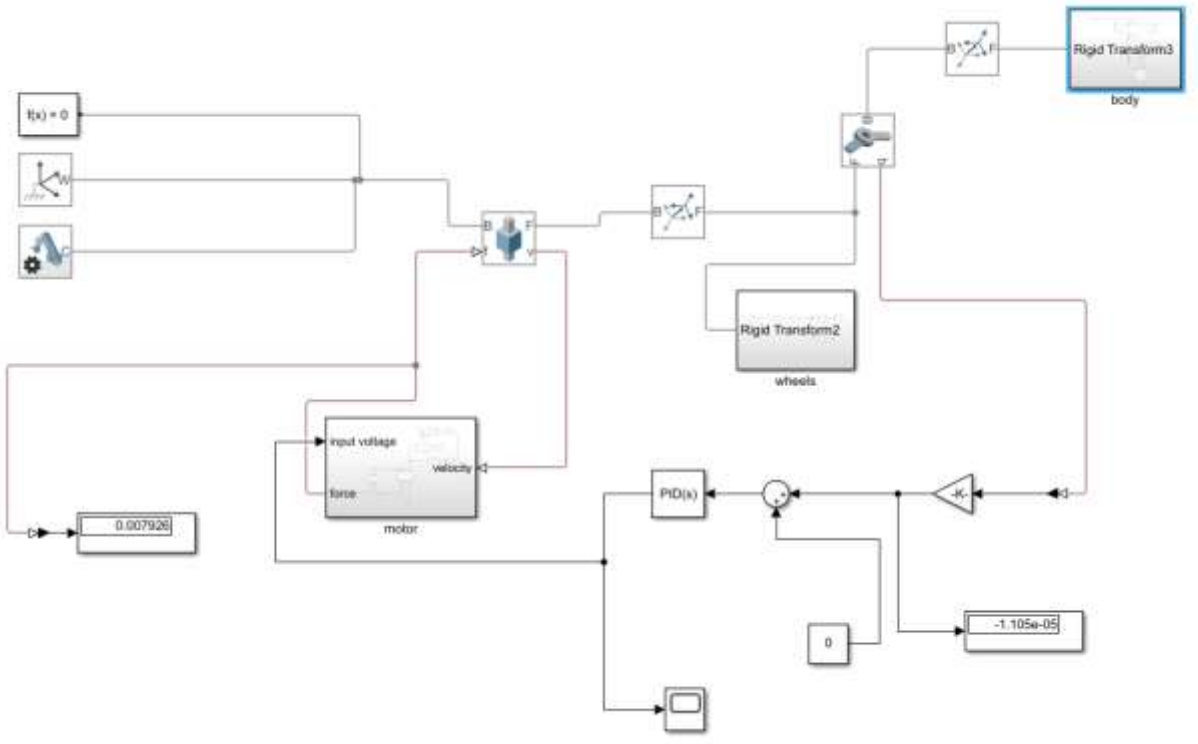
- The robot only balanced for 8 to 20 seconds and after several tunings it reached over 40 seconds yet on a carpet meaning friction was there, and it barely balanced on smooth surfaces.

- After many trials and surfing the internet for solutions we found the problem, the MPU-6050's gyroscope and accelerometer offsets weren't accurate and the parameters' tunings weren't near the desired ones, yet again it's a trial-and-error method so you never really know when you'll reach. And so, by trying some combinations from other codes on the internet we found what perfectly works for our case!

- Now the robot balances for forever with parameters set to Kp = 60, Ki = 140, Kd = 1.5

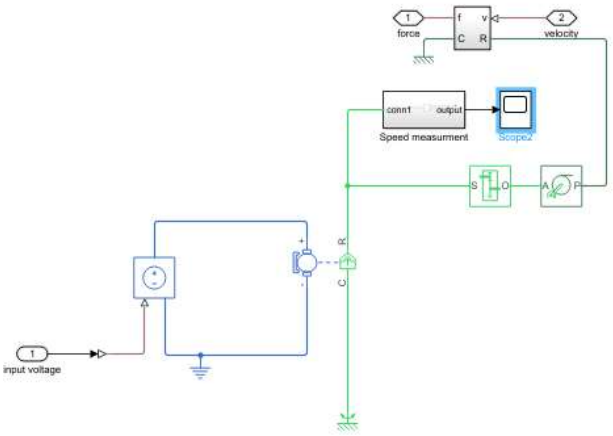
## Simulation

The circuit on MATLAB Simulink

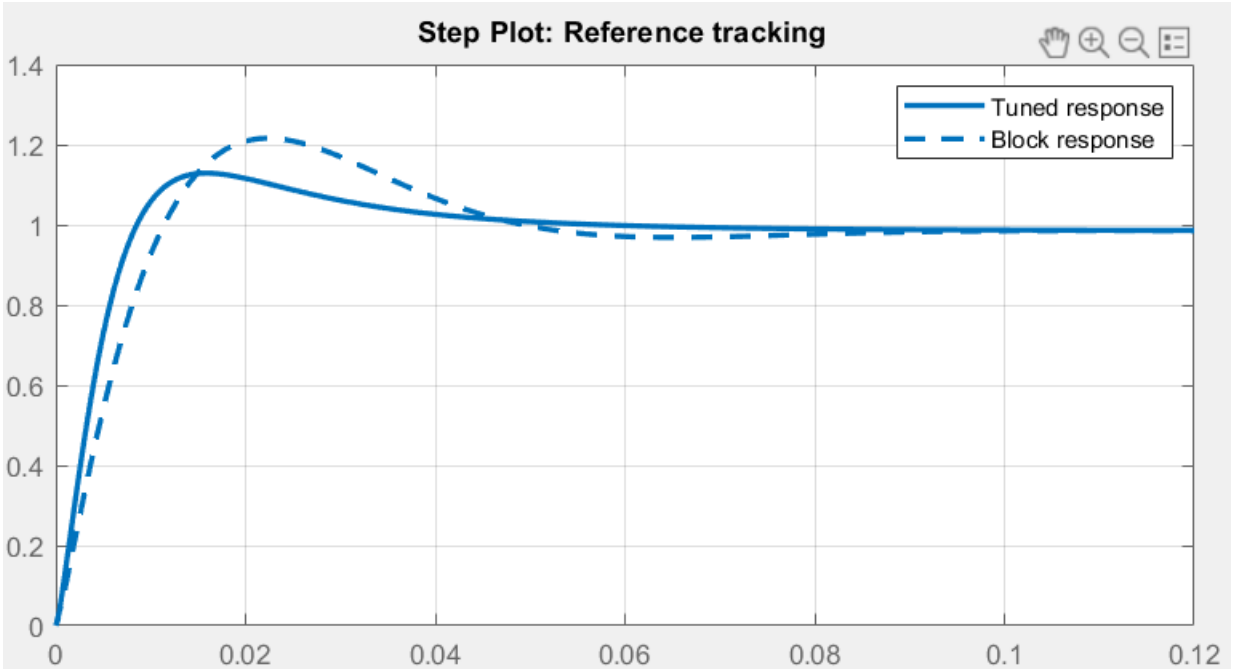



Using our design from Solid works and the PID controller connecting this circuit, the inclination angle is measured from the sensor in the revolt joint (which connects the body of the robot with wheels) and fed to the PID controller, the output of the PID controls the voltage of the dc motor.

motor:
dc motor is connected to a voltage controller and the output is rotational motion which is converted to translational motion with wheel axels and gear, the translational motion is converted to force and feed to the prismatic joint




PID Response:




(The simulation video will be attached to the PowerPoint presentation)

# Resources

1) PIDv2 library: https://github.com/imax9000/Arduino-PID-Library
2) Brett's blog posts | PID library explanation line by line: http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/
3) Useful resource for getting a sense of PID control: https://www.youtube.com/channel/UC923b-omXUs0dnWOTDD7FhA/videos
4) MPU-6050 datasheet "just getting a sense of it": https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

   MPU-6050 library: https://github.com/ElectronicCats/mpu6050