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

Code:

Press here to visit GitHub repository

I- Overview

- Before diving into the code explanation, it's good to mention that it's 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 this 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 motion.
- Third is the MPU-6050 layers and it has the following functions:
    - mpu_setup() to calibrate it and initialize its internal processor(DMP).
    - dmpDataReady() which is an ISR that's triggered when new data is available to be fetched out.
    - mpu_update() to constantly update the sensor's readings.
    - return_roll(), return_pitch() and return_yaw() to return inclination angles around the x-axis, y-axis and z-axis respectively.
- Fourth is the PID layer and it has the following functions:
    - PID_setup() to apply initial configurations as sampling time and mode either automatic or manual.

    - PID_run() to run the PID algorithm inside it and let it 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 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 readings are correct.
    - test_pid(int input) that takes 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 for each function and its usage.

II- Explanation

- In order to make the robot self-balance using 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 as 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 Arduino. You'll find the link for all these libraries down below.

- The code starts with initial calibration for the device that takes several seconds and then it starts reading the inclination angle.

PID Controller

- For the output from the system, it's the motors' speed ranging from (-255,255) and the negative sign indicates 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.
- The PID needs to know you initial set point in order to know where to maintain your system, as we're feeding it with an inclination angle it's around zero degrees -make your robot stand still and get this reading from the MPU6050-, it keeps comparing the input fed to it with this setpoint in order to compensate for the error.
- The Kp parameter acts as a driving force to make the robot's speed proportional to the direction its falling 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 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 try 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 it's 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, same approach, start with a really low value like 0.001 and start doubling it every time until your robot starts balancing or so 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 then cutting into 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.

III- Important Milestones throughout the testing process

- Initially the microcontroller used was the ESP-32, why especially? Because it has a 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 code was uploading until even this method stopped working. Another solution to the problem was to connect a 1 micro farad 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 to over 40 seconds yet on a carpet meaning friction was there, and it barely balanced on smooth surfaces.
- ADD YOUR FINAL UPDATE.

IV- Resources

1) PIDv2 library: https://github.com/imax9000/Arduino-PID-Library
2) Brett's blog posts: 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) MPU6050 datasheet "just getting a sense of it": https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf