Bilal Abdunur Munana

A Self-Stabilizing Platform

Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics

Bachelor's Thesis

15 April 2022

Abstract

Author:	Bilal Abdunur Munana
Title:	A Self-Stabilizing Platform
Number of Pages:	34 pages + 2 appendices
Date:	15 April 2022
Degree:	Bachelor of Engineering
Degree Programme:	Degree Programme in Electronics
Professional Major:	Electronics
Supervisors:	Anssi Ikonen, Senior Lecturer

Nowadays, electronics and mechanical devices are increasingly adapting the concept of self-stabilization. Similarly, as in the case of self-stabilization, self-stabilizing platforms have been utilized in numerous engineering disciplines. It is therefore necessary to examine how the Micro-electromechanical systems gyroscopes and accelerometer sensors are implemented in constructing functional self-stabilizing platforms.

Various concepts were applied in the implementation of the project to achieve a phased set up of a DIY self-stabilizing platform. Some of these concepts include C++ programming, 3D printing, data analysis and deduction. The process also involved testing the hardware components. A theoretical understanding of the operation of the sensors and their limitations as well as analyzing ways to overcome them was also necessary.

As result, a fully assembled DIY self-stabilizing platform that could successfully retain a balanced position with its balancing plate when an object was placed on it, was created. The platform can be further improved by use of bigger servo motors, as well as the use of smoother plastic enclosures other than the PLC 3D printed material.

Keywords: Self-stabilization, MPU6050 module, self-stabilizing platform

Contents

List of Abbreviations

CPU- Central Processing Unit

DIY- Do It Yourself

DMP- Digital Motion Processor

HPF- Low Pass Filter

IMU- Inertial Measurement Unit

I2C- Integrated Circuit

I/O- Input/Output

LPF -High Pass Filter

MEMS- Micro-electromechanical Systems

PWM- Pulse Width Modulation

SPI- Serial Peripheral Interface

1. Introduction

A steady increment in the use of self-stabilizing systems and platforms has been nothing short of evident. In various applications such as in mechanical and electronic devices, in which retaining a constant position regardless of movement or space fitting, self-stabilizing systems and platforms have been developed and designed to operate from day-to-day life situations such as in small action camera gimbals, to more complex systems such as in aircraft navigation.

This goal of the project was to examine a phased set up of a DIY self-stabilizing platform with a Micro-electromechanical System (MEMs) gyroscope and accelerometer.

The implementation of this project was achieved through the execution of five steps. The initial step was the acquisition and testing of the MPU6050 Inertial Measurement Unit (IMU) set up alongside the Arduino uno. The second step was to program the MPU6050 IMU to display its precise position coordinates whilst changing its position constantly. The servo motors were introduced in the third step to control the movement as dictated by the MPU6050 IMU. The fourth step involved enclosure 3D printing and in the final step, the platform was assembled.

In this report, the basic knowledge of self-stabilizing platforms is explained and the steps on how to accomplish and understand this concept better are demonstrated through the practical work procedures.

2. Theory

2.1 Self-Stabilization

Self-stabilization refers to a system's ability to recover automatically from unexpected faults [1]. There are two states in a self-stabilizing system, namely the initial state and the final state. A system qualifies to be self-stabilizing provided that from its initial state, it eventually reaches a correct final state through a finite number of execution steps. This concept is known as convergence. The system is guaranteed to stay in its correct state given that no fault occurs, which is known as closure [2]. Convergence and closure are the two conditions of a self-stabilizing system.

Self-stabilization is a concept that is incorporated in many modern computers and telecommunications networks and therefore a concept of fault tolerance in distributed systems. Distributed systems refer to computing environments, in which various components are spread across multiple computers on a network to execute a given task in a more efficient way as compared to a single computer or device. [3.] Fault tolerance is the ability of a system to continue functioning properly in the event of the failure of one or more faults within some of its components [4].

Self-stabilization has commonly been compared to the traditional fault tolerant algorithms and considered less promising amongst the distributed systems computing community. Traditional fault tolerance, however, is not always possible; for instance, it cannot be achieved when the system is started in an incorrect state. This is due to its complexity making it difficult to analyse and debug and consequently making it hard to prevent a distributed system from reaching an incorrect final state. [2.] A Self-stabilizing system reaches a correct final state regardless of whether it starts in an incorrect state.

2.2 Self-Stabilizing Platform

Self-stabilizing platforms are one of many applications of micro-electronic components alongside sensors with minimal power consumption. Their construction and assembly are compliant with suitable factors such as low cost and energy.

Self-stabilizing platforms can also be referred to as gyroscopic platforms or inertia platforms. They use gyroscopes to maintain a fixed orientation in any space fitting despite the movement of the system. [5.] In whatever direction the self-stabilizing platforms are moved or tilted, they tend to maintain a close to zero degrees angle in the direction of gravity as shown in figure 1. They tend to stay horizontal and there are two angles that ought to be measured, the roll and the pitch.

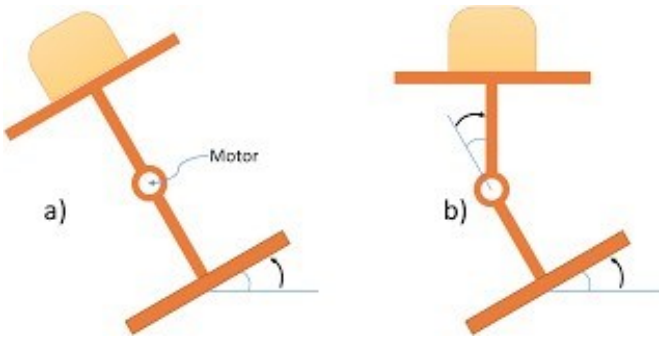

Figure 1, from a to b, platform adjusting and maintaining horizontal position when tilted. [6]

The roll and pitch angles of the platform can be measured through setting up a gyroscope and accelerometer. Combining a gyroscope and accelerometer allows the system to cope with the mounting (base) platform being tilted as shown in figure 1. In self-stabilizing platform set-ups, the construction of the platform is based on a MEMS sensor. The sensor chosen is the MEMS gyroscope and accelerometer in a single package- MPU6050 IMU.

2.3 IMU

An inertial measurement unit shown in figure 2 is a device that measures a body's movement. Gyroscopes and accelerometers measure rotational rate and linear acceleration, respectively. In common configuration, each vehicle axis contains an accelerometer, gyroscope within the IMU. Some IMUs also contain a magnetometer. The magnetometer shown in figure 3 serves as a heading reference. [7.]

IMUs measure the speed, acceleration, turn rate, and inclination of a body and they are commonly used for navigation for instance in aircrafts, where tracking attitude changes is crucial.

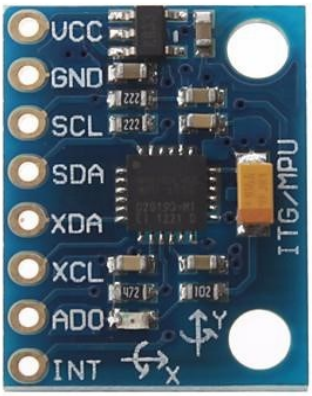

Figure 2. IMU 6 degrees of Freedom MPU6050 sensor [8].

Figure 3. Magnetometer [8].

An Inertial Measurement Unit is as a result of the combination of the gyroscopes and accelerometers which complement each other due to their individual weakness as known that an accelerometer measures G-forces, whilst a gyroscope detects its angular velocity. A gyroscope is likely to accumulate error and an accelerometer to

have gravity component shortcoming but when combined, these weaknesses are compensated swiftly. [6.]

However, an integration drift occurs over longer time periods in the system during the implementation of the IMU sensor when the gyroscope and accelerometer data is combined, since they fundamentally serve the same purpose of obtaining the angular position of the object. Therefore, error is induced in the sensor data due to the integration drift. The angle data contains so much noise as well. To overcome this issue, a complimentary filter can be used to minimize the noise. [9.]

2.4 Complementary Filter

A complementary filter is derived from the Kalman filter and is for a specific filtering class.
A complementary filter is implemented to filter out the noise and acquire more accurate angle readings from the IMU. The gyroscope data is then used in the calculation of the estimated angular position. As time passes however, the gyroscope begins to introduce errors in the output data which is brought about by the gyroscope drifting overtime. Therefore, the complementary filter can be used to obtain a more accurate estimate of the roll and pitch angles. As the accelerometer does not provide any data about yaw, it cannot be used to calculate yaw. Consequently, yaw is only measurable using a 6DoF IMU when using a gyroscope.

A high pass filter is implemented to negate the drift of the gyroscope, whilst a low pass filter deals with the temporary accelerometer variations. Given a high frequency signal passing through low pass filter G(s), an accelerometer output ($\boldsymbol{\alpha}$acc), a gyroscope output value ($\boldsymbol{\alpha}$gyro) and a high pass filter 1-G(s) with a low signal frequency passing through it. The angles $\boldsymbol{\alpha}$ for roll and pitch are calculated from equation 1. Noise in high frequency signals is mostly low frequency. In this case, it should be filtered using a high pass filter. The opposite is true for high frequency signals. [9.]

$$\boldsymbol{\alpha} = \boldsymbol{\alpha}\text{acc}(1\text{-}G(s)) + \int\boldsymbol{\alpha}\text{gyro}(G(s)) \qquad (1)$$

Equation 1 is illustrated in the diagram 4 below

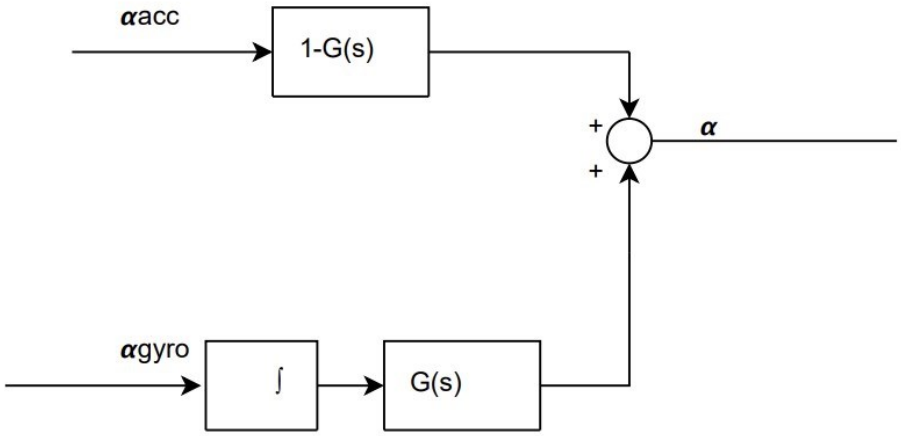

Figure 4. Simple illustration of the complementary filter designed in app diagram.

2.5 Gyroscopes.

Gyroscopes measure and maintain orientation and angular velocity.

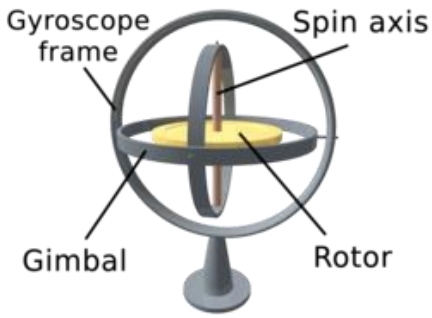

Figure 5. A gyroscope in its most basic form for understanding its operation principle [7].

The spin axis is free to rotate in any orientation in which it is unaffected by the tilting of the mounting in accordance with the conservation of angular momentum.

With this operation principle, gyroscopes as in figure 5 are further based on microelectromechanical systems technology and can be miniaturized into various electronic devices with minimal power consumption. The MEMS operate through monitoring the motion of a vibrating proof mass in figure 6 that is attached to a mounting frame through a MEMS spring-like structure in all spatial directions.

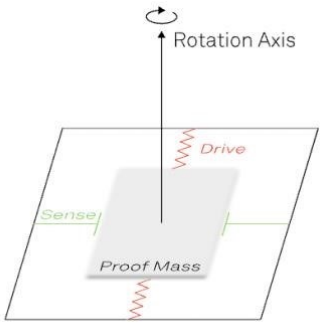

Figure 6. vibrating proof mass attached to the mounting frame [8].

MEMS gyroscopes are often combined with accelerometers providing six dimensional measurements from a single device as in figure 7.

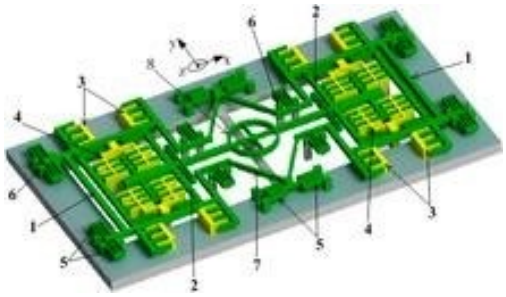

Figure 7. vibrating structure MEMS gyroscope [8].

Similarly, to other integrated circuits, these may provide either analog or digital outputs as in many cases, a single part includes multiple gyroscopic sensors.

The MPU6050 sensor module is going to be adopted in the stabilization of the platform. This module is a complete 6-axis motion tracking device that combines 3-axis gyroscope, 3-axis accelerometer, and digital motion processor all in a convenient package on a single board.
The module accommodates an on-chip temperature sensor as well as I2C bus interface to communicate with the microcontrollers not to mention auxiliary I2C bus to communicate with other sensor devices like 3-axis Magnetometer, pressure sensor and if the 3-axis Magnetometer is connected to auxiliary I2C bus, then the MPU6050 module can provide complete 9-axis Motion Fusion output. [8.]

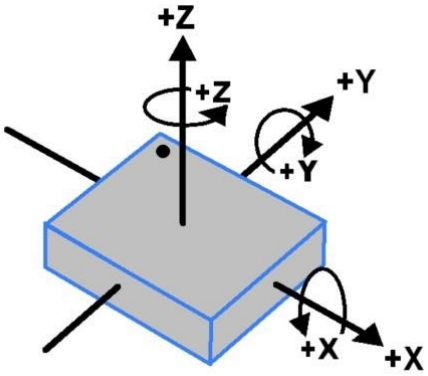

Figure 8. 3-axis gyroscope [10].

Figure 8 shows the 3-axis gyro in the MPU6050 module detecting rotational velocity along the X, Y, Z axes.

2.6 Magnetometer

The MEMS Magnetometer measures the earth magnetic field and this is achieved through the Hall Effect or an effect known as MagnetoResistive Effect. Majority of the sensors use the Hall Effect in figure 9 below.

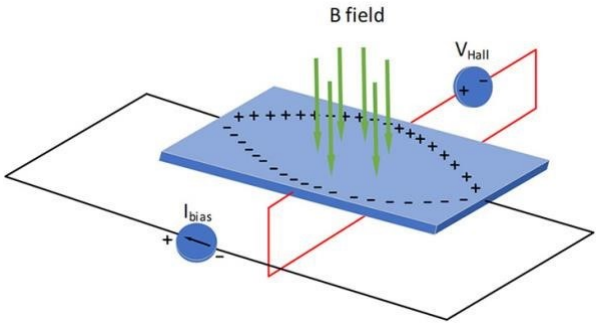

Figure 9. Magnetometer showing electron flaw in accordance with the Hall Effect [11].

Given a conductive plate as shown in figure 9 and a current flowing through it, the electrons flow from the negative terminal to the positive terminal. However, if a magnetic field is introduced near the plate, this would cause a disruption in the flow of electrons and this would cause their deflection to one side of the plate and the positive poles to the other side of the plate [12]. A voltage dependent on the magnetic field strength and its direction will be as a result and can be measured by a metre between the two sides of the plate.

2.7 MPU6050 Orientation and Rotation Polarity

During the rotation of the gyroscopes about the axes, an effect known as the Coriolis effect causes a vibration in the MPU6050 amplifying the resulting signal. The resulting signal is demodulated and filtered to produce a voltage proportional to the angular rate which is then digitized using a 16-bit analog to digital converter to sample each axis. The 16-bit analog to digital converter digitizes the output and the full range of acceleration being +/-2g, to +/-16g so when the device is placed on a flat surface, a result of 0g on X and Y and +1g on Z axis is expected.[13.] The standard unit is the g (gravity force) unit. The angular velocity along each axis is measured in degrees per second unit.

2.7.1 3-Axis Accelerometer

Correspondingly to the 3-axis gyroscope, the MPU6050 consists of the 3-axis accelerometer as shown in figure 10.

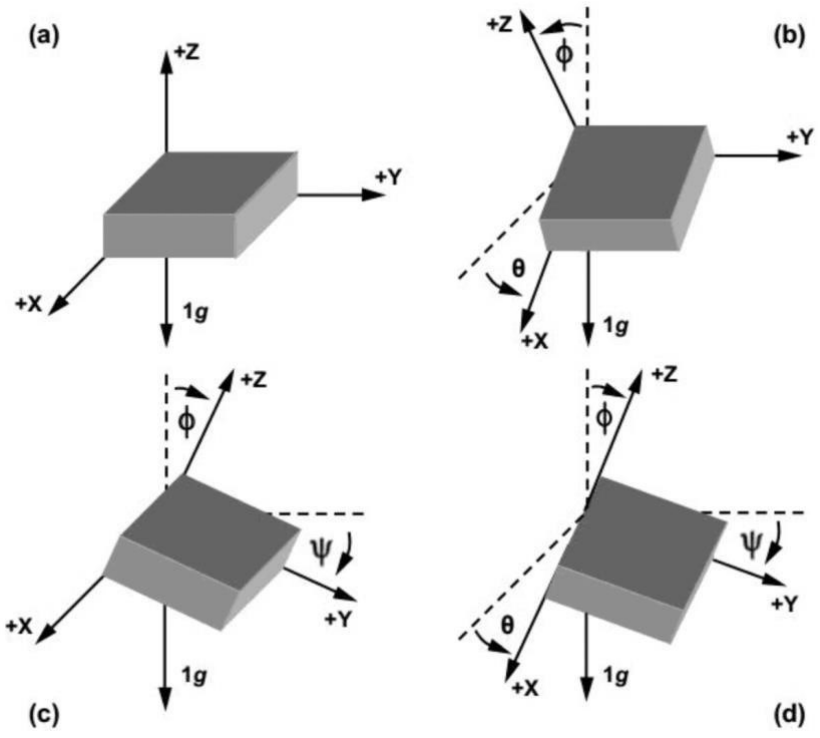

Figure 10. 3-axis accelerometer detecting angle of tilt along X, Y and Z [13].

2.7.2 Temperature Sensor

The MPU6050 contains an on-chip temperature sensor, and its outputs are digitized using ADC and can be read from the temperature sensor data register.

2.7.3 Digital Motion Processor

The digital motion processor is used to compute motion processing algorithms by transferring data from the gyroscope and accelerometer and processes the data and provides motion data like roll, pitch, and yaw angles.
The data results can then be read from the digital motion processor registers. [14.]

2.7.4 MPU6050 Module

The MPU6050 board to be used in this project contains 8 pins and is in the figure 11 below;

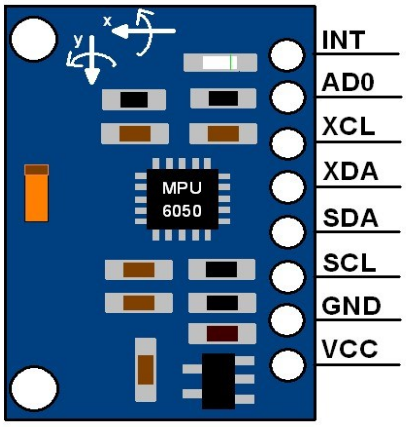

Figure 11. MPU6050 module [8].

The module pins are described below as;

VCC- Power supply pin connected to +5v of DC from power supply equipment.
GND- Ground connection pin.
SCL- Serial clock pin to the microcontroller's corresponding SCL pin.
SDA- Serial data pin to the microcontroller's corresponding SDA pin.
XDA- This is known as the auxiliary serial data pin and its function is to connect other inter-integrated (I2C) circuit interface enabled sensors SDA pin to MPU6050 and

similarly the auxiliary serial clock pin (XCL) is used to connect other I2C interface enabled sensors SCL pin to the module.
ADO is referred to as the I2C slave address LSB pin which when connected to the VCC, the slave address changes and is read as logic one.
Finally, the INT is the Interrupt digital out pin.

2.8 PWM and Servo Motors

The angular position of the output shaft of a servomotor is determined by a special form of pulse width modulation (PWM) and they are usually used in applications that require high precision. The servo motor shaft is designed to reach an operating range of 0 to 180 degrees, making it suitable for controlling the rods of the platform. In the stabilization of the platform, the aspect of speed in the servo motors is crucial. From the inverse proportion of the Torque-speed curve by the Servos, low speed results in a high torque and a high speed produces a low Torque [15].

To ensure that the motor is operating at the correct speed while still producing enough torque, the selection and control of the motor must be taken into account. Therefore, the servo is regulated through pulse width modulation (PWM).

Signals to the servo can only be either high (5V) or low (5V) at any given time, but by adjusting the time between high or low, high, or low signals can be sent. Duty cycles show how much the high proportionality in the signal and lower percentage.

As illustrated in figure 12 a duty cycle of 10 percent indicates that the digital signal is high 10 percent of the time. The servo angle is defined by the length of the pulse [1-2] ms with a 20 ms period.

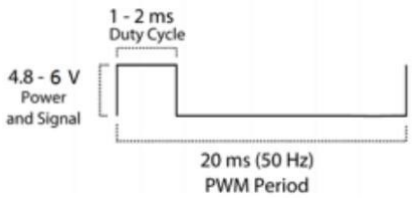

Figure 12. Servo motor duty cycle [16].

3. Materials and Design

This section describes a step-by-step procedure of the practical part of the design and construction of the stabilizing platform, as well as all the components required. It also examines how the concepts described and explained above contribute towards achieving the goal of the project.

3.1 Arduino

In this project, the Arduino uno is selected, a microcontroller based on the ATmega328P.
The arduino has 6 PWM pins and 14 digital pins in total both input/outputs, a ceramic resonator of 16MHz, USB connection as well as a reset button in figure 13 below
The Arduino is used as a communication interface for the MPU6050 module.

Figure 13. Image of Arduino [10].

3.2 MPU6050 with Arduino Interface.

With the previous introduction of the MPU6050 module in chapter 2.7.4, this subsection is going to examine interfacing this module with Arduino due to a large part of this project being based on this concept. Thankfully, this interface has been made easy with Jeff Rowberg's MPU6050 library for Arduino. [10.] In Jeff Rowberg's I2C library, there is a collection of uniformly documented classes for simple and intuitive interfaces to I2C devices.

This library is added to the Arduino IDE and a simple schematic, as shown in figure 14 is followed as below

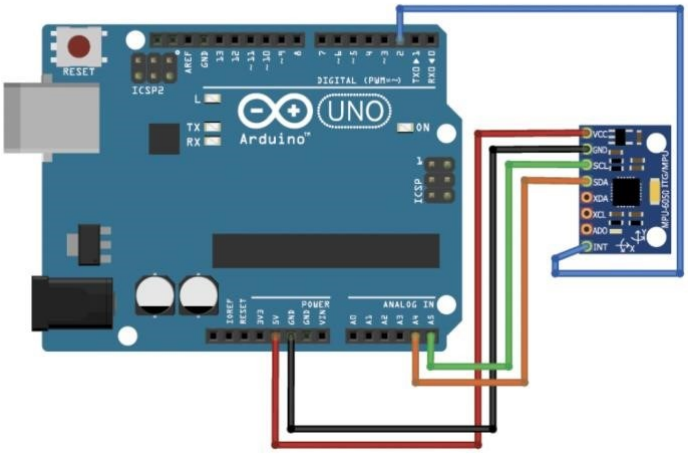

Figure 14. schematic showing an I2C communication between an Arduino uno and the MPU6050 [17].

The Arduino is connected to a PC through the USB connection cable and the Code Bender app is launched where the program of the Arduino is developed and then compiled.

3.2.1 MPU6050 Module Arduino Communication

For the I2C communication, numerous variables need to be defined and hence the Wire.h library [17] provides functions that enable the I2C communication. This library is also required for storing the data.
The wire library is initialized in the set-up section and the sensor reset. The sensor is reset through the power management register which also allows configuration of the power mode and clock source as shown in Table 1 below.

Table 1. power management register data sheet [18].

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6B	107	DEVICE _RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		

A full-scale range can also be selected using the configuration registers of both the accelerometer and gyroscope. However, in this project, the default values will be used as stated:
1000 degrees per second for the gyroscope
+/-8g range for the accelerometer. This is illustrated in the part of the code listing 1 below;

```
Wire.beginTransmission(MPU);
Wire.write(Ox1C);
Wire.write(0x10);
Wire.endTransmission(true);
Wire.begin Transmission(MPU);
Wire. write(0x1B);
Wire. write(0x10); Wire.endTransmission(true);
```

Listing 1. Configuration of registers [17].

3.2.2 ADXL345 Accelerometer

As discussed in the previous sections, the ADXL345 sensor is a 3-axis accelerometer, and it measures both static and dynamic forces of acceleration. Dynamic forces amongst many things can be brought about by vibrations and sudden or uniform movements to mention but a few.

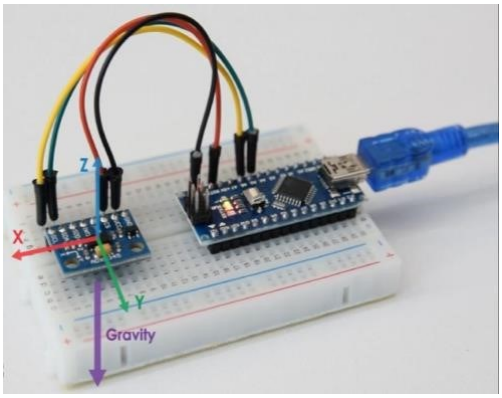

Figure 15. Mpu6050 module showing the 3-axis accelerometer [13].

The axes X, Y and Z show the direction of the forces acting on the accelerometer as shown in figure 15, the force of gravity acts opposite of the Z- axis pointing upwards. In this case the accelerometer is positioned flatly on the breadboard. The Z-axis output on the serial monitor will be 1g or 256. Since the gravitational force is perpendicular to the axes X and Y, their outputs are consequently going to be zero.

Once the position of the module changes, for instance turned to face upside down, then the value of the output of the Z-axis will change to -1g. Hence the orientation due to gravity can vary from -1g to +1g. This is illustrated in the following chapters.
The standard unit measurement for accelerometer sensors is gravity in other words ‘g’. However, acceleration is measured in meters per square seconds. It is known that the value of g is 9.8 meters per square second. (ms^-2)

3.3 Servo Motors

A servo motor can be defined as a self-contained electrical device that rotates parts of a machine with high efficiency and with great precision [15].
A motor with this capability can be rotated at an angle and position that a regular motor cannot, and it combines a regular motor with a sensor for positional feedback as well. The basic servo motors are used only in position sensing. In order to provide position and speed feedback, the motor is attached to an optical or capacitive encoder. The simplest case is to measure the position and the measurement is compared to the external input of the controller, the command position. If the output position is different from that required, an error signal is generated, which causes the motor to rotate either direction. [16.]

The Servo motors introduced to the arduino-Mpu6050 set up in this project are the MG90S Micro Servos as shown in figure 16. These Servo motors contain a high output power and are small and lightweight as they come in the Micro size.

Figure 16. A racestar MG90S servo motor [15].

These motors have a high stall torque and have a rotation of 180 degrees or 90 degrees in each direction. The servo motors have been implemented in numerous

projects such as RC projects due to their high quality and performance. They have an operating voltage of 4.8v-6.0volts. The stall torque is 1.8kg/cm at 4.8v and 2.2 kg/cm at 6v.

The Servo motor is connected to the Arduino-Mpu6050 module as shown in the circuit diagram figure 17

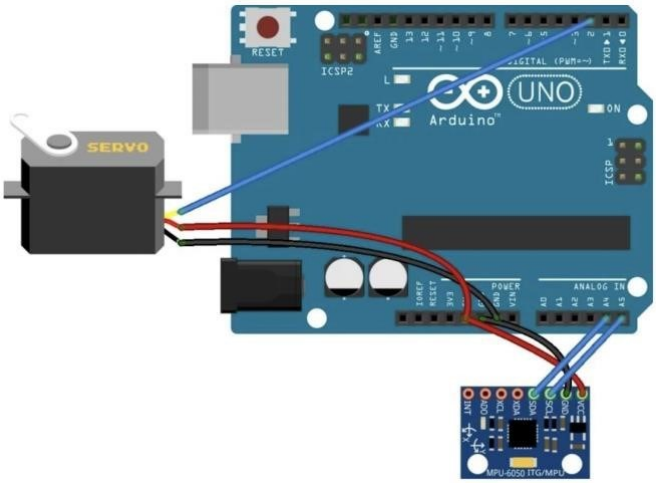

Figure 17. Arduino-Mpu6050 interface with servo motor [19].

The pins A4 and A5 on the Arduino as shown in figure 17 are used for the purpose of I2C communication and therefore the SCL and SDA pins on the MPU6050 are connected to A5 and A4 on the Arduino.

The servo motor has three wires and is connected to the Arduino as illustrated in figure 17. The Arduino code for the entire circuit is compiled and uploaded to begin the testing of the movement and directions of the Servo motors.

4. Practical Work

4.1 Plastic Enclosure

A plastic enclosure is necessary for the gimbal circuit constructed thus far and is designed following samples adapted from howtomechatronics with Autofusion desk software.

The plastic enclosure is designed so the circuit is placed and fitted inside and for clear and apparent demonstration of the operation of the self-stabilizing platform. An action camera such as a small go pro camera for instance or any other objects may be placed on the stabilized plate

4.2 3D Designing

As mentioned in the previous chapter, the plastic enclosure was designed in the software Auto Fusion desk [20]. There are ten different parts in the list and diagrams below.

-Battery holder
-Servo MG960S (3pcs)
-MPU6050 model

-Base (Roll Servo)

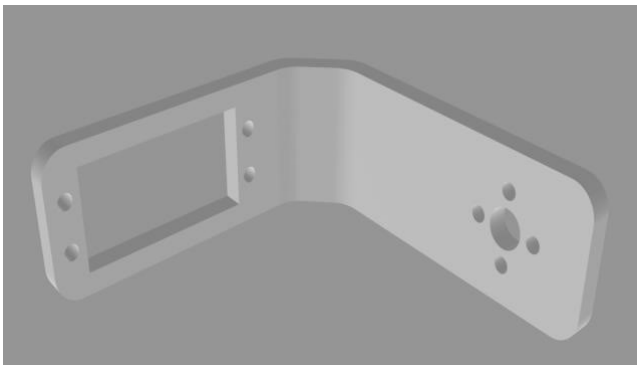

Figure 18. Roll [20].
The roll Servo is attached to the base printed part as shown in figure 18.

-Base Yaw Servo

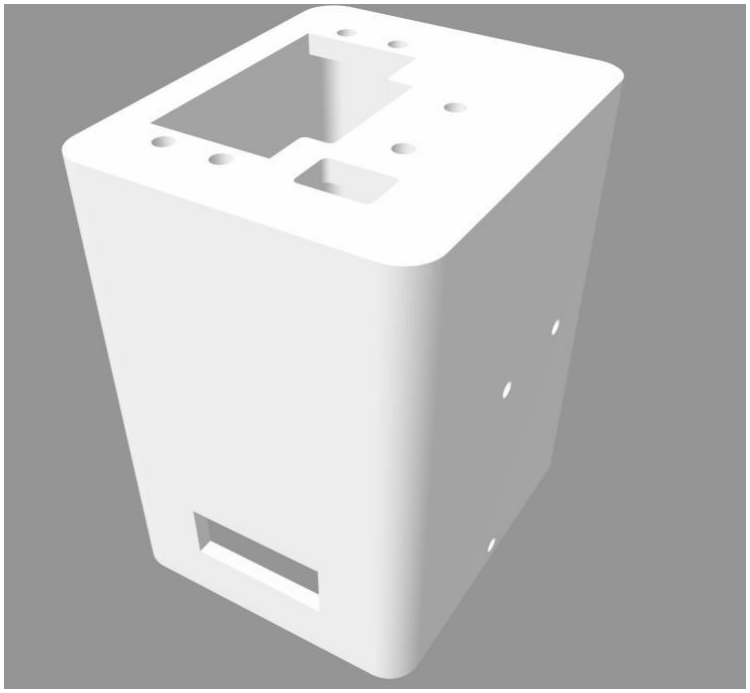

Figure 19. Yaw [20].

Inside the base yaw in figure 19 is where the battery and the arduino are placed with the connection wires through the holes.

-Bottom cover

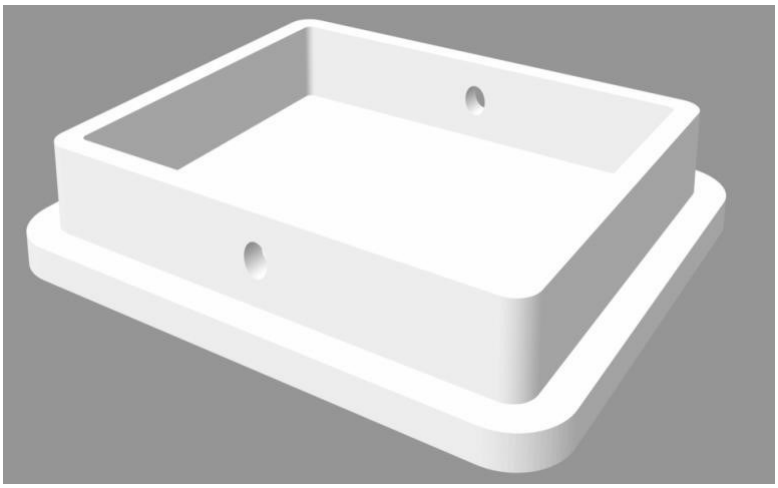

Figure 20. Bottom [20].

The bottom cover as shown in figure 12.3 is closing the plastic enclosure as a seal

-Base pitch Servo

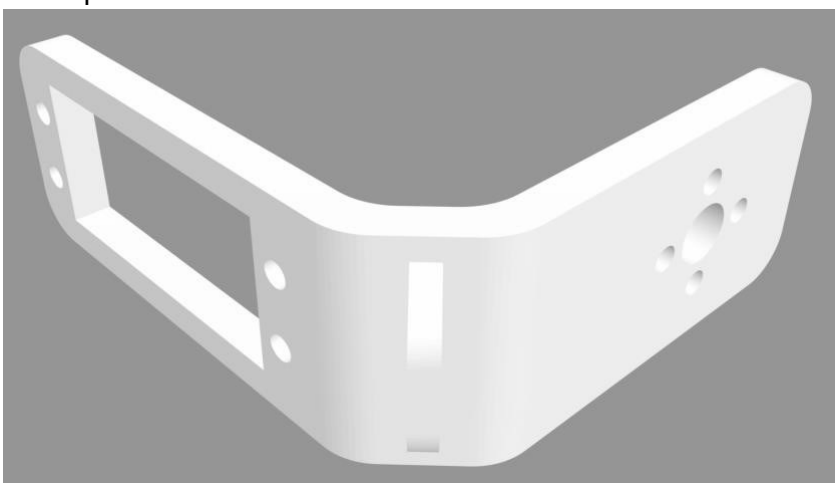

Figure 21. Pitch [20].

The pitch servo is connected with the screws to the base and is shown in figure 21.

-A complete 3D design diagram of the plastic enclosure of the self-stabilizing platform.

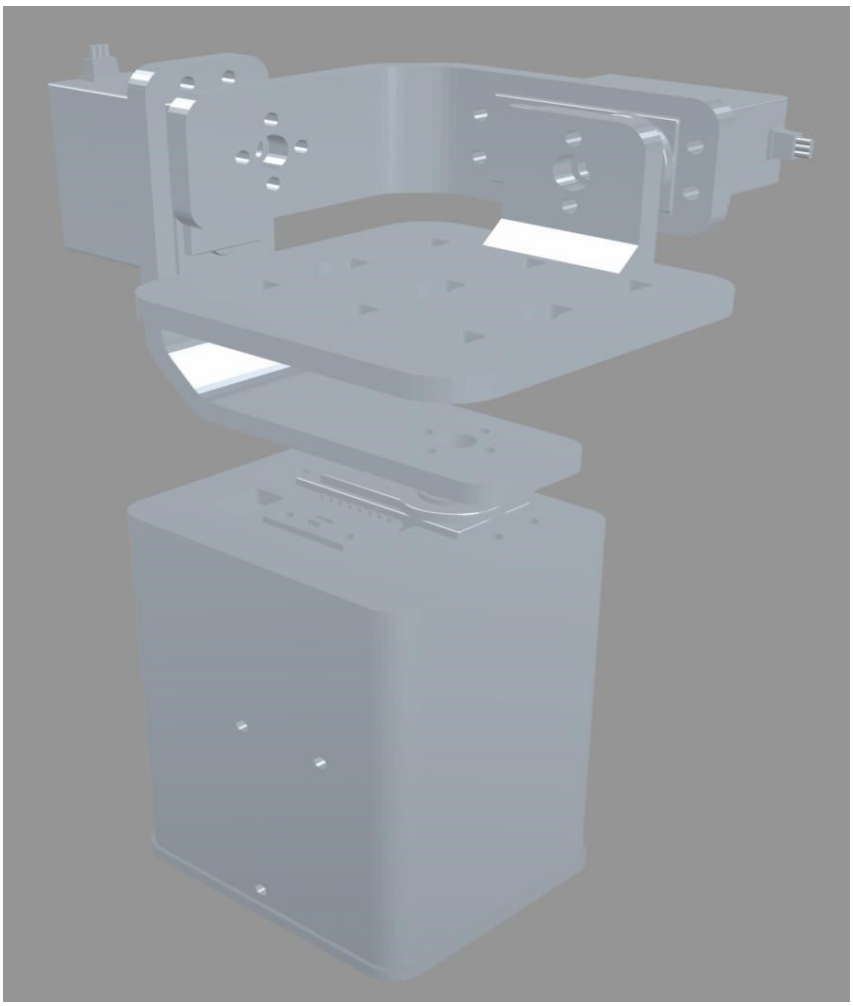

Figure 22. Plastic enclosure design [20].

The printed parts are all assembled as illustrated in figure 22 as a self-stabilizing platform enclosure.

5. Hardware

5.1 Hardware Block Diagram

The block diagram in figure 23 shows the hardware component connections and communication at various stages. The block diagram in simple terms is showing how the circuit is powered by the battery to the arduino whose input voltage is 7-12 volts. This was followed by the ArduinoMpu6050 interface which spins the motors in different directions according to the programming. The circuit was then assembled and placed into the 3D printed plastic enclosure after which it was able to self-stabilize a small action camera or any light object placed onto the balancing plate.

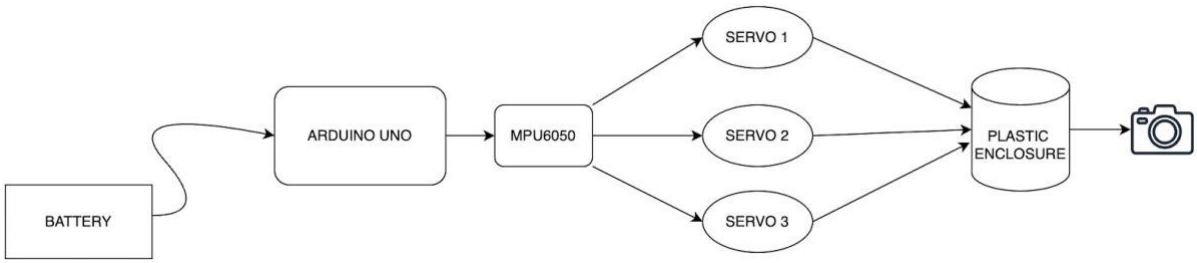

Figure 23. Hardware block diagram drawn in app diagram.

5.2 List of Components

Table 2 showing a list of components that were used in this thesis project.

Component	Amount	Parameters
Li-ion Battery	2pcs	1.5v each
Arduino Uno	1pc	7-12v input
MPU6050 module	1pc	-
MG90S	3pcs	4.8-6 volts
jumper wires	20pcs	-
USB	1pc	-

USB cable	1pc	-
breadboard	1pc	-
Yaw Servo printed part	1pc	-
bottom cover printed	1pc	-
Roll and pitch servo printed part	2pcs	-
Screws and nuts	10 pcs	-

5.3 Assembly

A set up of the components shown in figure 24. The circuit was assembled like shown below in figure 24.
The circuit was then inserted into the 3D printed plastic enclosure in chapter 3.1.2.

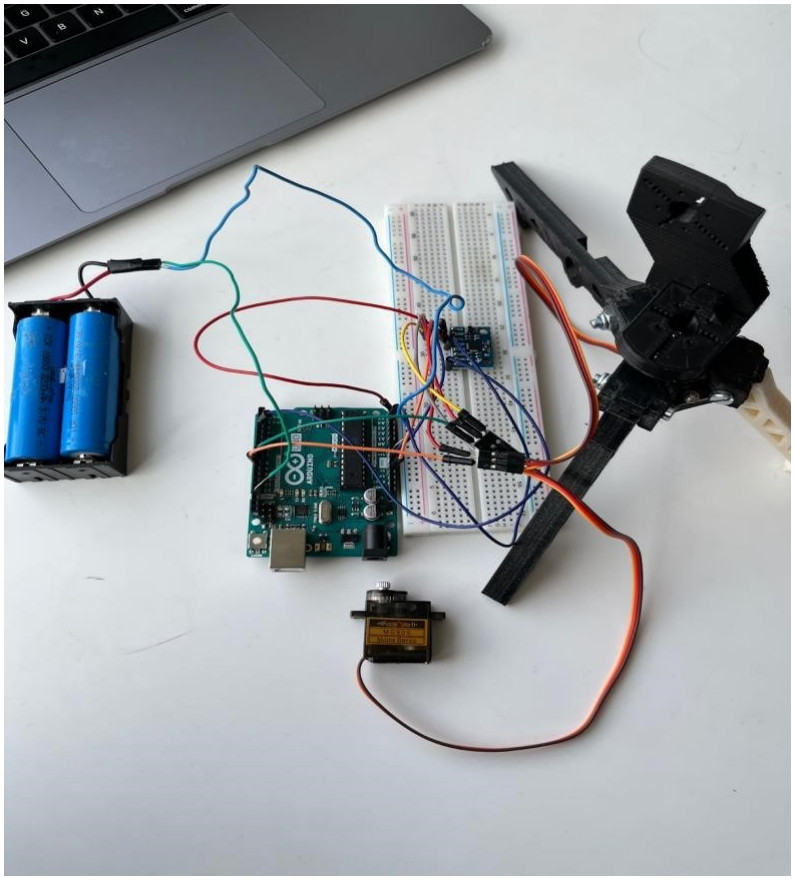

Figure 24. Assembled circuit without the plastic enclosure.

6. Software

6.1 Software Block Diagram

The block diagram in figure 14 shows the phased execution of the code in appendix 1 and the different functions of the components at each step.
The MPU6050 module is programmed through the arduino for its position to be determined by outputting three [x, y, z] values onto the serial monitor.
However due to the drifting of the gyroscope overtime as explained in chapter 2.4, there is an expected error in the output which necessitates the use of the if function in the block diagram (figure 25).

If there is an error in the coordinate outputs, the data is false and therefore is sent back to the MPU6050 module and through the complementary filter in the program. If, however there appears to be no error in the outputs then the outcome is true and therefore the respective coordinates are sent to the corresponding servos. Each Servo is assigned a coordinate in the program as explained in the following chapter and in Appendix 2

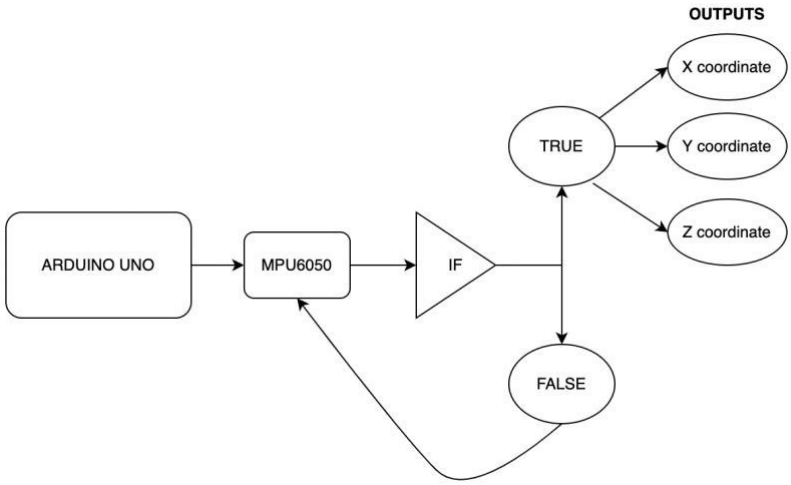

Figure 25. Software block diagram drawn in app diagram.

6.2 Mpu6050 Programming

The MPU6050 Arduino interface code is written and compiled in C and C++.
This is a modified version of the previously pre-written interface code by the online Arduino and mechatronics community and has been designed to execute the goal of this project.
Different parts of the code perform different functions as explained in the following sections.

Wire library

```
#include <Wire.h>
const int MPU = 0x68; // MPU6050 I2C address
float AccX, AccY, AccZ; float GyroX, GyroY,
GyroZ;
float accAngleX, accAngleY, gyroAngleX, gyroAngleY, gyroAngleZ;
float roll, pitch, yaw;
float AccErrorX, AccErrorY, GyroErrorX, GyroErrorY, GyroErrorZ;
float elapsedTime, currentTime, previousTime; int c = 0;
```

Listing 2. Including the wire library [17].

Firstly, the wire library is included and then variables for storing the data are defined as shown in listing 2.

Proceeding with the set-up section below, the wire library is then initialized, and the sensor is reset to the power management registers in listing 3. This is achieved by extracting the register address from the data sheet of the sensor in table 3.

```
void setup() { Serial.begin(19200);
Wire.begin();
Wire.beginTransmission(MPU);
Wire.write(0x6B);
Wire.write(0x00);
Wire.endTransmission(true);
```

Listing 3. Communication initialization [17].

Table 3. Register address from the sensor datasheet [18].

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6B	107	DEVICE _RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		

Table 4. Accelerometer readings data sheet [18].

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT[7:0]							

The main loop consists of accelerometer data reading as shown in table 4. The data for each given axis is stored as bytes or registers and the addresses of the registers are shown in the datasheet of the sensor.

```
void loop() {

Wire.beginTransmission(MPU);
Wire.write(0x3B);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;
AccY = (Wire.read() << 8 | Wire.read()) / 16384.0; AccZ
= (Wire.read() << 8 | Wire.read()) / 16384.0;
accAngleX = (atan(AccY / sqrt(pow(AccX, 2) + pow(AccZ, 2))) *
180 / PI) - 0.58;
accAngleY = (atan(-1 * AccX / sqrt(pow(AccY, 2) + pow(AccZ, 2)))
* 180 / PI) + 1.58; // AccErrorY ~(-1.58)
```

Listing 4. Accelerometer data reading and calculating roll and pitch values [17].

In order to acquire all the readings as shown in listing 4, we begin by reading the first register and then using a request form function in listing 5, all the 6 registers are read for the x, y and z axis.

```
Wire.requestFrom(MPU, 6, true);
```

Listing 5. Request form function [17].

The code reads the data for each register and the outputs are combined appropriately to obtain the correct values since they are 2s complements. All raw values are divided by 16384 to obtain output values between the range +2g and -2g suitable for calculating the angles.

Finally using the formulae (2) and (3) below, the roll and pitch data is calculated.

Accelerometer angle in the x-axis,

$$\alpha x = (\text{AccYpow}(\text{AccX}, 2) + \text{pow}(\text{AccZ}, 2)))*180/ - 0.58. \quad (2)$$

Accelerometer angle in the y-axis,

$$\alpha y = (-1 * \text{AccXpow}(\text{AccX}, 2) + \text{pow}(\text{AccZ}, 2)))*180/ + 1.58 \quad (3)$$

Similarly, the gyroscope data is obtained using the same method in the part of the code as shown listing 5 below.

```
previousTime = currentTime; currentTime = millis();
elapsedTime = (currentTime - previousTime) / 1000;
Wire.beginTransmission(MPU);
Wire.write(0x43);
Wire.endTransmission(false);  Wire.requestFrom(MPU, 6, true);
GyroX = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroZ = (Wire.read() << 8 | Wire.read()) / 131.0;
```

Listing 5. Reading registers and storing axis values [17].

Here the 6 gyroscope registers are read, and their outputs combined appropriately. These outputs are then divided by their previous sensitivity (131.0) to get the result in degrees per second.

Correction of the output values in listing 6.

```
GyroX = GyroX + 0.56; // GyroErrorX ~(-0.56)
GyroY = GyroY - 2; // GyroErrorY ~(2) GyroZ
= GyroZ + 0.79; // GyroErrorZ ~ (-0.8)
```

Listing 6. Calculated error values
[17].

As the outputs are degrees per second, they are multiplied by the elapsed time to obtain just degrees. The time value is captured before each iteration using the millis function.

A complementary filter is introduced in fusion of the accelerometer and gyroscope data as shown in listing 7.

```
roll = 0.96 * gyroAngleX + 0.04 * accAngleX; pitch
= 0.96 * gyroAngleY + 0.04 * accAngleY;
```

Listing 7. Combining accelerometer and gyroscope angle values through the complementary filter [17].

So 96% of the gyroscope data is used due to its accuracy and inability to be affected by external forces; however, the gyroscope still introduces errors in the data in an extended period of time. Therefore, 4% of the data from the accelerometer is used long term and it is enough to eliminate the gyroscope drift error.

After the code is verified and uploaded, the values gyroAngleX, gyroAngleY and yaw are displayed on the serial monitor and the extract below from the code shows how these values are obtained. These values indicate the position of the Mpu6050 and so they change accordingly with the position of the module and the values are calculated in the code as per listing 8.

```
gyroAngleX = GyroX * elapsedTime;
gyroAngleY = GyroY * elapsedTime;
yaw = GyroZ * elapsedTime;
```

Listing 8. Raw angle values in degrees.

Initially there were numerous inaccuracies with the results displayed on the serial monitor and this due to the looping of the value of gyroAngleX in listing 8 to compute a new value i.e

```
gyroAngleX = gyroAngleX + GyroX * elapsedTime
```

Listing 9. Initial looping of the gyroAngleX value [17].

and so, by omitting gyroAngleX from the loop, a more precise and accurate display of the values on the serial monitor was observed as shown in figures 26 and 27.

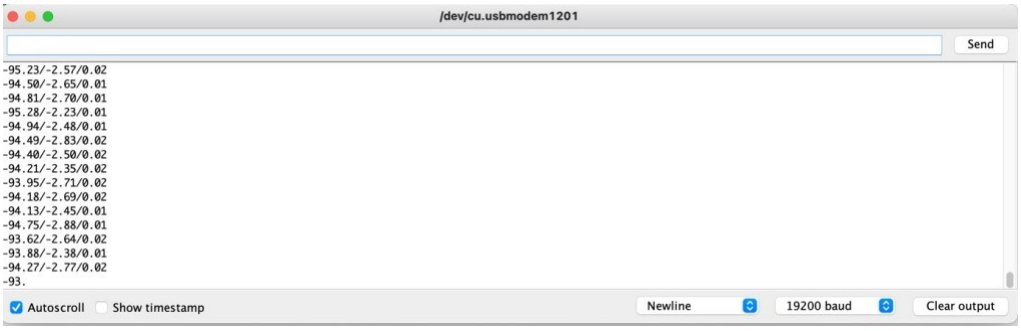

Figure 26. Image of display of serial monitor

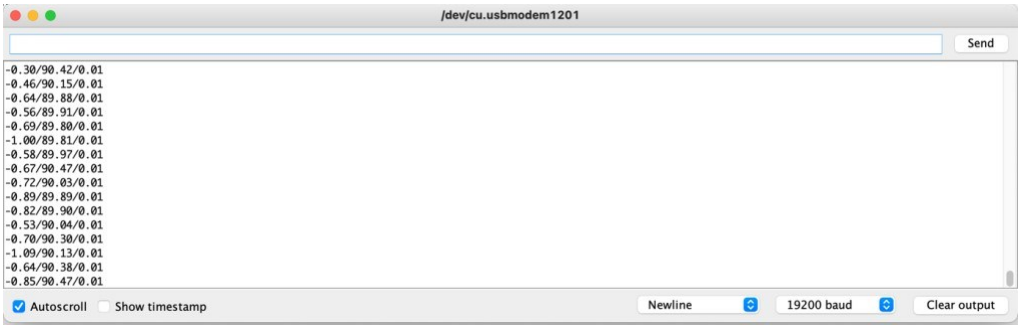

Figure 27. Image of the display of serial monitor

The figures 26 and 27 of the serial monitor display show the values being printed, the first value being gyroAngleX ‘roll’ gyroAngleY ‘pitch’ and the Yaw direction
This means that the MPU6050 Arduino interface has been successfully established and operates as expected from the code.

The last part of the code in listing 10 below is for the calculation of the error correction values from the accelerometer and gyro data. In this calculation, the ‘calculate_IMU_error’ can be referred to or assigned as a custom function while the sensor is in a flat still position.

```
void calculate_IMU_error() { while
(c < 200) {
Wire.beginTransmission(MPU);
Wire.write(0x3B);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
AccX = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
AccY = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
// Sum all readings
AccErrorX = AccErrorX + ((atan((AccY) / sqrt(pow((AccX), 2) +
pow((AccZ), 2))) * 180 / PI));
AccErrorY = AccErrorY + ((atan(-1 * (AccX) / sqrt(pow((AccY), 2)
+ pow((AccZ), 2))) * 180 / PI)); c++;
}
AccErrorX = AccErrorX / 200;
AccErrorY = AccErrorY / 200;
```
Listing 10. ‘Calculate_IMU_error’ function for error correction [17].

A reading of 200 outputs is obtained and the sum of the outputs is divided still by 200 to achieve the error value and whilst the sensor lies in a flat position still, the expected value should be 0 and finally using the serial print function, the values roll, pitch and yaw values are displayed on the monitor.

In order to obtain a 3d visualization as shown in figure 28, the data sent through the serial port is allowed into an online 3d process developing editor.

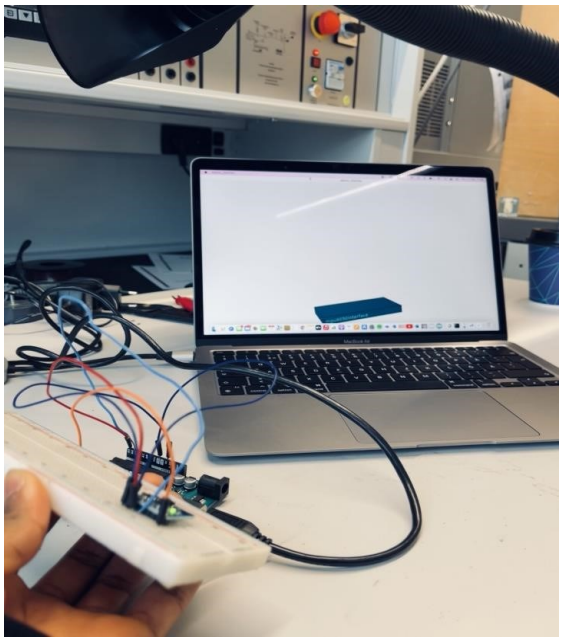

Figure 28. A 3d visualization of the Mpu6050 module on the screen.

6.3 Self Stabilizing Platform Code

The Arduino code for this section is a modification of the MPU6050_DMP6 example from the library and in this code, the outputs, yaw, pitch and roll are being used as shown in listing 11.

```
#ifdef OUTPUT_READABLE_YAWPITCHROLL mpu.dmpGetQuaternion(&q,
fifoBuffer); mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
ypr[0] = ypr[0] * 180 / M_PI; ypr[1] = ypr[1] * 180 / M_PI;
ypr[2] = ypr[2] * 180 / M_PI;
```

Listing 11. Acquiring Yaw, Pitch and Roll values.[21]

Once the values are obtained, they are converted from radians to degrees.
The sensor requires self-calibration and so the first 300 readings are ignored because this is known as its self-calibration process as shown in listing 12.
An alternative option is to wait for the sensor to self-calibrate.

```
if (j <= 300) { correct = ypr[0];
j++;
```

Listing 12. Self-calibration process [21].

The yaw starts at a random value unlike the roll and the pitch and so the first value after 300 readings for yaw is recorded and it's set to 0.
The values of roll, pitch and yaw are then mapped from sensor form to values suitable for servo control i.e. from the range [-90 90] to [0, 180] in listing 13.

```
{ ypr[0] = ypr[0] - correct;
int servo0Value = map(ypr[0], -90, 90, 0, 180); int
servo1Value = map(ypr[1], -90, 90, 0, 180);
int servo2Value = map(ypr[2], -90, 90, 180, 0);
```

Listing 13. setting yaw to 0 degrees [21].

Finally, these new values are sent to the servo motors as control signals using the 'write' function as in listing 14 below:

```
servo0.write(servo0Value); servo1.write(servo1Value);
servo2.write(servo2Value);
} #endif
}}
```

Listing 14. Controlling the servos according to the MPU6050 orientation [21].

In this case, the yaw servo was disabled so this could operate as a camera gimbal.

7. Results

The circuit in chapter 5.3 was assembled with the 3D printed plastic enclosure illustrated in chapter 4.2. A functional DIY self-stabilizing platform was the result as shown in figure 29.

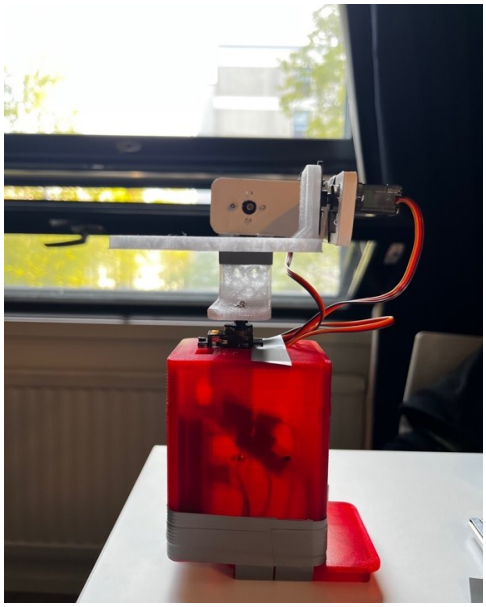

Figure 29. A fully assembled self-stabilizing Platform

The gimbal is held at an angle and the balancing plate is observed to retain its horizontal position shown in figure 30 indicating that the self-stabilizing platform operated as expected

Figure 30. self-balancing plate when the gimbal is held at an angle

8. Conclusion

This project evaluated two hardware module components and their associated functions in the concept of DIY self-stabilization. There was a considerable amount of learning about the capabilities and limitations of the MPU6050 module and how it is affected by phenomena such as Inertia. In the initial setup of the module, a large variance in the x, y and z direction parameters was determined during the compilation of the code, which allowed testing to greater extents in order to achieve the goal of this project.

The project was executed and divided into five steps which involved circuit construction set ups, programming, testing, 3d printing and finally assembling. There was plenty of information regarding this project in the arduino and mechatronics community online and therefore the focus was to learn and understand to a greater capacity the operation of both the software and hardware concepts involved in this project.

The goal of the project was achieved in the end, a simple gimbal was constructed, and the balancing plate observed to retain a horizontal stabilized position when the gimbal was held and moved at different angles and in different directions. The platform can be useful in day-to-day activities such as photography and videography through balancing small action cameras.

This platform can also be improved greatly by using bigger and more advanced Servo motors and a metallic enclosure instead of 3D printed plastic material which limits smooth movement of the gimbal in this project. Nevertheless, the project was successful and an interesting learning experience.

References

1. Dolev, S., n.d. *Self-Stabilization*. [online] Ki.pwr.edu.pl. Available at: <https://ki.pwr.edu.pl/lemiesz/info/SelfS.pdf> [Accessed 22 March 2022].

2. En.wikipedia.org. n.d. *Self-stabilization - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Self-stabilization> [Accessed 10 February 2022].

3. Splunk. 2005. *What Are Distributed Systems? An Introduction | Splunk*. [online] Available at: <https://www.splunk.com/en_us/data-insider/what-are-distributedsystems.html> [Accessed 20 March 2022].

4. En.wikipedia.org. 2008. *Fault tolerance - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Fault_tolerance> [Accessed 30 March 2022].

5. Popelka, "A self-stabilizing platform," Proceedings of the 2014 15th International Carpathian Control Conference (ICCC), 2014, pp. 458-462, doi: 10.1109/CarpathianCC.2014.6843648.

6. Johan Danmo, J., 2017. [online] Diva-portal.org. Available at: <https://www.diva-portal.org/smash/get/diva2:1200521/FULLTEXT01.pdf> [Accessed 22 March 2022].

7. En.wikipedia.org. n.d. *Inertial measurement unit - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Inertial_measurement_unit> [Accessed 21 February 2022].

8. Gyro), M., n.d. *MPU6050 GY521 Cheap 6DOF IMU (Accelerometer & Gyro) IMUs | JSumo.com*. [online] https://www.jsumo.com/. Available at: <https://www.jsumo.com/mpu6050-gy521-cheap-6dof-imu-accelerometer-gyro> [Accessed 1 April 2022]

9. Karlsson, A. and Cresell, J., 2016. *Self-stabilizing platform*. [online] Divaportal.org. Available at: <https://www.diva-portal.org/smash/get/diva2:957123/FULLTEXT01.pdf> [Accessed 1 March 2022].

10. Arduino Project Hub. 2019. *What Is MPU6050?*. [online] Available at: <https://create.arduino.cc/projecthub/CiferTech/what-is-mpu6050-b3b178> [Accessed 14 March 2022].

11. W. Collins, D., 2021. *How do Hall effect sensors work and where are they used?*. [online] Motioncontroltips.com. Available at: <https://www.motioncontroltips.com/how-do-hall-effect-sensors-work-whereare-they-used-in-motion-applications/> [Accessed 2 May 2022].

12. How To Mechatronics. 2019. *What is MEMS? Accelerometer, Gyroscope & Magnetometer with Arduino*. [online] Available at: <https://howtomechatronics.com/how-it-works/electrical-engineering/mems-accelerometer-gyrocope-magnetometer-arduino/> [Accessed 25 May 2022].

13. Wings, E., 2019. *ADXL335 Accelerometer Module | Sensors & Modules*. [online] Electronicwings.com. Available at: <https://www.electronicwings.com/sensors-modules/adxl335-accelerometermodule> [Accessed 2 March 2022].

14. Kok, M., Hol†, J. and Sch¨on‡, T., 2018. *Using Inertial Sensors for Position and Orientation Estimation*. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1704.06053.pdf> [Accessed 5 April 2022].

15. Gastreich, W., 2018. *What is a Servo Motor?' and How it Works | RealPars*. [online] PLC Programming Courses for Beginners | RealPars. Available at: <https://realpars.com/servo-motor/> [Accessed 19 April 2022].

16. DatasheetCafe. 2021. *MG90S Datasheet - Micro Servo Motor*. [online] Available at: <http://www.datasheetcafe.com/mg90s-datasheet-motor/> [Accessed 2 April 2022].

17. How To Mechatronics. 2019. *Arduino and MPU6050 Accelerometer and Gyroscope Tutorial*. [online] Available at: <https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050accelerometer-and-gyroscope-tutorial/> [Accessed 19 April 2022].

18. InvenSense.tdk.com. 2013. *MPU-6000 and MPU-6050 Register Map and Descriptions*. [online] Available at: <https://invensense.tdk.com/wpcontent/uploads/2015/02/MPU-6000-Register-Map1.pdf> [Accessed 6 February 2022].

19. aqibdutt, M., 2018. *Arduino-Servo motor interface.* [online] Available at: <https://maker.pro/arduino/tutorial/howto-control-a-servo-with-an-arduino-and-mpu6050> [Accessed 10 March 2022].

20. Thangs. 2022. *Self balancing platform | 3D model | Thangs*. [online] Available at: <https://thangs.com/designer/HowToMechatronics/3d-model/Self-balancingplatform-43942> [Accessed 11 March 2022].

21. How To Mechatronics. 2019. *DIY Arduino Gimbal | Self-Stabilizing Platform*. [online] Available at: <https://howtomechatronics.com/projects/diy-arduinogimbal-self-stabilizing-platform/ [Accessed 22.5 March]

Appendices

Appendix 1
MPU6050 ARDUINO CODE

```
#include <Wire.h> const
int MPU = 0x68; float
AccX, AccY, AccZ; float
GyroX, GyroY, GyroZ;
float accAngleX, accAngleY, gyroAngleX, gyroAngleY, gyroAngleZ;
float roll, pitch, yaw;
float AccErrorX, AccErrorY, GyroErrorX, GyroErrorY, GyroErrorZ;
float elapsedTime, currentTime, previousTime; int c = 0; void
setup() {
Serial.begin(19200);
Wire.begin();
Wire.beginTransmission(MPU);     MPU6050
Wire.write(0x6B);
Wire.write(0x00);
Wire.endTransmission(true);
Wire.beginTransmission(MPU);
Wire.write(0x1C);
Wire.write(0x10);
Wire.endTransmission(true);
Wire.beginTransmission(MPU);
Wire.write(0x1B);
Wire.write(0x10);
Wire.endTransmission(true);
delay(20);
calculate_IMU_error();
delay(20); }
void loop() {
Wire.beginTransmission(MPU); Wire.write(0x3B);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;
AccY = (Wire.read() << 8 | Wire.read()) / 16384.0; AccZ
= (Wire.read() << 8 | Wire.read()) / 16384.0;
accAngleX = (atan(AccY / sqrt(pow(AccX, 2) + pow(AccZ, 2))) *
180 / PI) - 0.58;

accAngleY = (atan(-1 * AccX / sqrt(pow(AccY, 2) + pow(AccZ, 2)))
* 180 / PI) + 1.58; previousTime
= currentTime; currentTime =
millis();
elapsedTime = (currentTime - previousTime) /
1000;   Wire.beginTransmission(MPU);
```

```
Wire.write(0x43);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
GyroX = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroY = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroZ = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroX = GyroX + 0.56; // GyroErrorX ~(-0.56)
GyroY = GyroY - 2; GyroZ
= GyroZ + 0.79;
gyroAngleX =  GyroX * elapsedTime; // deg/s * s = deg
gyroAngleY = GyroY * elapsedTime; yaw = GyroZ *
elapsedTime;
roll = 0.96 * gyroAngleX + 0.04 * accAngleX*30; pitch
= 0.96 * gyroAngleY + 0.04 * accAngleY*30;

Serial.print(roll);
Serial.print("/");
Serial.print(pitch); Serial.print("/");
Serial.println(yaw);
}
void calculate_IMU_error() { while
(c < 200) {
Wire.beginTransmission(MPU); Wire.write(0x3B);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
AccX = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
AccY = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0 ;

AccErrorX = AccErrorX + ((atan((AccY) / sqrt(pow((AccX), 2) +
pow((AccZ), 2))) * 180 / PI));
AccErrorY = AccErrorY + ((atan(-1 * (AccX) / sqrt(pow((AccY), 2)
+ pow((AccZ), 2))) * 180 / PI)); c++;
}
AccErrorX = AccErrorX / 200;
AccErrorY = AccErrorY / 200; c
= 0;
while (c < 200) {
Wire.beginTransmission(MPU); Wire.write(0x43);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
GyroX = Wire.read() << 8 | Wire.read();
GyroY = Wire.read() << 8 | Wire.read();
GyroZ = Wire.read() << 8 | Wire.read();
GyroErrorX = GyroErrorX + (GyroX / 131.0);
GyroErrorY = GyroErrorY + (GyroY / 131.0);
GyroErrorZ = GyroErrorZ + (GyroZ / 131.0);
c++; }

GyroErrorX = GyroErrorX / 200;
```

```
GyroErrorY = GyroErrorY / 200;
GyroErrorZ = GyroErrorZ / 200;
Serial.print("AccErrorX: ");
Serial.println(AccErrorX);
Serial.print("AccErrorY: ");
Serial.println(AccErrorY);
Serial.print("GyroErrorX: "); Serial.println(GyroErrorX);
Serial.print("GyroErrorY: ");
Serial.println(GyroErrorY);
Serial.print("GyroErrorZ: ");
Serial.println(GyroErrorZ);
}
```

Appendix 2

SELF-STABILIZING PLATFORM CODE

```
#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"
#include "MPU6050.h"
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
#include "Wire.h"
#endif
#include <Servo.h>
MPU6050 mpu;
Servo servo0;
Servo servo1;
Servo servo2;
float correct; int
j = 0;

#define OUTPUT_READABLE_YAWPITCHROLL

#define INTERRUPT_PIN 2 bool
blinkState = false;

bool dmpReady = false;
uint8_t mpuIntStatus;
uint8_t devStatus; (0 =
success, !0 = error)
uint16_t packetSize;
uint16_t fifoCount;
uint8_t fifoBuffer[64];

Quaternion q;
VectorInt16 aa;
VectorInt16 aaReal;
VectorInt16 aaWorld;
VectorFloat gravity;
float euler[3]; float
ypr[3];

uint8_t teapotPacket[14] = { '$', 0x02, 0, 0, 0, 0, 0, 0, 0, 0,
0x00, 0x00, '\r', '\n' };

volatile bool mpuInterrupt = false;
void dmpDataReady() { mpuInterrupt
= true;
}

//INITIAL SETUP void setup() {
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
```

```
Wire.begin();
Wire.setClock(400000);
#elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
Fastwire::setup(400, true);
#endif

Serial.begin(38400); while
(!Serial); mpu.initialize();
pinMode(INTERRUPT_PIN, INPUT);
devStatus = mpu.dmpInitialize();
mpu.setXGyroOffset(17);
mpu.setYGyroOffset(-69);
mpu.setZGyroOffset(27);
mpu.setZAccelOffset(1551); if
(devStatus == 0) {
mpu.setDMPEnabled(true);

attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady,
RISING);
mpuIntStatus = mpu.getIntStatus();

dmpReady = true;
packetSize = mpu.dmpGetFIFOPacketSize();
} else {
//Serial.println(F(")"));
}
servo0.attach(10);
servo1.attach(9); servo2.attach(8);
}  void loop() { if
(!dmpReady) return;

while (!mpuInterrupt && fifoCount < packetSize) { if
(mpuInterrupt && fifoCount < packetSize) { fifoCount
= mpu.getFIFOCount();
}
}
mpuInterrupt = false; mpuIntStatus
= mpu.getIntStatus(); fifoCount =
mpu.getFIFOCount();
if ((mpuIntStatus & _BV(MPU6050_INTERRUPT_FIFO_OFLOW_BIT)) ||
fifoCount >= 1024) {

mpu.resetFIFO();
fifoCount = mpu.getFIFOCount();
Serial.println(F("FIFO overflow!"));

} else if (mpuIntStatus & _BV(MPU6050_INTERRUPT_DMP_INT_BIT)) {
while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

mpu.getFIFOBytes(fifoBuffer, packetSize);
```

```
fifoCount -= packetSize;

#ifdef OUTPUT_READABLE_YAWPITCHROLL
mpu.dmpGetQuaternion(&q, fifoBuffer);
mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
ypr[0] = ypr[0] * 180 / M_PI; ypr[1] =
ypr[1] * 180 / M_PI; ypr[2] = ypr[2] * 180
/ M_PI; Serial.println(ypr[0]); if (j <=
300) { correct = ypr[0]; j++;
}
// After 300 readings else
{
ypr[0] = ypr[0] - correct;
int servo1Value = map(ypr[1], -90, 90, 0, 180); int
servo2Value = map(ypr[2], -90, 90, 180, 0);

servo1.write(servo1Value); servo2.write(servo2Value);
}

#endif
}
}
```