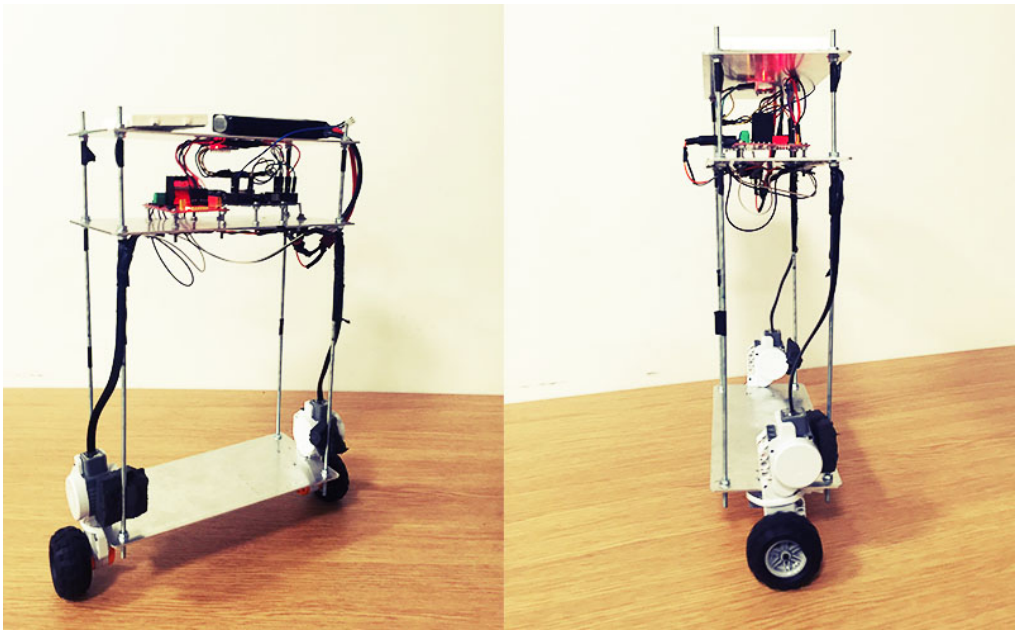
BEng (Hons) Computer Systems Engineering

School of Engineering and Digital Arts

Project and Implementation for

Two-Wheeled Balancing Robot



Thesis of: Ayomide Sebiotimo

Supervisor: Dr Konstantinos Sirlantzis

Date of Submission (08/04/2016)

University of Kent | The UK's European university

Declaration

I certify that I have read and understood the entry in the School Student Handbook on Plagiarism and Duplication of Material, and that all material in this assignment is my own work, except where I have indicated appropriate references.

Signed: _____A.Sebiotimo _____

Date: _____08/April/2016_____

Summary

A self-balancing robot utilizes two Lego NXT motors to maintain a balanced position. The theory of a self-balancing robot is based on the Inverted Pendulum principle, it is a principle that focuses on unstable systems. The inverted pendulum is utilized by analysing its mathematical model to design and implement a self-balancing robot with an Arduino microcontroller. The project aims to produce a stable, responsive self-balancing robot. The implementation of the robot consists of a 3-axis accelerometer and a 3-axis gyroscope mounted in a breakout board (MPU-6050 GY-521). A fusion algorithm was implemented to filter both the accelerometer and gyroscope data to a reliable and accurate usable data output. Therefore, two methods were tested, Complementary filter and a motion fusion algorithm integrated on the MPU6050; Digital Motion Processor (DMP). The two-wheeled robot chassis consisted of three aluminium plates that were used to hold the various components of the robot. The balancing robot is powered through a high-current (220mAh) Lithium Polymer (LIPO) battery as well as a motor driver (L298N) consisting of an H-bridge circuit. A control loop called PID (Proportional, Integral, and Derivative) was implemented to correct the error off the set-point of the robot. The PID implemented provided the ability to control speed and direction of the motors through Pulse Width Modulation. Consequently, the results achieved provided a robust system enabling the robot to stay self-balanced at a still position for approximately 5+ minutes; the timing depended on the testing environment.

Acknowledgements

I would like to express my gratitude to all the colleagues who assisted me with the development of this project. First, I would like to thank my supervisor Dr Konstantinos Sirlantzis for giving me the opportunity to work on this project and for his insight knowledge on this project. Secondly, I would like to thank Mr Harvey Twyman, Mr Tony Brazier and Mr Simon Jakes for helping me with the mechanics of the project.

Table of Contents

7. Appendix A

1. Introduction

1.1 Background

What is the definition of a ‘robot’?

A robot is a “reprogrammable, multifunctional manipulator designed to move materials, parts, tools or specialized devices through variable programmed motions for the performance of variety of tasks” [1].

Robotics is an essential part of the human society. The ultimate goal is to create a machine that is able to replicate the human ability to think and act on its own. This development continues all through humanity. Over the past ten years, developments in technology have tested many methods to replicate the human ability; these methods include developments of better functioning microprocessors that perform better processing and computations; better sensors have been developed to provide machines with better accurate information of the environment; these newly developed and improved systems motivates engineers to develop better robust systems. The different types of systems developed recently include, manipulator robots, legged robots, wheeled robots, autonomous underwater vehicle robots, aerial robots, etc.

1.2 Literature Review

1.2.1 Introduction

The idea of mobile robots has gained propulsion over the last decade in a number of different industrial settings around the world; manufacturing industries, work places i.e. hospitals, schools and the ordinary home. This is due to the unstable dynamics of the system operation, a system that defies the logic of the human brain. It is an engineering problem based on control theory of inverted pendulum and is much like trying to balance a stick on the tip of a finger.

This problem is not uncommon in the field of engineering and it has been widely used around various technologies. The inverted pendulum has long been the interest of young engineers and robotic enthusiasts; the uniqueness of this system has drawn attention to researchers around the world. In the study of engineering cybernetics an established field; which deals with the question of control engineering and its behaviour on dynamical systems. An example based on the use of engineering cybernetics is the Sojourner robot, a robot that was designed to manoeuvre on the surface of Planet Mars [2].

1.2.2 Inverted Pendulum

Recent developments of newly mobile robots have expanded since the past decade; out of the many developments, one that stands out is the “self-balancing” robot. Self-balancing evolves from a principle namely described as ‘an inverted pendulum’. To develop a reliable and capable control system for the self-balancing robot, an understanding of the parameters within the system is essential. Representation of the system can be achieved through a mathematical model. Inverted pendulum theory is traditionally known as Pole and Cart theory *fig1* [3], the self-balancing robot does not completely replicate the Pole and cart theory but it borrows its principles. In this case, the wheels of the robot replicate the cart while the chassis of the robot replicates the pole. The ‘pendulum’ is weight suspended from a pivot point so that it can swing freely [3]. On the other hand, inverted pendulum has its centre of mass above its pivot point enabling the pendulum to be naturally unstable [3]. The following principle is applied to the design approach of the balancing robot in *fig4*. In this particular case, the wheels represents the pivot point as well as placing the microcontroller, motor driver, and the LIPO battery at the top layers of the chassis forces the centre of mass above the wheels. The gyroscope is allocated under the top layer for better angle readings.

Figure 1 Pole and Cart Implementation

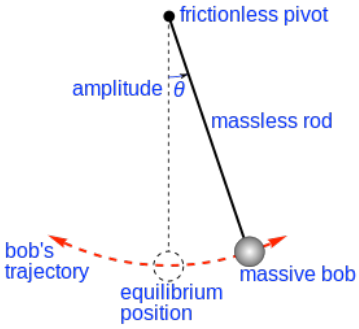


Figure 2 Pendulum (Pendulum [3])

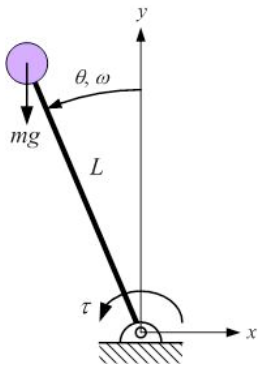


Figure 3 Inverted Pendulum (Inverted Pendulum Control System [4])

1.2.3 Inverted Pendulum Mathematical representation

Figure 4 displays the mathematical model of a typical balancing robot, it consists of the two points mH and mL connected to each other, having the distance L as the length of the robot. The lower point mL represents the motor/wheels drive and the upper point mass mH represents the rest of the body frame (centre of mass above mL). The $'s'$ is the moved distance of the lower point mL and the angle theta.

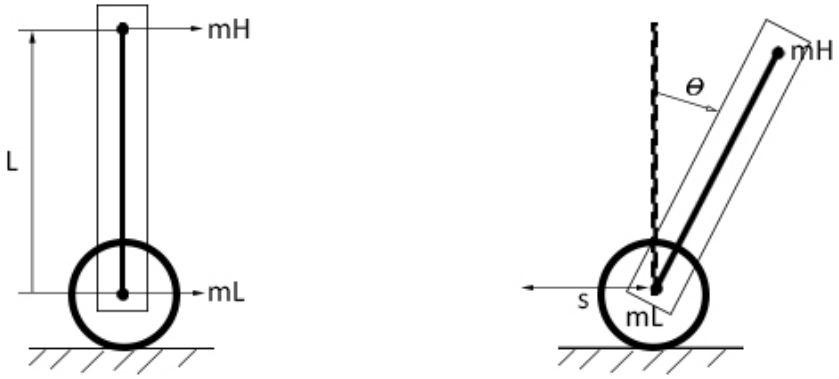


Figure 4 Mathematical model of an Inverted pendulum /Self-balancing robot

1.2.4 Autonomous Stability

Autonomous stability is the understanding of an unmanned vehicle to accomplish a balanced decision on its own. These decision either depend on its own ability to make decisions or either through a method of pre-programmed commands i.e. PID. The aim of this project is to create a robot that balances on its own without human interaction.

Most two-wheeled robot consists of several sectioned layers (at most 2 to 3 layers) where various components are installed and integrated. Components of the robot may include microcontrollers, sensors, wheels, power source, etc. The microcontroller acts as the central processing unit consisting of some type logic algorithm. Sensors such as gyroscope are required to provide information of the robot's position and its surroundings. The wheels are located at the base of the chassis to provide motion (usually dc motors or servo motors).

1.2.5 Existing two-wheeled robots

One example of a balancing robot is the Segway *fig4*; the Segway is a well-known commercial product; a two-wheeled vehicle that has seen deployment for leisure, functional disabilities, etc.

Figure 4 Segway [4]

The demand for Segway has seen a steady growth in society as more people are realising that the Segway can be useful in everyday life activities; especially use of transportation.

In a pilot study of alternative mobility for disabled people, it was concluded that the Segway is useful device for a broad range of populations with functional disabilities and also reduces fatigue, to avoid hazardous conditions, dangerous environments and assistance and monitoring for people with medical conditions, [5] etc. In reference to ethical issues, devices such as Segway has enlightened disabled populations to better opportunities i.e. aging populations. Industrial environments has also seen the deployment of Segway, used for manoeuvre and distribution.

Besides the development of the Segway, studies of prototype two-wheeled balancing robots have been widely reported. For example, nBot, Balanduino, hands free Segway etc. The nBot *fig5* is an early-implemented version of the inverted pendulum principle. It consists of inertia sensors, motor encoders and on-vehicle microcontrollers. The nBot was featured on NASA's Cool Robot of the Week [6].

Figure 5 nBot Balancing Robot [6]

Since the development of the nBot, there has been active research on commercial systems. For example, Balanduino *fig8* [7] is an Arduino based self-balancing robot that is commercially available for users to operate and add extra features. The Balanduino can be controlled with a vast majority of devices such as game controllers, mobile phones, etc. The Balanduino consists of a “single main board” that acts as a microcontroller. The main board utilizes the integration of the Arduino programming environment. The Balanduino balances using a combination of 3-axis accelerometer and a 3-axis gyroscope. In terms of power, the Balanduino uses a 4200mAh 11.1v 3-cell LIPO battery. The Balanduino also consists of a digital H-bridge motor controller; model L298N [5].

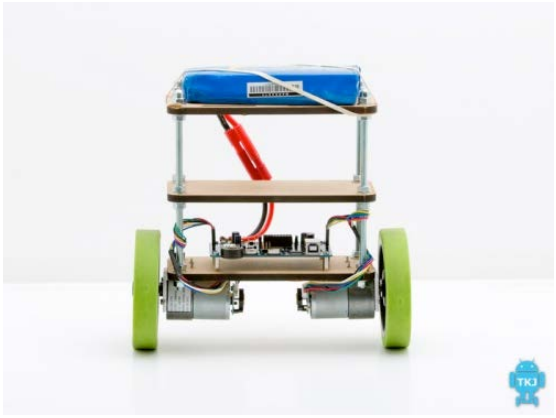


Figure 6 Balanduino Self-balancing Robot

1.3 Aim of the project

1.3.1 Why balancing robots?

Recent major applications towards control theory of the inverted pendulum are based on transportation and health, for example the Segway is designed for use of transportation that has contributed much positivity towards society. Balancing Robots can be employed in many applications within society including carers, assistants and security; industrial and manufacturing environments have also been presented with these benefits. This field of research is essential as robots offer an opportunity of improving the quality of life for every member of society.

1.3.2 Aim

The aim of this project is to design a two-wheeled vehicle based on the inverted pendulum and put to test, the inverted principles. The two-wheeled vehicle is required to keep the wheels beneath the centre of the robot chassis' mass therefore achieving a balanced position. The vehicle is required to autonomously balance on its own whereby the inverted pendulum is in equilibrium state, a state where its position stands upright at ninety degrees, and also move independently from any other tethering. Ultimately, anytime the robot is played upright manually, it will always fall off due to disturbance (pushing), the aim is to design a system to prevent the robot from falling. The robot's system will be able to utilize an ATMega32 microprocessor implemented on the Arduino UNO microcontroller. A gyroscope is used to provide information about the state position of the robot, while a PID controller is required to perform control feedback to the microcontroller.

1.3.3 Objectives

The project objectives were broken into several key goals:

1. Review and evaluate literature, research on the inverted pendulum theory and two wheeled balancing robots. This provided the basis for the design approach based on previous experiences and procedures by others. Examine the necessary considerations that may affect the performance of the system.

2. To identify potential resources required with for the development of the robot especially hardware components.
3. Test finalised components against specification requirements to identify any faults.
4. To build a circuit required in order for the robot to stay balanced, to utilize the necessary components i.e. gyroscope, microcontroller, motor driver, etc.
5. Develop a logic algorithm to keep the robot balanced, consisted of testing different filtering method to filter out noisy data from the gyroscope. Also, to design a PID control algorithm using Libraries provided from previous experiences.
6. Testing, analyses of the different filtering methods and evaluate the performance of the robot under each method. Consists of improving the robustness and stability of the robot.

1.4 Risk Assessment

The risk assessment *Table 1* consists of the perceived risks associated with the work undertaken during the process of the project.

Table 1 Risk Assessment of Project

Task	**Hazard**	**Risk Control Measures**
Literature review and research tasks.	Eyestrain.	Regular breaks, use of adequate lightning and ergonomic furniture
Manufacture and assembling of robot assemblies and components	Physical injuries from tools or materials. Electrocution from power tools. Exposure to Battery. Damage to sensitive components and costly parts.	Use of personal protection equipment Adequate training in tools and material usage is undertaken.

Programming, software design, simulation and analysis	Eyestrain Fatigue	Regular breaks
Report writing and compilation	Eyestrain Fatigue	Regular breaks
Robots operation	Falling, tipping and collision	Add impact support to the robot to avoid damage to the chassis

1.5 Conclusion

The report aims to evaluate the approach taken to implement a two-wheeled balancing robot. Main topics include design implementation, simulations and test analysis. The aim of the project is to put the Inverted Pendulum principle to test. This project proves that the Inverted Pendulum is achievable.

2. System Description

2.1 Introduction

The system description of the entire project consists of the research conducted from the literature review to support the requirements and objectives of the project. This particular area consists of the hardware and software components used to form the balancing robot. The description also includes some design considerations to avoid potential faults and to provide better stable performances.

The two-wheeled robot will be broken into two distinctive sections. The first is the ‘hardware system’; this section consist descriptions of the sensor, chassis, motor driver, battery, and microcontroller. Secondly, the ‘software system’, which aims to describe the fusion algorithm i.e. complementary filter and DMP as well as the PID (Proportion, Integral, and Derivative) control system. Figure 7 shows an overview of the hardware system.

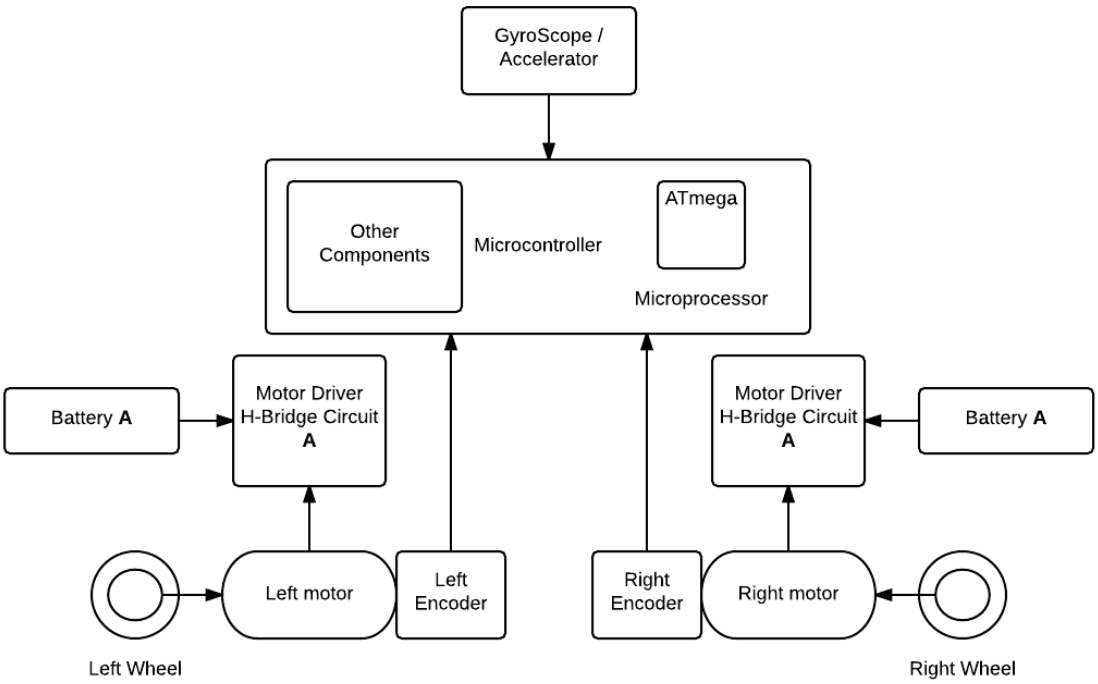


Figure 7 Block Diagram Hardware overview

2.2 Balance Process

It is important to understand the balancing process of the robot's system. The diagram *fig8* below describes the basic functionality of the balancing robot system.

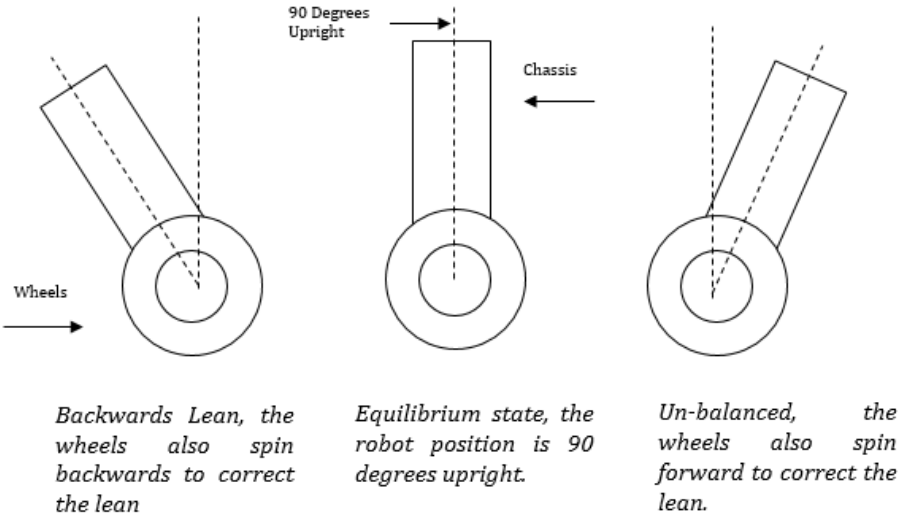


Figure 8 Balance Process

2.3 Requirements

There were several steps considered to implement the algorithm. Although the main deliverable was the control algorithm, the hardware design was also an important stage in the project.

2.3.1 Bill of materials

- Arduino UNO R3 x1
- MPU GY-521 MPU 6050 Module 3 Axis Gyro+3 Axis Accelerometer x1
- L298N H-Bridge motor drive controller x1
- Lego NXT motors x2
- Toggle Switch x1
- 12C 2004 Serial 20 x 4 LCD Module x1
- LIPO Battery
- Three Aluminium sheets
- 4 metal threaded rod
- Nuts and washers
- Breadboard
- Jump wires

2.3.2 Drive mechanism

The drive mechanism of the robot consisted of two Lego NXT servo motors powered through an L298N H-bridge circuit motor driver. Overall, the drive mechanism was acceptable for this type of robot. However, this setup was far from ideal. Ideally, a DC motor with or without encoders is preferred due to the fact that the NXT motors were particularly designed for the Lego Mindstorms development kit and has rarely been used with the Arduino.

2.3.2.1 Lego NXT motor

Lego NXT motors contain DC motor that has a geared ratio of 1:48, it also consists of an optional rotational encoder and a PCB. The encoder provides the means of measuring distance travelled and its direction, which is the used as feedback within the motor control computations. This also provides a function for deriving velocity within the microcontroller. A cable is required to drive this motor with regular 9v sources, which is connected to the microcontroller and the motor controller. The Lego motors have the following characteristics:

- Dual power supply up to 9v
- Internal incremental encoder
- Rotational speed of 170 rpm
- No-load current 60mA
- Stalled torque 50 N.cm
- Stalled current 2A
- Ratio 1:48

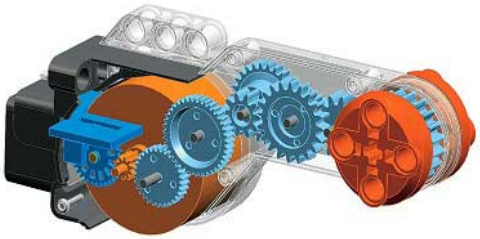


Figure 9 Lego NXT motor internals

Figure 10 shows that the maximum power is obtained at a torque load of about 15 N.cm. The NXT motor is powered with a 7.4V Lithium polymer battery.

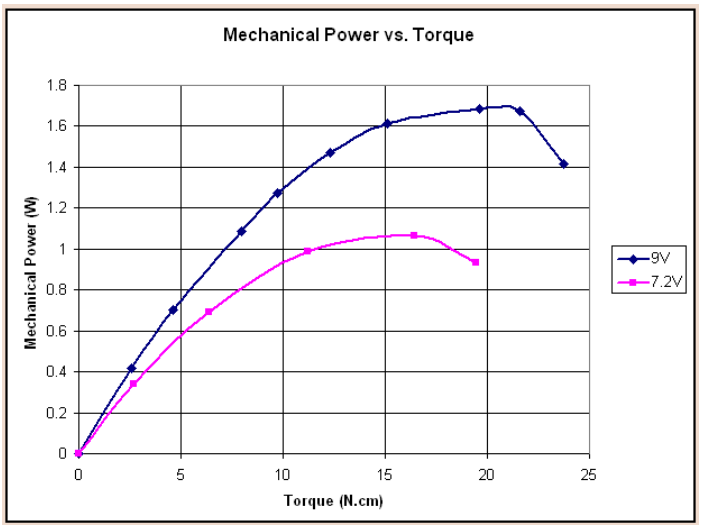


Figure 10 Chart-rpm torque

2.3.2.2 Encoder

The NXT motors uses an RJ-12 connection cable, which offers the ability to read encoder signals. The signals within the cable have the following specification. The NXT motor is powered with a 7.4V Lithium polymer battery.

- First supply signal (0 to 9v)
- Second power signal (0 to 9v)
- Ground
- Encoder Supply (5v)
- First encoder signal
- Second encoder signal

The Encoders offer an alternative way to keep the robot balanced as to using the gyroscope. The Lego NXT motors are equipped with incremental encoders, which provides cyclical outputs only when the encoder is rotated

The NXT motor encoders track how far the robot has rotated counting up to 360 degrees for a one full wheel rotation. The encoders enables the measurement of discount rather than angle. The encoders is implemented in way to make sure the desired distance/position is reached every time.

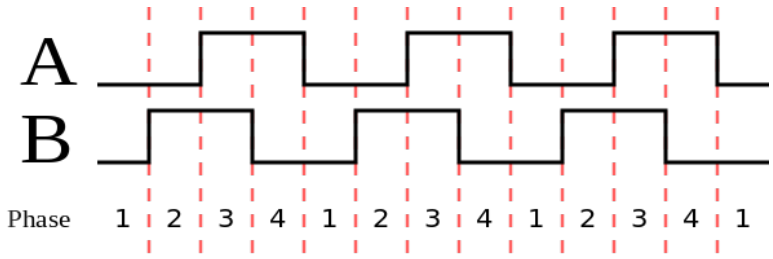


Figure 11 Incremental Encoder

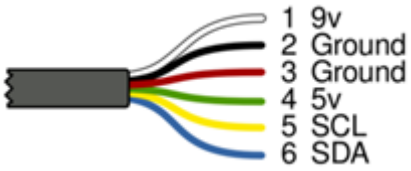


Figure 12 RJ-12

2.3.2.3 L298N H-bridge motor driver

In order for the robot to maintain a stable position, force is required to push both wheels back and forth; in this case a motor driver is an example of a device that can cause movement. The motor controller will need to work together with the Arduino controller. The function of the motor driver is to take low-current control signal and then turn it into a higher current signal powered by batteries. The motor driver is to be directly connected to the Lego NXT motors. The H-bridge controller enables voltage to be applied across a load in either direction therefore enabling the NXT motors to move clockwise or anti-clockwise.

The L298N is a dual motor controller that consists of DC and stepper motor control. The L298N is based on the H-bridge IC, which allows control of speed and direction of the NXT motors. The L298N can be used with motors that have a voltage of between 5V and 35V DC.

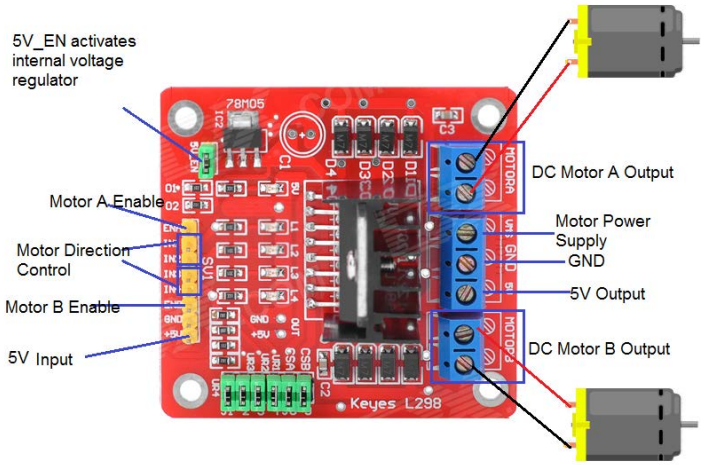


Figure 13 L298n Motor drive controller

2.3.3 Sensor MPU6050 GY-521

Gyro sensors are devices that sense angular velocity, a sensor that is able to detect change in rotational angle and freely predicate any orientation. The gyroscope is be able to provide the robot with the ability to interpret the environment; as without this, the robot would blindly execute series of instructions without the capacity to re-evaluate its position and make adjustments.

The MPU6050 consists of 3-axis Gyroscope (offers degree of freedom) and 3-axis Accelerometer. The MPU6050 is powered by 3.3V power supply and communicates via 12c protocol. Accelerometers and Gyroscopes are the most common types of sensors utilized in robots and machines that require stability control as they provide the means of measuring acceleration, velocity and direction.

2.3.3.1 Accelerometer

Accelerometers provide a means of measuring acceleration along an axis, also known as force. They are susceptible to position noise, which can be avoided by incorporating filtering on their output lines. The main purpose of an accelerometer is to measure changes in speed. It is classed an analogue device as the voltage varies with respect to the rate of change. This device will be used to determine the change with relation to the robot's tilt, its change in acceleration. The accelerometer output

will be combined with the gyroscope output to compensate for gyroscope drift thus creating a better stability control signal. The accelerometer will only use two axis, one axis to detect upward/downward motion whilst the other will detect forward/backward motion.

2.3.3.2 Gyroscope

A gyroscope presents the means for measuring rate of rotation (angular rate) but a common problem is that the output drifts overtime. A more reliable and accurate reading can be achieved by fusing both a gyroscope and inclinometer together. This uses the inclinometer to detect the rotation and correct the output signal from the gyroscope.

The gyroscope sensor will be used to determine the angle of tilt of the robot chassis. The gyroscope output will be combined with the accelerometer output at set intervals to compensate for drift thus creating a reliable and accurate signal. The MPU6050 is mounted under the top layer of the robot's chassis.

2.3.3.3 Sensor Fusion

The MPU6050 consists of an in-built fusion algorithm DMP (Digital motion process). Sensor fusion is required to combine both accelerometer and the gyroscope data to form a single value. The fusion algorithm uses multiple responses from the accelerator and gyroscope and converts both data into a more reliable output signal. This fusion algorithm is beneficial because majority of sensors have a degree of unreliability that becomes inherent over time as well as noise.

Figure 14 MPU6050 GY-521

2.3.4 Microcontroller Arduino UNO

The Arduino UNO is a type of microcontroller that runs on a microprocessor called ATmega32, which is the main processing unit of the robot. The ATMega (Atmel) microprocessor is able to run a set of instruction given in form of a programming language; it is also the centre of communication between hardware and software. The number '32' signifies that the processor can process data in 32-bit connected to data buses. The ATMega pins can be used as input or output; input in the form of receiving signals from the gyroscope and output in the form of giving out voltage 5v towards the motor drivers.

2.3.5 LCD Screen

The LCD screen is used to manually control the PID logic algorithm. The PID values are controlled via 10k ohm potentiometers, which are displayed on the LCD screen.

2.3.6 Power Supply Unit, LIPO Battery 2.2A, 7.4V, toggle switch

The LIPO battery is used to power up the robot system. The power supply uses a toggle switch, which powers up not only the Arduino but also the motor driver.

2.3.7 Chassis

The chassis was built of three layers of thin and strong aluminium plates. This made the robot very light. Four holes were drilled into each of the layer to allow the threaded rod to pass through. Additional holes were required to mount the Arduino, Lego NXT motors, motor driver, toggle switch and the MPU6050.

Figure 15 Initial Chassis Design

Figure 16 Base plate with NXT motors

2.3.8 Financial Summary

Table 2 Financial Summary

Description	**Quantity**
Arduino Uno	1
L298N Motor Driver	1
MPU6050 GY-521	1
LIPO battery	1
Lego NXT motor	2 (FREE)
LCD	1
Threaded rods	4 (FREE)
Washers, nuts and bolts	-
Jump wires	-
TOTAL	£23.96

2.3.9 Conclusion

Overall, the hardware components will be implemented together to form the circuit to achieve the functionalities in order for the robot to fulfil a balanced position.

3 System Implementation, Approach and deliverables

3.1 Deliverables

The following are the deliverables of the system implementation.

- Chassis Design
- Design Software Flowchart
- Circuit Design
- Calibrate gyroscope
- Get raw data from the gyroscope
- Implement fusion algorithm to filter raw data
- Control motors and read encoder
- Achieve PID control
- Testing

3.2 Software Flow Chart

The programming was completed in C programming language using the Arduino development platform. The Arduino microcontroller is required to read the gyroscope whereas, based on the inputs provided, the microcontroller is then able to act, provide computations and make judgements on the required behaviour to drive the motors leading to a stable and balanced position.

Figure 19 illustrates the basic flow of the how the software algorithm is proposed to run. The main program begins with an ‘Initialise’ block which consists of manually switching the toggle switch to power up the robot, this process is executed either upon start-up or via the reset button on the Arduino. By doing this, the software program activates the MPU6050 sets its offsets. The offset is achieved through calibration to be discussed in the subsequent paragraphs. The next sub-process is reading the gyroscope, which consist of utilizing an open source library particularly made for the MPU6050. The library is then defined in the main program and with

the use of some function in the main program the angles of the gyroscope are achieved. For example, ‘getValues ()’.

The second sub-process consists of the fusion algorithm. This block uses a fusion algorithm of filtering the accelerator and gyroscope data together to form a single value. In this particular case, the MPU6050 attempts to utilise two different fusion algorithm; Complementary and the use of Digital Motion Processing (DMP) provided by InvenSense. Depending on the outcome of the filtering, the best output will be used throughout the project. The output from the filter is then passed on as a single value to the microcontroller/Arduino.

The next process is to utilize the values achieved from the previous step, based on the angle provided from the gyroscope, a PID control is initialised to control the motor based on the judgement of the gyroscope. The PID is able to control and send signals using PWM pulse width modulation to control the motor speed required to keep the robot in a balanced position. The motor controller will receive the signals via PWM pins from the microcontroller, it will then be able to calculate the amount of power needed to move the NXT motors. As the motors come with encoder, the encoders can also be used as inputs to detect the direction of the motors.

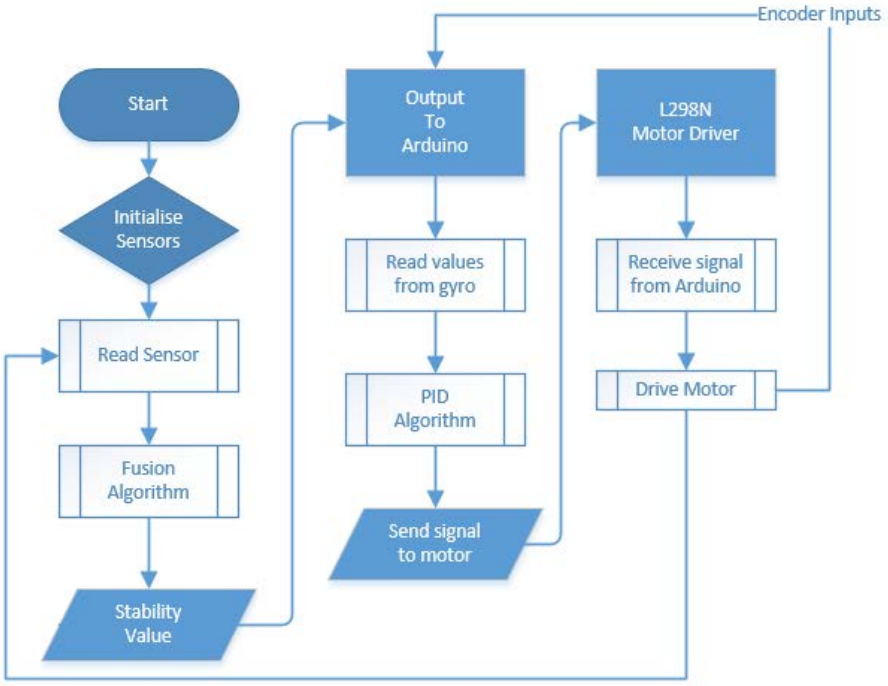


Figure 17 Software Flowchart

3.3 Initial Circuit Design 1

Components form the building blocks of circuits, which in turn provide the capability, and reality of a machines operation. Components include breakout boards, switches, sensors and IC's. They are typically assembled together to form a robust, compact and localised circuit. Components are given tolerances in which they may vary from the prescribed rated value. They are also rated on three power carrying capability and temperature variation characteristics. The aim was to build a circuit that will maintain tolerance over significant time and operational circumstances.

Figure 18 shows the initial circuit design proposed early in the development stage of the project. The circuit consists of the RJ-12 cables connected to the NXT motor. The NXT cables are then linked to the L298N motor driver directly connected to the Arduino. Although the final circuit is powered via a LIPO battery; figure 18 does not illustrate this. Initially, there were speculations that the robot could achieve a balanced position using only the encoders on the NXT motors and without the gyroscope. This experiment proved to be difficult to achieve without the gyroscope. So therefore a further improvement on the circuit commenced.

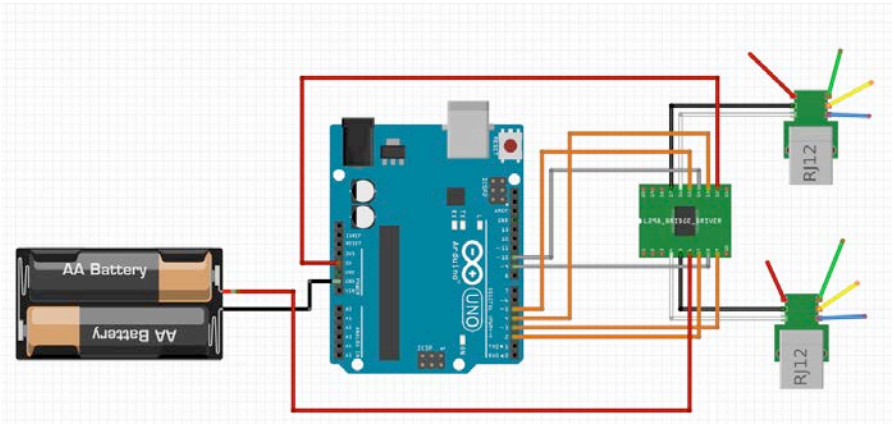


Figure 18 Initial Circuit Design

3.4 Initial Circuit design 2

The initial circuit as shown in figure 19 represents a visual

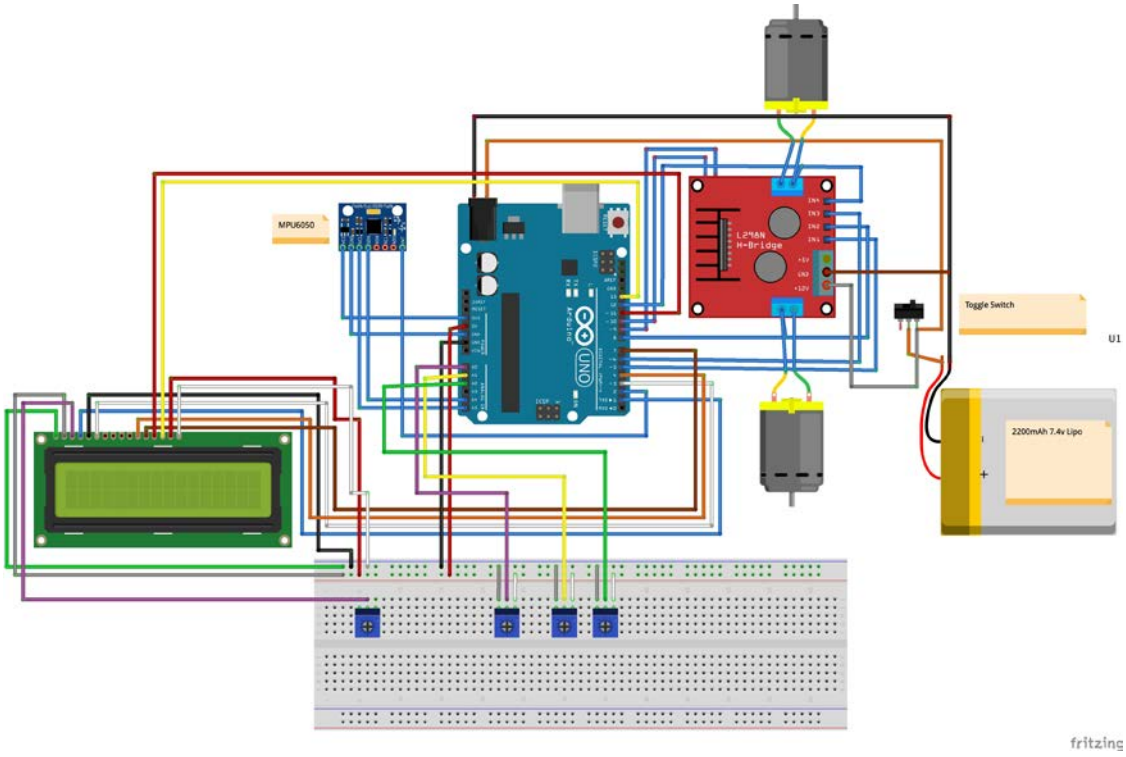


Figure 19 Initial Design 2

Implementation of the physical robot system. The sketch of the circuit was designed using a software application Fritzing. Using this application I was able to visually implement the parts and modify the circuit. As shown in the figure 21, this circuit includes the following parts, LCD, which is connected directly to the Arduino and the breadboard required to control the PID parameters. The MPU6050 gyroscope is another part that directly connects to the Arduino, which is required to feedback angle rotation data to the Arduino and powered via the 3.3volts pin on the Arduino. The motor driver L298N utilizes the digital pins on the Arduino and is powered by the 7.4V LIPO battery. The circuit also consists of a toggle switch which is able to control the power of both the Arduino microcontroller and the (L298N) motor driver.

3.5 Chassis Design

The balancing robot chassis is shaped into a rectangular shape sectored into three different layers where various components are installed and integrated. Located on each layer are the unique components of the robot's system. The middle layer consists of an Arduino and the L298N. The top layer consists of the LIPO battery and a breadboard required for the encoders. A combination of support pads and cable tie were used to support the motor bracket. Nuts and washers were utilized to hold tightly the chassis in order to withstand collision. The following images are the schematics of the three layers required for the chassis. The Lego NXT motor Level consists of the Lego NXT motor, the second level (middle) consists of the Arduino as well as the motor driver and the power toggle switch. Each of the layers were approximately 9inches by 4inches. The required drilled holes were approximately 4 millimetres.

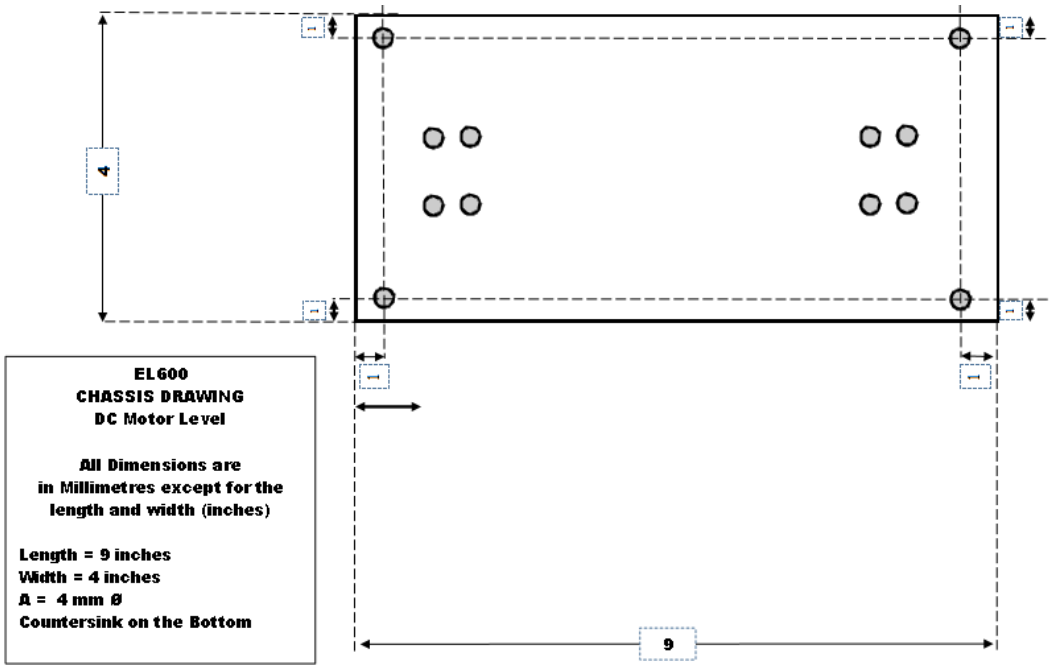


Figure 20 Lego NXT Motor Level

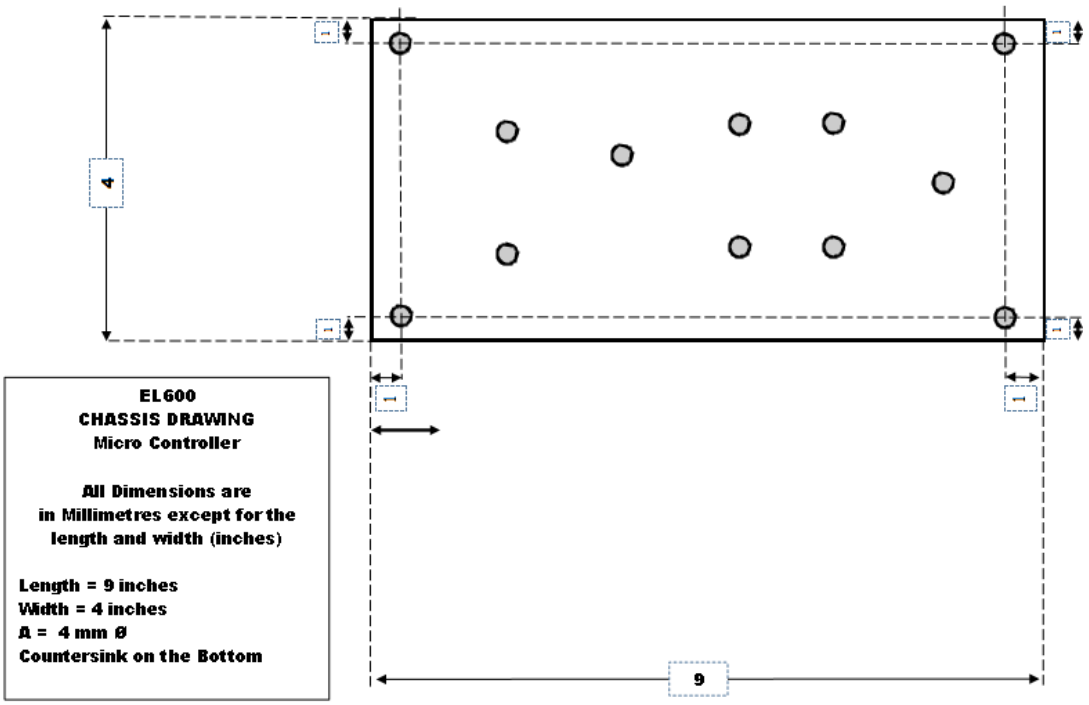


Figure 21 Middle layer (Arduino and motor driver L298N

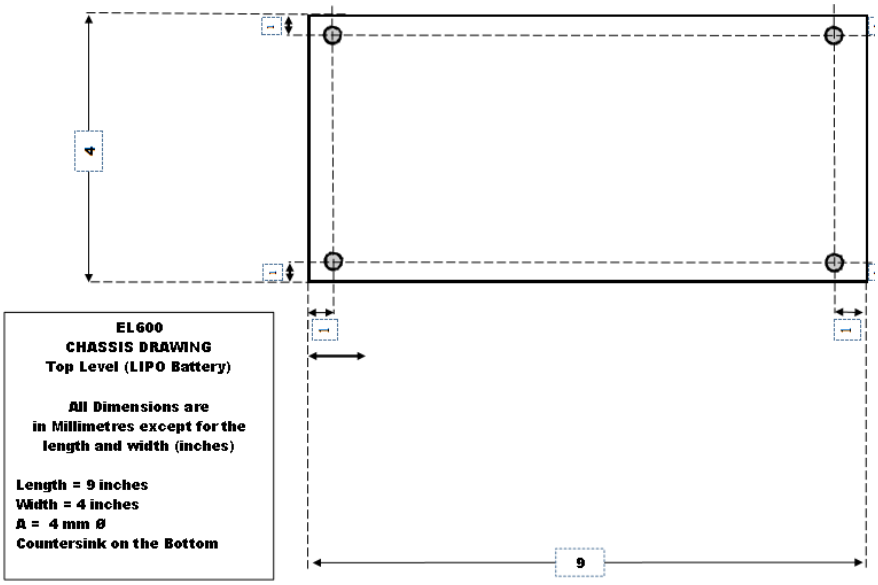


Figure 21 Top Level (Consisting of LIPO Battery and Breadboard)

The top level of the robot only required four holes to be drilled as the LIPO battery and the breadboard were mounted onto the aluminium plate with Velcro. Both components were placed on the top layer of the chassis substituted for weight to maintain stability so therefore more torque can be generated. This principle can be equated to the inverted pendulum as well as trying to balance a stick at the tip of a finger, which has proved difficult to maintain in a vertical position.

3.6 Implementing Sensor

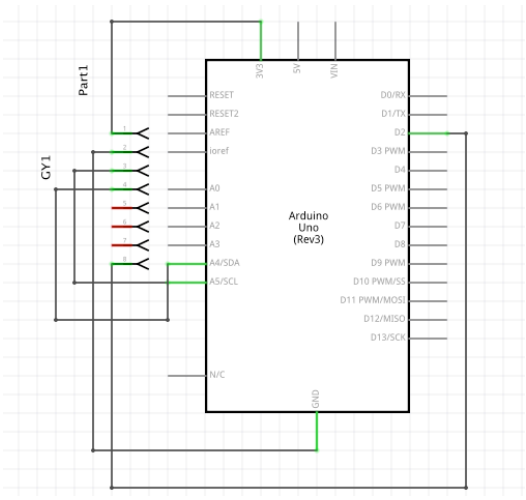


Figure 22 MPU to Arduino Schematic

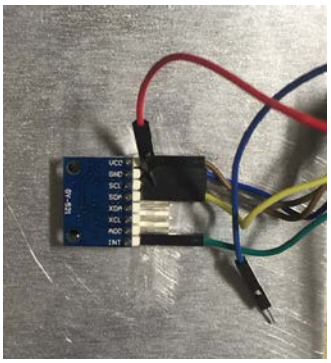


Figure 23 MPU6050 mounted under the top layer

In order get an understanding of the functionalities provided by the MPU6050, a number of Arduino sketch examples were put to test. For example, familiarisation of YAW, PITCH and RAW, Calibration etc.

3.7 Calculating YPR (yaw, pitch and raw)

To prevent tipping over, the centre of gravity is always changing and so to comply with the changes, the wheels (the base line) needs quickly get directly under the new position of the centre of gravity. This process is achieved by measuring the deviation from the 'upright' position to bring the wheels directly under the position.

The MPU6050 provides acceleration measured in G forces, in three axis and angular velocity (rate of rotation) in deg/sec in three axis.

Yaw is the rotation about the Z axis, pitch is the rotation about the X axis and roll is the rotation about the Y axis. As the robot drifts back and forth, the yaw value is not needed. So therefore, the formula to calculate pitch and row is shown below:

$$Roll = atan2(Y, Z) \times \frac{180}{\pi} \qquad (1)$$

$$Pitch = atan2(X, \sqrt{(Y \times Y + Z \times Z)}) \times \frac{180}{\pi} \qquad (2)$$

The teapot demo provided by Jeff Jrowberg libraries were able to visualise the values live. The values were then initialised into a function in the main code '*mpu.dmpGetYawPitchRoll()*' in the main code.

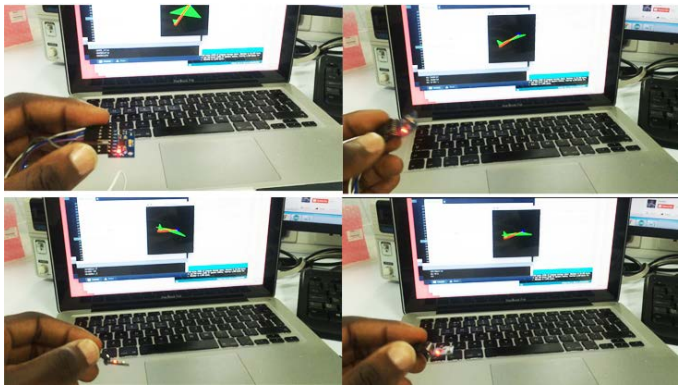


Figure 24 Tea Pot Demo

The MPU6050 provides the robot the ability to decipher the surrounding as without this ability, the robot would aimlessly execute a sequence of commands without the scope to re-evaluate its position and make adjustments. To achieve data from the sensor, calibration was acquired to achieve the offset position of the robot. From calibration, the raw data of the outputs were achieved using an Arduino sketch provided by '12Cdevlibs' [14], the next procedure after achieving the raw data outputs was to filter the data. As discussed, the MPU6050 consists of a Digital Motion Processor and by using the DMP; the filtered data outputs were achieved. Based on the outputs, the MPU6050 provided a feedback response back to the Arduino which ran the PID control algorithm. The control algorithm will be able to run forever and attempt to fulfil a balanced position until the power toggle switch is off. In order to implement the sensor, the following procedures were put to tasks:

- Getting the sensor values
- Calibration
- Apply the fusion filter algorithm
- Calculate Yaw, Pitch and Raw

3.8 Getting the sensor values

It is possible to use the gyroscope without using any filter algorithm, it is easier to code, the gyro gives fast angular velocity measurement; although, the data output is still very noisy. As discussed earlier on, the MPU6050 consists of a Digital Motion Processor (DMP), the DMP uses a FIFO buffer to store data that arrives to the MPU6050 chip asynchronously. The storage structure of a FIFO buffer, which is an array of adjoining memory is based on a first-in, first-out basis. The structure of the storage.

Sensor fusion is the means to merge or fuse data from the gyroscope and accelerometer sensors to achieve a reliable and less noisy value in order for the robot to fulfil a balanced position without drifting. Figure 22 shows the serial monitor performing raw data outputs while reading the gyroscope, sketch provided by 'I2Cdevlibs'. As shown in the image, the output records a lot of jitter and noise, which needs to be filtered by the microcontroller before the data can be used. To filter the raw data, a fusion algorithm is required to achieve better accurate and stable values. The readings from the serial monitors are divided into accelerometer (ax, ay, az) and gyroscope (gx, gy, gz). It is possible to calculate the tilt of the sensor manually through the use of a number of formulas; for example, Kalman filter is an adaptive filter that can be used to filter sensor data from the accelerometer and gyroscope. Although due to the difficulty and time of implementing the Kalman filter, this method was unused. The initial filter implementation of the robot utilized the complementary filter (results to be discussed in the following paragraph). The final implementation of the project uses the DMP fusion algorithm.

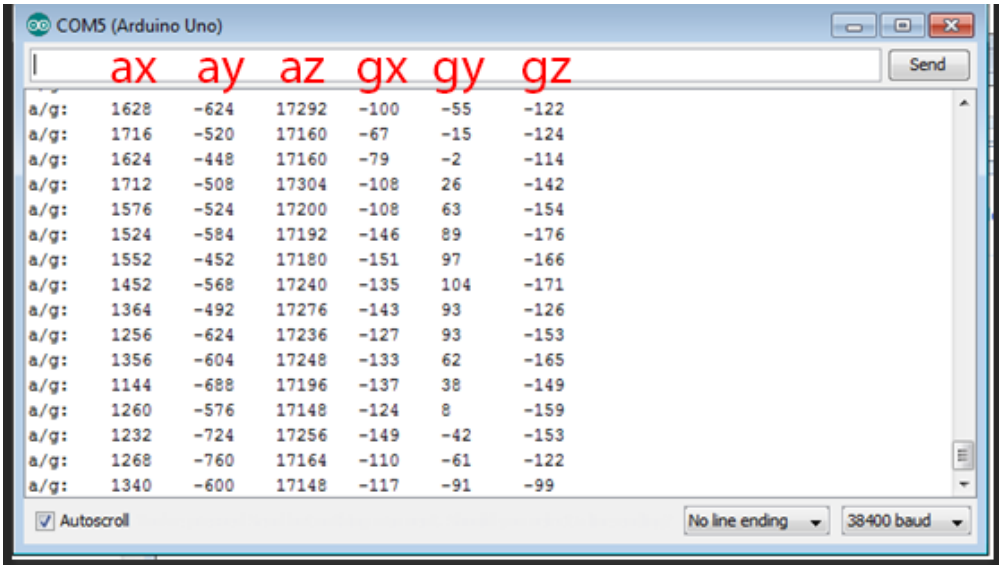


Figure 25 MPU6050 Raw Values Results

The final Arduino sketch consists of a function namely ‘getValues()’. This function aims to access the data from the DMP by reading packets from the FIFO buffer on the mpu6050 chip. This is achieved by assimilating and tracking FIFO counts whereby if the count is greater than 1, it clarifies that a packet is available to read and acquire data outputs such as the YAW, Pitch and Roll.

3.9 Gyroscope Calibration

The next implementation was in regards to MPU6050 calibration. The MPU6050 communicated to the Arduino via I2C bus, which uses two special pins, SDA and SCL. The MPU6050 is powered by connecting the 3.3v; GND (ground) to the Arduino as well as an interrupt pin (pin2) was connected to the Arduino. To kick start the calibration, an open source library for the MPU6050 was utilized. The library offered an I2C scanner example, which scans the 12C-bus for devices. This sketch proved that the 12C communication was successful and able to communicate (I2C device found at address *0x68*). The sketch for calibration was provided by 12C library, which was able to calculate the offsets required for the robot to balance. The

procedure of the calibration consisted of manually placing the robot in the assumed stable position while the calibration was on going.

Table 3 Accelerometer Offsets

AcceX	**AcceY**	**AcceZ**
-2702	-1235	1154

Table 4 Gyroscope Offsets

GiroX	**GiroY**	**GiroZ**
30	-9	31

3.10 Complementary Filter

The initial filter algorithm implemented onto the robot was the Complementary filter; which is an easier fusion algorithm to achieve less noisy sensor data. The complementary filter theory was easier to understand, it was able to fix the noise. The equation of a typical complementary filter is as follow:

$$d_{ang} = (hf_{tc}) \times \left(d_{ang} + x_{gyro} \times dt\right) + (lf_{tc}) \times (x_{acc}) \qquad (3)$$

d_{ang} - The desired angle of the gyroscope

x_{gyro} - The raw value of the gyro (gy, gx, gz)

$\left(d_{ang} + x_{gyro} \times dt\right)$ - Integration

x_{acc} - The raw value of the accelerometer

hf_{tc} - High-pass filter coefficient

lf_{tc} - Low-pass filter coefficient

dt - Sample Rate

f_c - Cut-off frequency

TC - Time Constant

The complementary filter is a sum of a high-pass filter and a low pass filter. The purpose of the high-pass filter is to pass signals to frequencies above the cut-off frequency f_c and reduce the force to a point lower than the cut-off frequency. The high-pass portion of the equation is:

$$(hf_{tc}) \times (d_{ang} + x_{gyro} \times dt) \qquad (4)$$

The product of a coefficient hf_{tc} and $(d_{ang} + x_{gyro} \times dt)$ is the time constant TC; contrarily reciprocal to f_c. The-low pass portion of the equation is $(lf_{tc}) \times (x_{acc})$, which performs the inverse of a high pass filter by passing signals to frequencies lower the cut-off frequency and reduce the force to a point higher than the cut-off frequency. The aim is to pick a time constant that would define the boundary between the gyroscope and the accelerometer. For example, a time constant of 0.28 is achieved as follow:

$$TC = \frac{hf_{tc} \times dt}{1 - a} = \frac{0.96 \times 0.01}{0.034} = 0.28\,\text{sec} \qquad (5)$$

If the period is less than half a second, the gyroscope is considered more important and performs its integration then followed by the accelerometer. Otherwise, for period more than half a second, the accelerometer is given more load than the gyroscope leading to drift occurring.

Selecting the coefficients required is achieved by:

$$hf_{tc} = \frac{TC}{TC + dt} = \frac{0.28}{0.28 + 0.01} = 0.96 \qquad (6)$$

This is required so that the gyroscope is filtered out first allowing more time to filter the accelerometer accordingly. Otherwise, if the loop time exceeded more than half a second, the accelerometer data is given more weighting than the gyroscope, this may result to unwanted drifting. For example, to execute the complementary filter

100 times per seconds, the time constant for both the low pass and the high-pass would be:

$$r = \frac{hf_{tc} \times dt}{1 - hf_{tc}} = \frac{0.98 \times 0.01\ sec}{0.02} = 0.49\,\text{sec} \qquad (6)$$

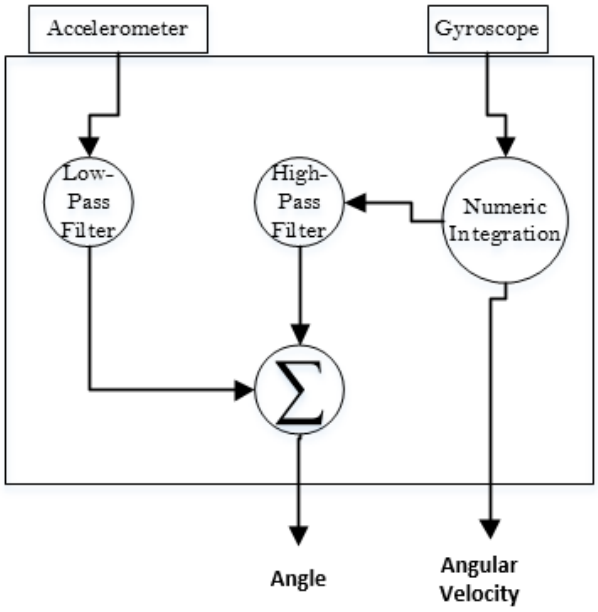


Figure 26 Complementary Filter

3.11 Digital Motion Processor

As described earlier on the MPU6050 incorporates not only a 3-axis gyroscope and a 3-axis accelerometer but also comes with an on-board Digital Motion Processor that is able to process motion fusion algorithms. The DMP supposedly acquires data from the accelerometer, gyroscope and additional 3rd party sensors such as magnetometers. The filtered data is then stored in a FIFO buffer or can be read from the DMP's register. In order for the DMP to function with its maximum ability towards performing its motion fusion algorithm, it has to offload timing requirements and processing the host processor, in this case, the Arduino. The DMP is assumed to run at a high rate of approximately 200Hz in order to create accurate results with low latency which is convenient with the Arduino. Overall, the DMP can be utilized in a way to minimize power, simplify timing, and software architecture. Although the

fusion algorithm utilized by the DMP is not provided by INVENSENSE Motion, it offers an alternate method for frequent polling of the MPU6050 [8].

3.12 Complementary Filter vs. DMP

As discussed, the fusion algorithm for the DMP hasn't been published by INVENSENSE, a comparison of the filtered data outputs of the accelerometer and gyroscope from the MPU6050 using complementary filter in comparison to the digital motion processor was approved. Using an Arduino sketch provided by 'Geekmomprojects' [9] together with another IDE, Processing. The Arduino sketch was able to send the data over the serial port to Processing sketch to display the comparison graphically. Below is a screenshot from the Processing graph. As shown, the calculations for pitch and roll of the MPU605 were fairly close; the DMP was able to calculate Yaw, which the complementary couldn't and was concluded that the DMP was more accurate.

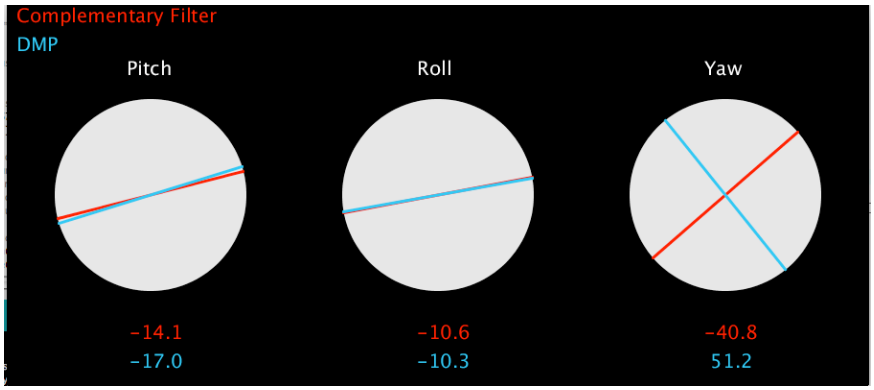


Figure 27 Complementary Filter vs DMP

3.13 Utilizing DMP filter data outputs

Figure 28 shows the initial setup to achieve the filtered data output from the DMP, the DMP demo is initiated by sending any character to the serial monitor.

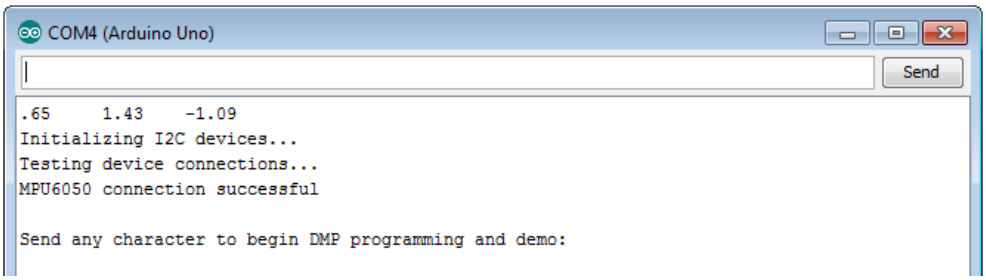


Figure 28 DMP initialisation

Figure … displays the YAW, PITCH and RAW data from the, in comparison to figure 25 the filtered data outputs are more reliable and accurate.

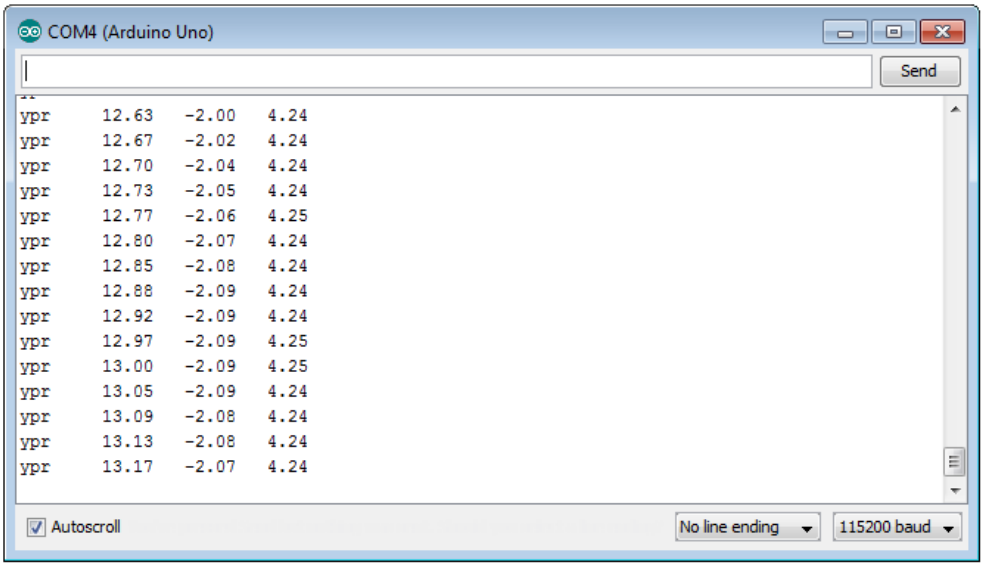
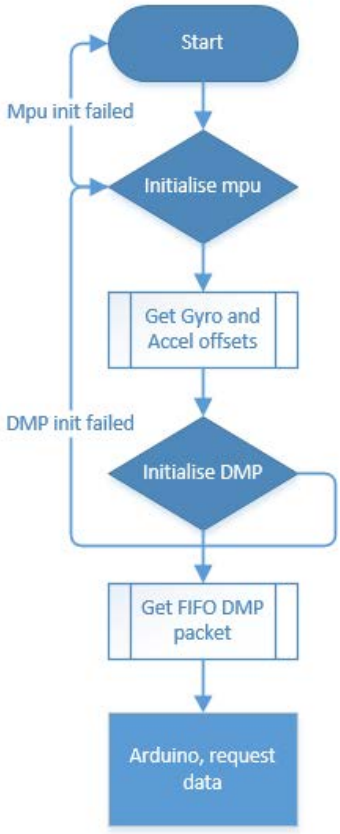
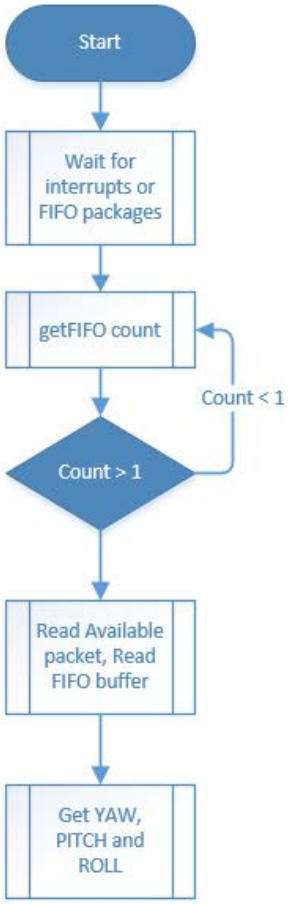


Figure 28 DMP filtered Yaw, Pitch and Roll



Flow Chart 1



Flow Chart 2

Flow chart 1 displays the flow of how the gyroscope is initialised. The first initialisation occurs by activating the mpu6050 itself as the DMP is one of the parts that forms the MPU6050. If the initialisation fails, the MPU is re-initiated, this would mean running a check on the pins connected to the Arduino or running a check on the power supply. The aim of the sub process after the initialisation of the MPU6050 is to gather the gyroscope and accelerometer offsets. By determining the offsets, the DMP is then initialised and able to produce accurate data outputs. As discussed earlier, data is collected from the FIFO buffer of the DMP, the next process is to read the packets available from the DMP and send it to another function in the Arduino sketch (Flow chart 2).

Flow chart 2 displays the flow of how the Arduino supposedly receives the data from the gyroscope. The final Arduino sketch consists of a ‘getValue()’ function which is intends to run a query to get the values from the gyroscope. The function waits for interrupts or FIFO packages to arrive this is verified by the number of FIFO counts.

For FIFO count greater than one signifies that there are available packets to read, otherwise, the function aims to continually request FIFO packages. The next sub process after acquiring the FIFO count is to read the available packet by accessing the FIFO buffer. From accessing the FIFO buffer, the YAW, PITCH and ROLL data are accessed as well as other data such as Quaternion and Gravity.

```
fifoCount -= packetSize;
        mpu.dmpGetQuaternion(&q, fifoBuffer);
        mpu.dmpGetGravity(&gravity, &q);
        mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
    }
```

3.14 Implementing PID Control

The PID controller is the most common form of feedback. In process control today, more than 95% of the control loops are of PID type, most loops are actually PI control [10]. PD control was initially implemented, however the control was unsuccessful as the motors produced a slow reaction towards tilt. Favourably, the Arduino platform provides an Arduino sketch to access PID functions required for any Arduino based projects. A PID controller aims to calculate an error value as the difference between a measured input and a desired set point. The controller attempts to minimize the error by adjusting an output. The PID parameters are as follow:

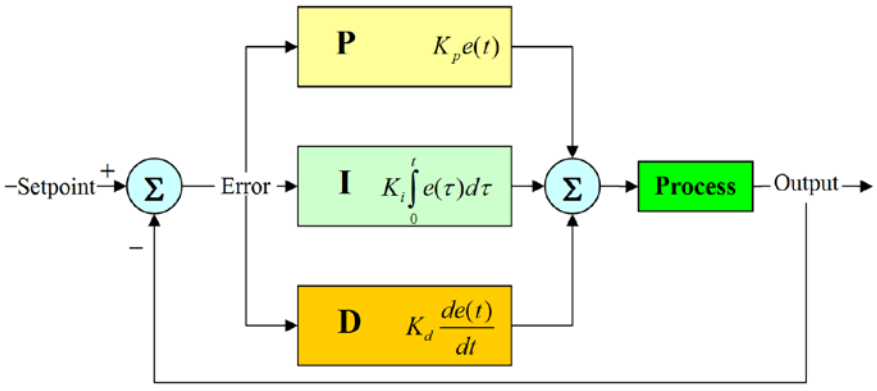


Figure 29 Industrial .NET -PID Controllers [11]

The Setpoint is the value/position (usualyy a value = 0) that we want the NXT motor to remain, the output must be equal to the setpoint, otherwise this returns an error signal.

Proportional – The P-term parameter of a PID control is the force with which the robot will correct itself. The lower the P-term parameter will display the robot's failure to balance itself and a higher P will display an overshoot behaviour.

Integral – I-term parameter of a PID is in control of response time to correcting itself from drifting. Supposedly, the higher the I-term is, the faster it will react to correct tilt/lean.

Derivative – Is in control of sensitivity towards the set point of the control system. It is used to smoothen/depress the robot oscillations. A lower D has no effect to avoid oscillation, the value of D-term supposedly dependant on P & I. The higher the amount of D-term is applied to the system, the more overshoot there is.

3.15 PID Mathematical Explanation

The PID algorithm is summarised by 10:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt})$$

Output $u(t)$ is equal to the sum of the proportional $K_p e(t)$, Integral $K_i \int_0^t e(t)dt$ and Derivative $K_d \frac{de(t)}{dt}$. To achieve proportional only controller, the output equation is a product of gain K_p and error $e(t)$.

$$u(t) = K_p \times e(t)$$

For example, assume an error signal of 0.25 and a gain of 10, the output $u(t)$ returns a value of 2.5.

The error is set-point subtracted from the process value which illustrates how far the plant is from where it should be.

$$e(t) = setpoint - \ process\ value$$

If the gain is too large, it will result to oscillation but tuning will produce a stabilised output by enabling the gain to maintain a steady state error. In order to maintain the gain, a steady state error is achieved which is the distance from set-point namely 'offset'.

To achieve Proportional and Integral $K_i \int_0^t e(t)dt$ control, the following equation is achieved:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt$$

In a PI control, the integral term is an accumulation of instantaneous values that the signal has been, from the set-point. The purpose of the Integral is to accelerate the process/plant towards the set point and to also eliminate steady state error. For example in figure 30 the error signal is represented as the red line; as displayed the error has the same area on the positive and negative side. So therefore the purpose of integrating is to cancel the area and set it to zero.

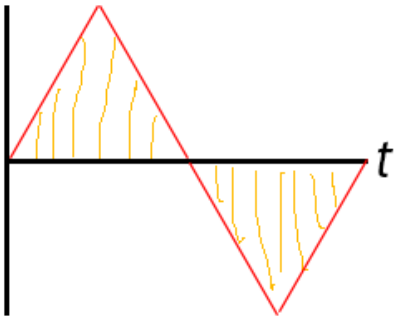


Figure 30 Error Signal

The Derivative is used to control the rate of change of the plant, it aims to reduce the magnitude of the overshoot produced by the Integral parameter and improve stability. By adjusting the rate of change, the effect is very noticeable due to overshoot and so it is important to achieve a stable derivative signal so the robot stays balanced without overshooting. To achieve a PD control the following equation is achieved:

$$u(t) = K_p e(t) + K_d \frac{d}{dt} e(t)$$

The PD controller is a combination of gain and error multiplied and added to the derivative error multiplied the K_d coeeficient to achieve the controller output.

3.16 Initial Setup Arduino PID library

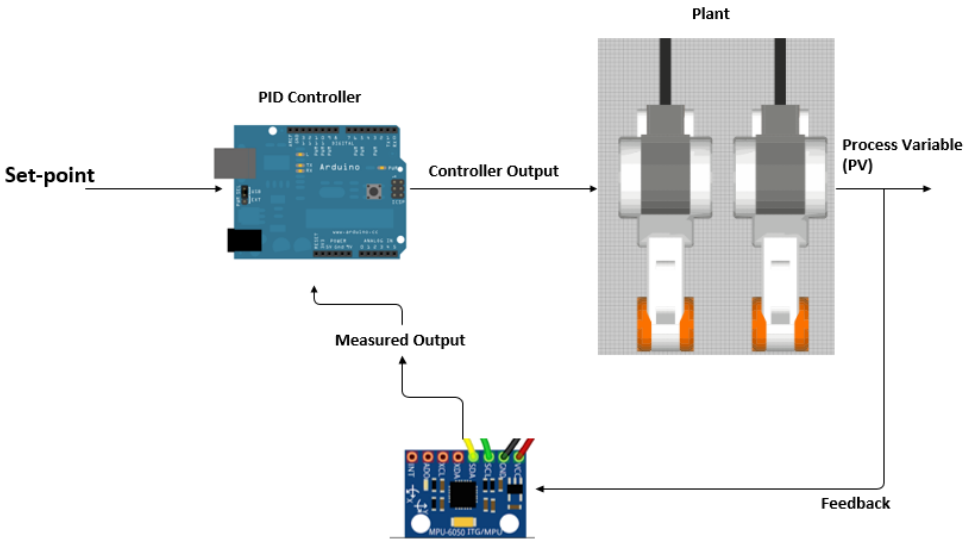


Figure 31 Closed Loop feedback representation

The set-point is the value that we want the process to be. For example, in the Arduino sketch, the set-point is set to 0 this means we want the gyroscope/robot chassis to maintain a stable/balanced position at zero. While the robot tilts, the task of the PID controller is to be able to correct the tilt/lean in order to keep the robot at set-point zero.

The PID controller looks at the set-point and compares it with the actual value of the Process variable known as the Plant. If the values of the set-point and the process variable are the same then the speed of the motors are halted. However, if there is a disparity between the set-point and the process variable we have an error and corrective action is applied whenever the set-point is less or greater than the process variable.

The sensor picks up the angle reading and based on the data output from the gyroscope, a feedback is implemented to notify the PID controller. This loop runs forever until the robot maintains a stable position at set-point to zero

3.17 Implementing Motors

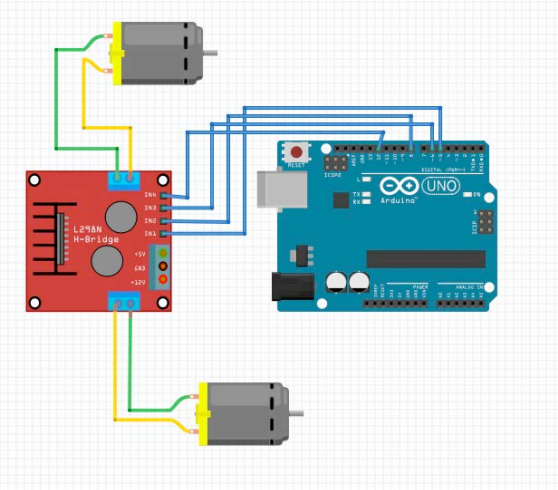


Figure 32 Motor connection to the Arduino

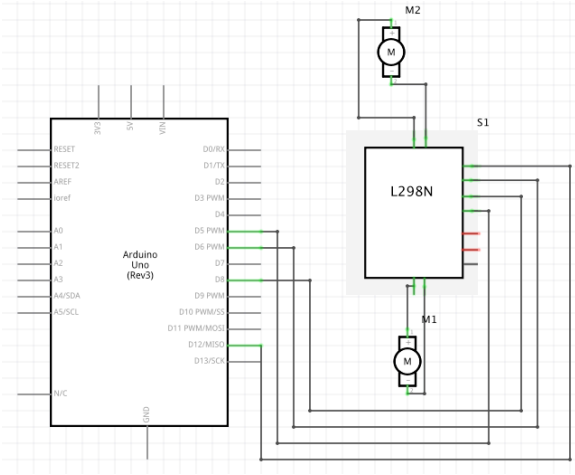


Figure 33 Motor connection to Arduino Schematic

Figure 32 displays a virtual implementation of the NXT motors, the L298N motor driver uses four output pins, which are directly connected to the Arduino. The L298N also use two current sensing pins which are directly connected to the Arduino representing the PWM (pulse width modulation) pins. Although figure … does not display the encoders in use, the current developed stage of the robot does not make use of the encoders.

The Lego NXT motors were central to the functionalities of the robot to achieve a balanced position. The motors were utilized for manoeuvre, otherwise, the robot is incapable of mobility and to achieve a balanced a balanced position would have not been achieved.

The NXT motors were connected directly to the L298N driver to measure the rate of turn from each motor and its direction. This is achieved by changing the output voltage signal and setting its required values accordingly.

PWM is a technique for getting analog results with digital means, PWM provides an output voltage varied by a duty cycle. The duty cycle is determined by an on-off simulation of voltages between a high voltages of 5v compared to 0v ‘pulse width’. The speed of the motors are achieved by updating the percentage of the duty cycle.

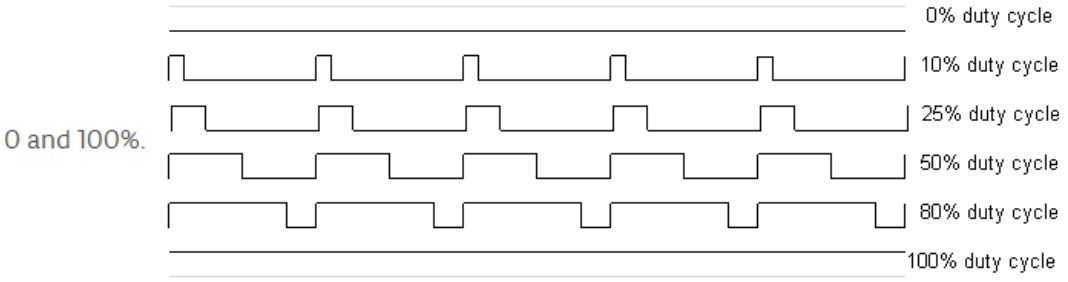


Figure 34 Pulse Width Modulation Duty cycle

The required speed for the motors in order for the robot balanced is usually between 0% and 25% cycle as the robot needs to stay in a still position to balance. This position is called the equilibrium state, this point is consistently exceeded in different direction causing oscillation. The Arduino cannot drive the motors directly at the necessary current so the L298N is utilized for this purpose.

The L298N motor driver consisted of an H-bridge which consisted of multiple LED indication of the direction each motor was driven and was also useful to indicate whether there were any faults.

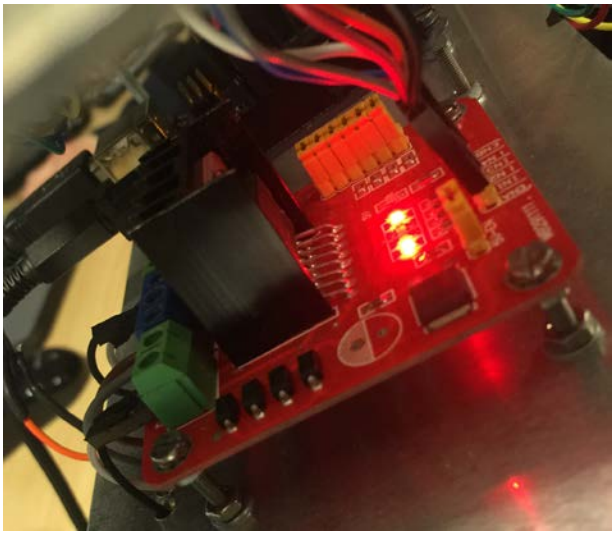


Figure 35 L298N LED indicators

The stalled torque of the NXT motors are approximately 50 N.cm including a stalled current of 2 A. If the robot chassis travels forward, the NXT motors will rotate to correct the lean; same goes for backward tilt. The motor control is powered via a toggle switch which is connected to supply voltage of 7.4V LIPO battery. Although, this was not the case from the initial setup, the initial setup of the robot was powered via a 5 voltage battery. It was then discovered that 5v was not efficient enough to power the NXT while running an early Arduino sketch for the balancing robot. Figure 36 displays a modified version of the base plate for the motors, in order for the NXT motor to fit well. A bracket was also cut out to attach the robot chassis to the Lego NXT. A wire strap was also used to hold tightly the motors also with nuts and washers. Although these materials were not able enough to keep the Lego NXT motors in its comfortable position as the motors were only designed to be used with the Lego NXT Mindstorms development pack.

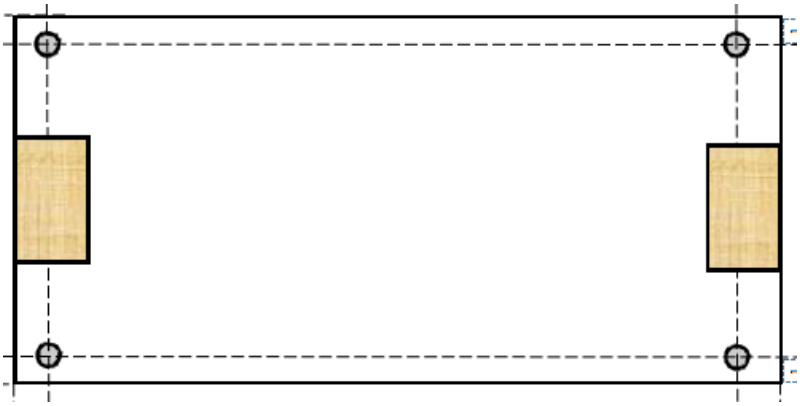


Figure 36 Modified base plate for the NXT motors

Figure 37 Side view of the NXT mounted to the base plate

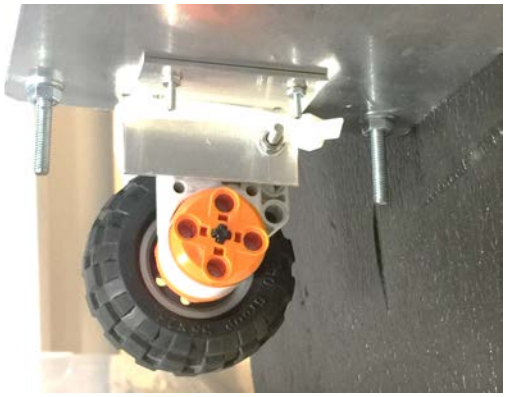


Figure 38 Motor bracket connect to the base plate and motor

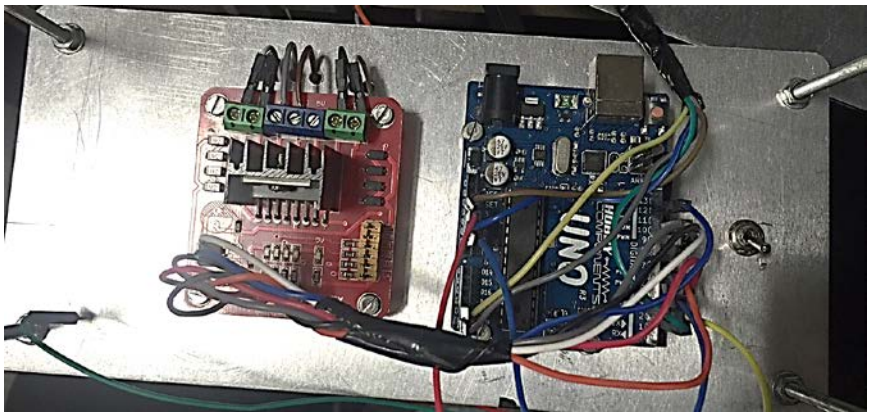


Figure 37 L298N mounted to the Arduino

3.18 Implementing Power Source

A power source is essential for providing energy to a machine, as without power supply the machine would simply not function. If an insufficient power source was implemented. A maschine would not function correctly now would it perform adequately. The robot uses a LIPO battery which is one of the essential component required to maintain autonomous operation.

The LIPO battey required some modification as shown in figure …. The circuit uses a power switch which controls the power supply to the arduino and the l298n motor controller.

Figure 40 Toggle switch mounted on the middle layer of the chassis

Figure 41 Modified connector for the Arduino to the LIPO battery

3.19 Software Code

The main Arduino sketch consists of multiple parts of code gathered from multiple online sources. The final Arduino uses parts from the ‘Chappie Self-balancing robot’ source code [13]. Parts of the source code uses various libraries for the MPU6050 and motor control. Some of the variables were modified to best suit the parameter for the robot system.

3.20 Conclusion

The current stage of the two-wheeled balancing robot is able to balance for at least 2+ minutes, the balancing time varies depending on the platform the robot is tested on (more to follow in section 4). The balancing tends to drift after a certain time, the development of the encoders are still in process to monitor and control the drift.

4 Test Results and Discussion

Testing was mostly done on an ad-hoc basis. Any code change was immediately tested and modified. Data was also occasionally collected and then analysed on the computer using serial monitor.

4.1 Complementary Filter

The Arduino sketch for the complementary filter was not capable of keeping the robot balanced. The PID control which included the complementary on utilized PD controller only. The PD controller was not substantial enough to keep the robot balanced, the robot was unable to accelerate faster to tilt and correct the lean to get into an equilibrium state. As discovered in the later stages of the project, the Integral parameter of the PID proved to be important because the robot was able to react quickly to leaning. The Arduino sketch for the complementary filter was developed from an existing Arduino sketch [12].

4.2 Tuning Parameters

This stage was the most difficult part of the project, as to acquiring the PID values in order for the robot to reach a stable position. The following steps were taken:

The initial circuit enabled me to control the PID values, the setup consisted of using 4 potentiometer, one of which was for controlling the brightness of the LCD screen. The other three potentiometer were for the P, I, and D parameters. By manually controlling the PID parameters via the potentiometer enabled the ability to control the values of the PID without having to continually connect the Arduino on the

chassis to the serial monitor. This method granted the ability to achieve an idea of the range of values to keep the robot to balance for a few seconds. To achieve the PID values, the following procedure took place [13].

1. Set the Proportional, Integral and Derivative terms to 0
2. Adjust the Proportional term so the robot starts to move back and forth about the balance position. The P term should be large enough for the robot to move but not too large otherwise the movement would not be smooth.
3. Adjust I term by increasing to the point whereby the robot accelerates faster when off balance.
4. Increase D so that the robot would move about its balanced position more gentle otherwise significant overshoots will occur.

The PID controller testing was manual. After each retune of the controller, I would subjectively access the stability of the robot. Since the tuning procedure was quite simple, this was not a problem. The jitter was occasionally noted down. These measurements were just to provide a general indication of the performance of the algorithm for tuning it even further. Good PID settings were logged in the log book before being changed.

Figure 42 PID to LCD setup

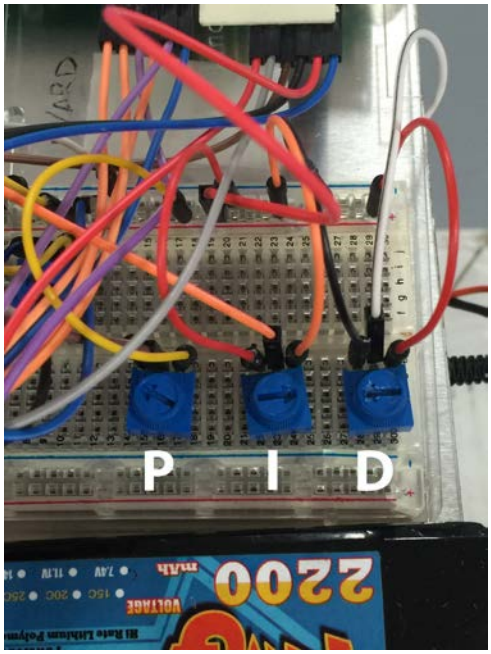


Figure 43 PID control parameter setup via three Potentiometer

The initial PID output limits by default were between 0 and 255 due to the settings of the function namely ‘SetOutputLimits(min, max)’ from the Arduino PID library. Below is a table of the initial values attempted to balance the robot via trial and error.

Table 5 Initial PID values achieved

Proportional	Integral	Derivative
60	90	0
70	100	0
70	100	0
70	100	10
80	150	5
70	150	8
80	150	8
100	150	8
60	150	8
65	150	8
65	150	10
65	150	9

At this stage of fine tuning the parameters, some of the results from table proved to cause either too much overshoot or less responsive integral to tilt.

The results from the table above were not responsive and the robot drifted a lot. The default PID output limits were set to 255 max to match the PWM duty cycle. The I-term at its max (255) was not enough for the robot to accelerate faster when tilting. It was then later on discovered that the output limits could be modified. The next solution was to increase the output limits, the following output limits was discovered through trial and error using the ‘SetOuputLimits()’ function. A minimum output limit of ‘-8000’ was selected and a maximum of ‘8000’ was used.

```
pid.SetMode(AUTOMATIC);
pid.SetOutputLimits(-8000, 8000);
pid.SetSampleTime(10);
```

Figure 44 Setting PID output limits

The following PID values were achieved after the modification of the pid output limits.

```
// Balance PID controlle
#define BALANCE_KP 100
#define BALANCE_KI 1000
#define BALANCE_KD 9
```

Figure 45 Values of the PID parameter achieved

As displayed in figure … the Ki parameter is more set to 1000+, the floating number after 1000 was achieved by using a Matlab script which consisted of a PID controller for a DC motor modelled in Simulink. The Simulink program was based on a closed-loop system by using the PID controller block. Using the PID block enable to ability to tune the gains of the PID controller. Although the Matlab only used DC motor, meanwhile the Lego NXT were servo motors, the effect of using this script invalidated but was still implemented into the Arduino code. A comparison without the floating number was put to test, and was concluded that fine tuning with Matlab was useful even though the changes were not significant enough for the stability of the robot.

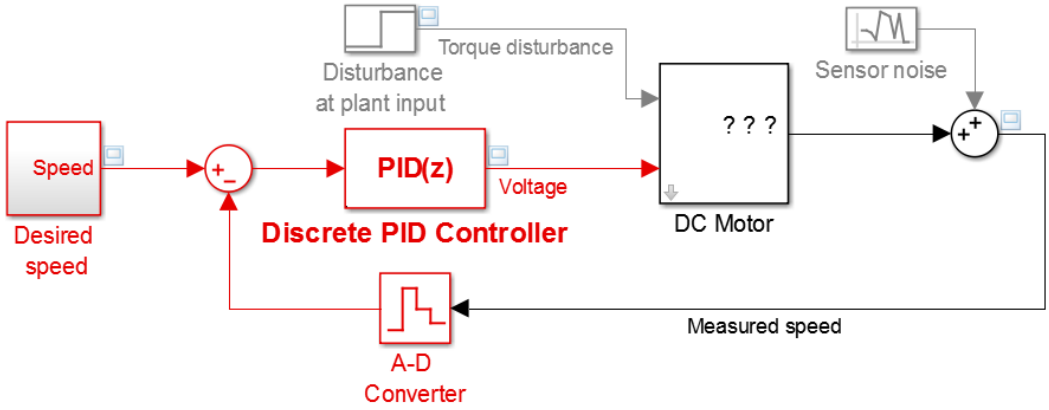


Figure 46 Simulink PID Control Simulation

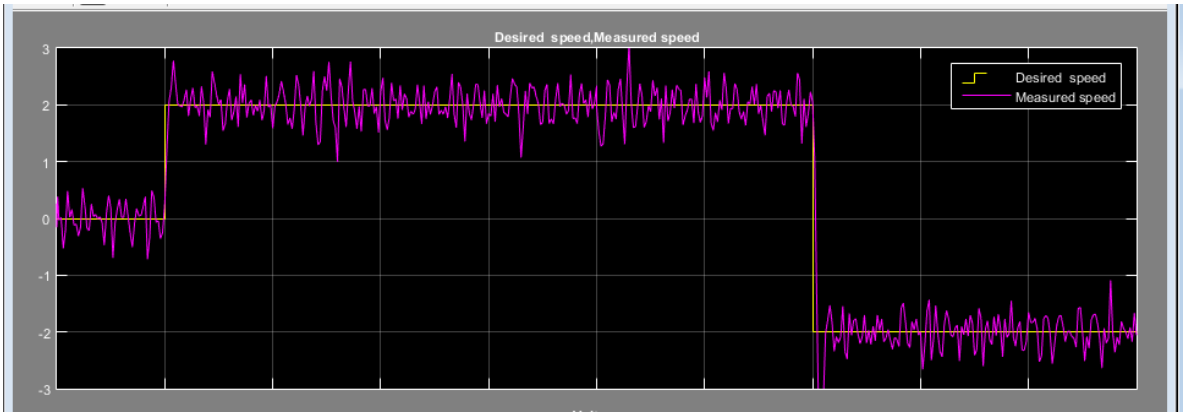


Figure 47 Simulink PID response to desired output and measured output

```
#define BALANCE_KP 100.463921040815
#define BALANCE_KI 1000.79326368624
#define BALANCE_KD 9.21893218402366
```

Figure 48 Simulink PID response

4.3 Balancing Performance

The PID gains from figure 48 were achieved through using the Simulink PID controller. Even with these values, the robot is capable of balancing in certain testing circumstances. For example in figure… the robot is able to balance with additional

components placed on the chassis. Through continuous testing it was discovered that the PID parameters achieved did not suit all testing environment. It was discovered that the robot was able to main stability longer on a carpet than on the wooden platform. The current stage of the robot is also unable to counter tethering (pushing), during testing the motors were unable to withstand pushing, and instead as soon as the robot is pushed the motors will drift away and fall.

Figure 49 balancing an object on the base layer of the Robot

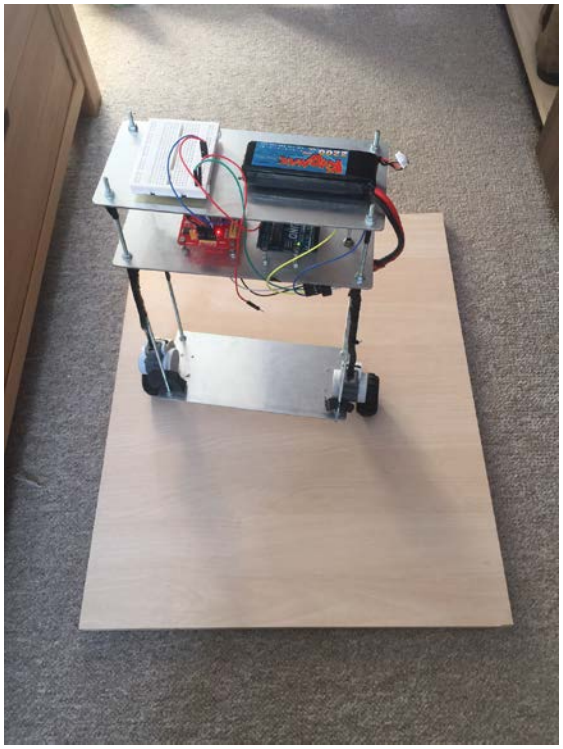


Figure 50 Balancing on a wooden platform

5 Conclusion and Recommendations

5.1 Conclusion

The overall project was successful in achieving a two-wheeled self-balancing robot even though it was unable to counter itself from tethering (human interaction by pushing). The balancing robot was able to balance for at least 5 minutes+. Part of considerations for achieving a self-balancing robots was due to the completion of the objectives stated at the early stage of the project. Some difficulties were encountered, mainly in the programming area, theory and hardware implementation i.e. PID control and Fusion Algorithm. The majority of open sources online developed a guide on meeting the objectives. Initially the development of the program was unsuccessful, however due to more research the difficulties were conquered; credits to the open sources provided for the implementation of the balancing robot. Overall, part of the deliverables were achieved apart from the robot being able to counter tethering and the implementation of the encoders. The performance of the DMP was up to expectations as there were no errors that occurred. The DMP accurately achieved the gyroscope and accelerometer data outputs.

5.2 Recommendations

The following recommendations are provided to enhance the functionalities of the robot for the future.

- Use a better appealing chassis material, for example plexi glass. Although, this has no effect on system performance.
- Reduce the height of robot’s chassis to counter drift.
- Add weights to counter balance the robot.
- The current stage of the motor is able to balance 2+ minutes depending on the testing environment. Although the robot is unable to counter tethering, one way avoid drifting could be achieved by integrating the encoders.
- Create a PCB to avoid untidiness of cables floating around.

- The use of the Lego NXT were not suitable for the project, a more efficient motor is using a DC motor.

List of Tables

References

[1] Robot Institute of America, 1979

[2] Future directions in control in an information-rich world - R.M. Murray, K.J. Astrom, S.P. Boyd, R.W. Brockett, G. Stein. Appears in IEEE Control Systems Magazine 2003, Page 10. - http://users.cms.caltech.edu/~murray/preprints/mur+03-csm.pdf

[3] Inverted Pendulum -https://en.wikipedia.org/wiki/Inverted_pendulum

[4] Segway - http://www.segway.com/

[5] A Pilot Study: The Segway Personal Transporter as an Alternative Mobility Device for People with Disabilities. Presented in part to the International Seating Symposium, March 11-14, 2007, Orlando/Lake Buena Vista, FL. Bonita Sawatzky, Ian Denison, Shauna Langrish, Shonna Richardson, Kelly Hiller, and Bronwyn Slobogean; Volume 88, Issue 11, Pages 1423-1428. - http://www.archives-pmr.org/article/S0003-9993(07)01343-3/fulltext#sec1

[6] nBot Balancing Robot - http://www.geology.smu.edu/~dpa-www/robo/nbot/

[7] Balanduino by TKJ Electronics - http://wiki.balanduino.net/Overview

[8] InvenSense - http://www.invensense.com/products/motion-tracking/6-axis/mpu-6050/

[9] Geek-mom-projects - http://www.geekmomprojects.com/mpu-6050-redux-dmp-data-fusion-vs-complementary-filter/

[10] PID Control - http://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/astrom-ch6.pdf

[11] PID Controllers - http://www.codeproject.com/Articles/49548/Industrial-NET-PID-Controllers

[12] Gyroscopes, Accelerometers and the Complimentary Filter - https://bayesianadventures.wordpress.com/2013/10/20/gyroscopes-accelerometers-and-the-complementary-filter/

[13] PID tuning steps - http://www.instructables.com/id/Chappie-Self-Balancing-Robot/?ALLSTEPS

[14] I2CdevLibs - http://www.i2cdevlib.com/devices/mpu6050#source=

[15] L298N datasheet - http://www.tech.dmu.ac.uk/~mgongora/Resources/L298N.pdf

Appendix A

Arduino Sketch Code (C programming)

```
#include <I2Cdev.h>

#include <Wire.h>

#include <MPU6050_6Axis_MotionApps20.h>

#include <helper_3dmath.h>

#include <PID_v1.h>                          //Arduino PID library

#include <digitalIOPerformance.h>            //library for faster pin R/W

#define DIGITALIO_NO_INTERRUPT_SAFETY

#define DIGITALIO_NO_MIX_ANALOGWRITE


#define RESTRICT_PITCH

MPU6050 mpu;

// MPU control/status vars

bool dmpReady = false;  // set true if DMP init was successful

uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU

uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)

uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)

uint16_t fifoCount;     // count of all bytes currently in FIFO

uint8_t fifoBuffer[64]; // FIFO storage buffer


// orientation/motion vars

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

```
VectorFloat gravity;    // [x, y, z]            gravity vector
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector


volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin has gone high
void dmpDataReady() {
   mpuInterrupt = true;
}


// Balance PID controller Definitions
#define P 100
#define I 1000
#define D 9
#define BALANCE_PID_MIN -255              // Define PID limits to match PWM max in reverse and foward
#define BALANCE_PID_MAX 255


#define ROT_KP 50.4376950551922  // 50
#define ROT_KI 500.195586913476  // 300
#define ROT_KD 4.66968339454473 //4


#define IN1      5        //6
#define IN2    8        //12
#define IN3      6        //M21
```

```
#define IN4    12       //M22
#define ENA     9       //M1E
#define ENB     10      //M2E


#define LOOPTIME        100                 // PID loop time
#define FORWARD         1                   // direction of rotation
#define BACKWARD        2                   // direction of rotation

unsigned long lastMilli = 0;                // loop timing
unsigned long lastMilliPrint = 0;           // loop timing
long count = 0;                             // rotation counter
long countInit;
long tick_no = 0;
boolean run = false;                        // motor moves

// Motor Misc
#define PWM_MIN 0
#define PWM_MAX 255
float ENA_SLACK=100;
float ENB_SLACK=0;
#define ENA_MAX 255
#define ENB_MAX 255
```

```
int Aspeed, Bspeed, MotorSlack,moveState=0,d_speed,d_dir;

double yaw,input,out,setpoint,originalSetpoint,Buffer[3];
double yinput,yout,ysetpoint,yoriginalSetpoint;

double bal_kp,bal_ki,bal_kd,rot_kp,rot_ki,rot_kd;

PID pid(&input,&out,&setpoint,P,I,D,DIRECT);
PID rot(&yinput,&yout,&ysetpoint,ROT_KP,ROT_KI,ROT_KD,DIRECT);

String content = "";
  char character;

void setup()
{
  #ifdef DEBUGING
  Serial.begin(115200);
  #endif

  init_imu();

  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
```

```
pinMode(IN4, OUTPUT);
analogWrite(ENA, 0);
analogWrite(ENB, 0);

pid.SetMode(AUTOMATIC);                 //For info about these,see Arduino PID library
pid.SetOutputLimits(-8000, 8000);
pid.SetSampleTime(10);
rot.SetMode(AUTOMATIC);
rot.SetOutputLimits(-50, 50);
rot.SetSampleTime(10);

setpoint = 1;
originalSetpoint = setpoint;
ysetpoint = 0;
yoriginalSetpoint = ysetpoint;



bal_kp=P;
bal_ki=I;
bal_kd=D;
rot_kp=ROT_KP;
rot_ki=ROT_KI;
rot_kd=ROT_KD;
```

```
          /*
SetTunings(...)*************************************************************
          * This function allows the controller's dynamic performance to be adjusted.
          * it's called automatically from the constructor, but tunings can also
          * be adjusted on the fly during normal operation

*******************************************************************************
***/
          pid.SetTunings(bal_kp,bal_ki,bal_kd);                    //change PID values
          rot.SetTunings(rot_kp,rot_ki,rot_kd);


         }


         void loop()
         {
          getvalues();       //read values from imu
          new_pid();         //call pid
         }


         void init_imu()
         {
          // join I2C bus (I2Cdev library doesn't do this automatically)
          Wire.begin();
          mpu.initialize();
          devStatus = mpu.dmpInitialize();
```

```
        // supply your own gyro offsets here, scaled for min sensitivity
        mpu.setXGyroOffset(30);
        mpu.setYGyroOffset(-9);
        mpu.setZGyroOffset(31);
        mpu.setXAccelOffset(-2702);
        mpu.setYAccelOffset(-1235);
        mpu.setZAccelOffset(1154);
        // make sure it worked (returns 0 if so)
        if (devStatus == 0) {      // turn on the DMP, now that it's ready
            mpu.setDMPEnabled(true);
            // enable Arduino interrupt detection
           // Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
            attachInterrupt(0, dmpDataReady, RISING);
            mpuIntStatus = mpu.getIntStatus();
            dmpReady = true;
            // get expected DMP packet size for later comparison
            packetSize = mpu.dmpGetFIFOPacketSize();
         }
       }

       void getvalues()
       {
           // if programming failed, don't try to do anything
          if (!dmpReady) return;
```

```
        // wait for MPU interrupt or extra packet(s) available
        while (!mpuInterrupt && fifoCount < packetSize) {
        }
        mpuInterrupt = false;
        mpuIntStatus = mpu.getIntStatus();

        // get current FIFO count
        fifoCount = mpu.getFIFOCount();

        // check for overflow (this should never happen unless our code is too inefficient)
        if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
           // reset so we can continue cleanly
           mpu.resetFIFO();
        } else if (mpuIntStatus & 0x02) {
           // wait for correct available data length, should be a VERY short wait
           while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

           // read a packet from FIFO
           mpu.getFIFOBytes(fifoBuffer, packetSize);

           // track FIFO count here in case there is > 1 packet available
           // (this lets us immediately read more without waiting for an interrupt)
           fifoCount -= packetSize;
```

```
            mpu.dmpGetQuaternion(&q, fifoBuffer);
            mpu.dmpGetGravity(&gravity, &q);
            mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
          }
          yinput = ypr[0]* 180/M_PI;
          input = -ypr[1] * 180/M_PI;          //change sign if negative
        }



        double commit_slack(double yOutput,double Output,bool A)
         {
          //Compensate for DC motor non-linear "dead" zone around 0 where small values don't result in movement
          //yOutput is for left,right control
          if(A)
          {
           if (Output >= 0)
           Output = Output + ENA_SLACK - yOutput;
           if (Output < 0)
           Output = Output - ENA_SLACK - yOutput;
          }
          else
          {
            if (Output >= 0)
```

```
  Output = Output + ENB_SLACK + yOutput;
  if (Output < 0)
  Output = Output - ENB_SLACK + yOutput;
 }
  Output = constrain(Output, BALANCE_PID_MIN, BALANCE_PID_MAX);
 return Output;
}

void new_pid()
{
 //Compute error
 pid.Compute();
 rot.Compute();
 //Convert PID output to motor control
 Aspeed = commit_slack(yout,out,1);
 Bspeed = commit_slack(yout,out,0);
 nxtSpeed(Aspeed, Bspeed);      //change speed
}

// Motor control functions
void nxtSpeed(int Aspeed, int Bspeed) {
 // Motor A control
 if (Aspeed >= 0)
 {
```

```
    digitalWrite(IN1,HIGH);
    digitalWrite(IN2,LOW);
  }
  else
{
  digitalWrite(IN1,LOW);
  digitalWrite(IN2,HIGH);
}

  analogWrite(ENA,abs(Aspeed));

  // Motor B control
  if (Bspeed >= 0)
  {
    digitalWrite(IN3,LOW);
    digitalWrite(IN4,HIGH);
  }
  else
  {
  digitalWrite(IN3,HIGH);
  digitalWrite(IN4,LOW);
  }
  analogWrite(ENB, abs(Bspeed));
}
```