Engineering Thesis Project

Smart Two Wheels Balancing Robot

Academic Supervisor: Dr Nasser Asgari

Student Name: Hasan Alshahrani

Student ID: 2207449

Monday, 29 May 2020

Submitted to the College of Science and Engineering in partial fulfilment of the requirements for the degree of Master of Engineering (Electronics) at Flinders University-Adelaide Australia.

Certificate of Originality

I certify that this work does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university and that to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature:

Hasan Nasir Alshahrani

Monday, 29 May 2020

Acknowledgements

I am thankful to those with whom I have had the pleasure to collaborate with in this project. Every one of the individuals from my Dissertation Committee has accorded me with broad personal, as well as professional guidance, and imparted me with both scientific research and life in general. I might particularly want to express gratitude toward Dr. Nasser Asgari, my supervisor and Topic Coordinator. I will never forget his support. As the college Engineering Services Team, they have shown me beyond what I would ever give him recognition for here. Leading by example, they have demonstrated to me what a good engineer should be.

Also, I would like to extend my sincere gratitude to my friends and family for their endless support for this project. Special thanks to my parents; for their guidance and adoration in whatever I pursue and for being my ultimate role model.

Abstract

This project is about a smart two wheels balancing robot that has the capability of self-balancing and can move from one location to another. Three main goals were identified for this project, i.e., balancing the robot using the PID algorithm, making the robot move while maintaining the balance using the RC transmitter and receiver, and autonomously making it navigate using GPS, compass, wheel encoders, and other sensors associated with Mission Planner software. For the achievement of these goals, an extensive literature review was performed to understand the concept and working of self-balancing robots. By utilising the available literature, it was determined that the inverted pendulum theory and control theory must be adopted for the design and development of the smart two wheels balancing robot system.

By using the theory, this robot system was developed with the ability to rotate its motor in the direction of the tilt. A cascaded PID algorithm was successfully developed to control the heading angle, pitch angle, and wheel velocity of the robot, which allows the robot to perform balancing and autonomous navigation tasks. This system is programmed to rotate the motor and wheels in the direction of the tilt of the robot's body, in both forward and backward directions. The motors of the robot are programmed to keep on rotating until the body of the robot reaches its set-point angle, which is 0°, i.e., the upright position of the robot. A simulation using Matlab/ Simulink is used to verify the model and tune the constant used for the PID algorithm. Two programming environments are used to program the ArduPilot Mega (APM 2.8) microcontroller and Arduino Pro Mini microcontroller, these are; ArduPilot-Arduino and Arduino IDE respectively. Additionally, Mission Planner software is used to control the robot in autonomous navigation.

Contents

List of Figures

List of Equations

List of Tables

Chapter 1: Introduction

This project involves a smart two-wheeled balancing robot, which has the ability of self-balancing along with autonomous navigation. The motor is programmed to counteract the robot's tilt or falling motion to ensure that the robot maintains its balance. The operation mainly contains correcting elements or actions and feedback. The smart two wheels balancing robot is based on the concept of an inverted pendulum. Balancing is better when the centre of mass is higher in comparison with the robot's wheel axes. The higher mass centre means a greater moment of inertia that corresponds to lowered angular acceleration and slower fall (Ghani et al., 2011). With the consideration of control theory, it is essential to keep some variables, in this case, steady, i.e., the attitude and velocity of the robot, with the use of a unique controller known as the Proportional Integral Deviation (PID).

This project aims at providing valuable information regarding the smart two wheels balancing robot project backed by relevant evidence sourced from various online articles, journal articles, and research papers. While performing the literature review, the referenced research papers and articles have evidence-based information about the working and designing of the smart two wheels balancing robot. One of the primary objectives of this report is to provide sufficient information and data on the various ways through which the robot can be designed (Wu et al., 2011). This project has three main goals:

I. Making the robot balanced using the PID algorithm.
II. Making the robot move while balancing using the RC transmitter and receiver.
III. Making the robot navigate with the help of GPS, compass, and wheel encoders associated with Mission Planner software.

Another key aim of the project is to develop a nontraditional robot that can navigate quickly through various terrains, obstacles, change the direction instantly in place, and navigate efficiently in narrow places in which robots with +3 wheels cannot perform successfully.

In theory, the inverted pendulum two wheels self-balancing robot is naturally unstable. However, a larger moment of inertia is created due to the higher centre of gravity, which reduces the rate at which the robot falls (Mahler & Haase, 2013). This slow fall can be leveraged by the continual movement of the wheels of the robot in the direction in which it falls to keep the balance.

The robot will be programmed in such a way that if the body of the robot leans forward, then the wheels of the robot will roll forward to counteract the fall (Azimi & Koofiger, 2013). For balance control, a PID loop is integrated into the software. In this PID loop, the proportional parameter takes the error of the angle and sends it to the motor to keep the wheels of the robot rolling towards the fall direction. The integral parameter of the PID takes the total error of all the angles, which helps in cancelling out the issues of the centre of gravity (Sadeghian & Masoule, 2016). Furthermore, the derivate parameter of the PID loop is critical as, without it, the acceleration of the robot cannot be controlled.

The movement of the smart two wheels balancing robot is different from ordinary robots. The basic robot simply leans in the direction of the movement. This might work for a short time, but the robot accelerates quickly and eventually falls. If it tries to get back to the correct position, the forward motion of the robot will stop (Peng et al., 2012). Instead, the robot has to move forward vertically while balancing.

Another critical component of the robot is its ability to navigate. To make a robot navigate, it is crucial to know precisely the latest position of the robot and where it is headed (Sun et al., 2015). Using GPS is a handy option, but it is only accurate to a few metres. To attain a fair accuracy down to the centimetre scale, more improved technology is needed. Wheel encoders are a better choice for the smart two wheels balancing robot. Wheel encoders allow close to fair accuracy in millimetres and are considered an excellent complement to the GPS (Warren et al.,2011). Furthermore, the Arduino Pro Mini microcontroller will be used to read the pulses per second sent by encoders. The data is relayed towards the APM microcontroller through the I2C interface. The rest of the robot's controlling software will include the control software containing the waypoint navigation, which is the simplified and modified model of the ArduCopter that was the open-source drone project. Additionally, the robot is controlled for autonomous navigations using Mission Planner software. This project is completed in seven stages; these are:

I. The first stage is to search and gather all the needed components for the robot, i.e. 2.8 APM microcontroller, 328 Arduino Pro Mini microcontroller board, telemetry radios, R/C transmitter and receiver, GPS model, compass model, motor driver shield, encoders, motors, and wheels.

II. The second stage is the design and structure of the robot, including 3D printing parts.

III. The third stage is the connection of the electronics, connecting the APM, Arduino Pro Mini, motor driver shield, and GPS module along with the telemetry radio for the autonomous operation. For controlling the robot manually, the R/C receiver is connected (Warren et al.,2011). Finally, the electronics circuit of the robot will be connected to its motors.

IV. The fourth stage is the final assembly of the robot in which all the three main components of the robot, i.e., the electronics and motors after the connection, are integrated into the robot body, and the wheels are attached to the robot.

V. The fifth stage is the simulation using Matlab/Simulink to ensure that the robot is stabilisable or controllable with some parameters that need to be tuned.

VI. The sixth stage is programming the robot using ArduPilot-Arduino software to program APM and Arduino IDE software to program the Arduino Pro Mini microcontroller.

VII. The final stage of the project is testing the robot to ensure that the system works as expected and solving any issues encountered in the working behavior of the robot.

Chapter 2: Literature Review

The two wheels balancing mobile robots have been an active field of study as they provide a general mechanical design with higher manoeuvrability. In this field, different developments have been made in the method of ensuring more stability and autonomous navigation of the robot from one place to another. According to Romlay, Albert, and Chris (2019), the nonlinear controller, linear controller, along with self-adapting controller all contribute in effective control of the two wheels balancing robot systems. Mobile robots have the capability of performing sophisticated tasks of carrying loads and avoiding obstacles on their own. That is why the experiments and tests are done on robots focusing on the desired speed, setting trajectory, obstacle avoidance, balancing, and achievement of the desired rates (Romlay, et al., 2019). However, it is essential to point out the fact that it is tough to compare the controller system of the robot entirely as a testing subject along with the requirements of every research because they are different. That is why the more direct objective and comparison of a mobile robot is demonstrated for really evaluating the effectiveness and robustness of the control method in future research. The research of Romlay et al. (2019), presents controller methods for self-balancing two-wheeled robots in the nonlinear controller, linear controller, along with the adapting an algorithm (self-learning) theme. Experiments that were performed in this research were based on evaluating the robot system in their speed, trajectory, acceleration, interactions to their surrounding area, and avoidance of obstacles.

In recent years, the research regarding the two wheels balancing robots has gained momentum in a wide variety of robotics research and development centres around the world. Much of this traction is due to the natural unstable dynamics of these robotic systems (Ghani et al., 2011). As the two wheels balancing robot requires an adequate controller for maintaining itself in the upright position without the requirement of any external force. Thus, it is crucial to develop two wheels balancing robot with an excellent platform and explore the various controllers' efficiency in the control-based system on an inverted pendulum model. Currently, different controllers are being implemented upon the two wheels balancing robots like, for example, the controller of pole placement, quadratic linear regulator, the fuzzy controller logic, and the PID controller. These robots can be categorised by their ability to balance on their own two wheels and spin at the spot (Juang & Lurrr, 2013). Because of the result of the added manoeuvrability, easy navigation is a

possibility for these robots on different terrains. Also, they can make sharp turns, traverse curbs, and small steps and have the ability to carry loads as well.

Furthermore, the two wheels balancing robots have a small footprint as compared to the three, four, or five wheels robots, which enables them to move around the corridors and the tight corners much more quickly. These capabilities of two wheels balancing robots have the potential of solving many challenges in the robotics industry and the society in general (Miasa et al., 2010). As a result of its diversity in applications, this robot shows that it can be used as a human transport machine like the hoverboard products. The models like iBot, Pegasus, and Segway are an example of a design of two wheels balancing robot that can be used as the human transport machine (Miasa, et al., 2010). Additionally, the motorised wheelchair systems also utilise this technology by providing the operator of the wheelchair system with more excellent manoeuvrability and accessibility to places that are difficult to reach for disabled persons.

The two wheels balancing robot's design and functionality are quite like an inverted pendulum. With the two wheels balancing robot case, the weight of the robot instead of being on its bottom surface is placed on the top of the robot, and it is ensured that the robot balances that weight while it is moving to ensure that it does not fall (Alarfaj & Kantor, 2011). This is based on the control principle, and this is the simple principle that has been used for several years (Alarfaj & Kantor, 2011). The control principle mainly means balancing an object to ensure the equilibrium state or prevent the object from falling, for example, balancing a stick on one's finger or balancing a rotating football on one's finger. This same concept can be used in the two wheels balancing robot systems by ensuring that the robot moves in the direction where it's top is falling. These actions allow the robot to keep centre of mass of its body directly above its centre of gravity or the base to bring its top back into an equilibrium position (Yau et al., 2009). The two wheels balancing robot is freefall system because these robots are an unstable system with two wheels, meaning that the robot might fall either in a forward or backward direction without the influence of any internal or external force. This simple logic, if the robot is not balanced the motors of the robot has to move in the direction of the falling course to keep the top in an equilibrium position (Unluturk et al., 2013). Therefore, knowing about the direction in which the robot has to move, one has to simply activate the motors of the robot to balance the robot.

The two-wheeled robots have a significant advantage in comparison with the robots (humanoid type) because two wheels robots are much quicker and have the capability of changing their direction much more quickly while moving. Moreover, it is making them helpful in different real-world applications. In the category of wheeled robots, the two wheels balancing robots, i.e., Ninebot and Segway, are becoming more and more popular and are being used as patrol transporters or commuting. Additionally, the self-balancing robots like QB Anybots are now in use as the robot service platform (Chan et al., 2013). Due to their rising popularity, self-balancing advanced robots are developed now for real-world applications. Golem Krang robot was established in Georgia technology institute, which can be taken as an example of two wheels balancing robot system. The base of the robot is similar to the general two wheels robot, but the only difference is that this robot has anthropomorphic arms. The two wheels of a robot are mainly utilised for balancing the robot's robotic arms and body that are fixed on its upper body, and it is primarily designed for performing tasks like moving objects or obstacles. Another example of two wheels balancing robot is the Ballbot, which was designed at the University named Carnegie Mellon (Thao et al., 2010). The Ballbot robot system was designed in a way that it is capable of balancing itself on a sphere, which allows the robot to switch orientation and movement easily.

Furthermore, another developer named KAIST has developed two wheels balancing robot that has the upper body having 5 degrees of freedom; DOF. With the use of its upper body, the robot has the capability of maintaining the dynamic balance basing on 0-moment point (An & Li, 2014). Different types and forms of balancing robots are being used at various applications and in different environments, which is why it is vital to ensure the prevention of accidents. This importance was heightened when the owner of Segway died while riding the two-wheeled Segway robot in 2010 in an accidental fall. Moreover, the same accidents have occurred with two-wheeled self-balancing robots, which illustrates why it is essential to ensure that the stability of the robot is considered as much as possible in its design (Sun & Gan, 2010). There are more effective control algorithms to improve the stability of the balancing robots, but they do not guarantee safety. The self-balancing part of the robot is critical in its design because it has the role of maintaining the balance of the robot while it is moving through various terrains. The robot maintains the balance using movements of the body and the wheels. The robot is driven with the help of two actuators that consists of the DC motor along with the belt/pulley mechanism (Lee & Jung, 2012). The DC motors that are installed in the robots are Amp flow (A28-150) DC motors that are operated at

approximately 24V and have around 3 HP (i.e., 2.2 kilowatts) power. This motor has the maximum speed at approximately 6100 rpm, along with the maximum torque rate at 13.9 Nm. To increase the actuator torque, the belt/pulley mechanism can be used. It is vital to ensure that the reduction ratio is at 10:1 approximately. Due to the pulley's limited size, the two-belt/pulley mechanisms are connected serially (Lee et al., 2013). This means that the single belt/pulley mechanism's reduction ratio is chosen to be approximately 3.14:1 (i.e., 44:14), along with the reduction ratio final at 9.88:1 (i.e., 44^2:14^2). Furthermore, each motor of the robot has the rotatory encoder at 1000 ppr for measuring the angle.

To measure the tilt angle along with the robot's angular velocity, the Inertial Measurement Unit (IMU) is made. IMU mainly has the inclinometer (M1, DAS) along with the gyroscope (CRS 03-02, sensing Silicon systems). Although the inclinometer has the capability of accurately measuring the tilt angle of the robot's body, it is incapable of measuring the accurate angle at the first rotation due to the centrifugal force. The gyroscope in two wheels balancing robot is used in measuring the angular velocity of the body of the robot to help in estimating the angle of the body with the integration of the angular velocity. The drift might happen because of the error accumulation at a steady-state (Su et al., 2010). For compensating the limitations of two sensors, complementary filters are applied. The additional filters have the lower pass filter for the inclinometer along with the higher pass filter for the gyroscope. Both screens cutting-off frequencies are determined experimentally. The Control Moment Gyroscope (CMG) module can be used in two wheels balancing robots, and this is placed on a balancing robot. This module mainly consists of two CMG's, where each CMG has approximately two actuators. One of the actuators is responsible for rotating the flywheel, and the other actuator is responsible for turning the gimbal where the wheel is present (Almeshal et al., 2013). CMG creates the torque in the direction, which is perpendicular to the rotational axes of gimbal and flywheel.

Torque magnitude is the product of angular velocity at momentum and gimbal of the flywheel. So as to generate enough amount of torque, the vital thing is to have a more significant moment of inertia and the quicker flywheel speed. For assuring that the CMG torque is accurate, two flywheels are required to be rotated at a constant rate. Every flywheel is, for this reason, is integrated with the 270 Watts DC motor (MABUCHI, RS-775 wc) along with the rotatory encoder for controlling rotation. In the CMG, the gimbal motor used does not require high speed as it

requires just a sufficient amount of torque. That is why the DYNAMIXELMX-106T from robots can be selected (Yim et al., 2018). When balancing two wheels robots get any sort of disturbance, the stability of the robot is the main thing that is affected by the disorders that are then integrated with the backward and forward directions. Security of the two wheels balancing robot must be handled dominantly at a sagittal plane. The CMG module of the robot generates torque with a concept of precession motion.

This is done when flywheel rotates, along with gimbal, which contains the wheel rotating along the direction that is perpendicular towards flywheel rotational axis, i.e., y-axis, torque which is generated alongside the course of a cross product in both vectors, i.e., the x-axis (Gonzalez et al., 2017). With the utilisation of the CMG controller, the standard two wheels balancing robots are capable of maintaining their balance while moving even if they face significant disturbances. For maintaining the stability of the robot against disorders, the robot moves the wheels and tilts the body to ensure equilibrium. For safety, the robot must be in its location, or when the disturbance happens, the CMG module of the robot is capable of generating the reaction torque, which corresponds to interference. CMG module that is integrated into the two wheels balancing robot operates gimbal motor with the utilisation of the PID controller, based on the feedback of disturbance measured by an observer. The aim is the elimination of the impacts of trouble; the desired disturbance F_r is settled at 0 (Chiu et al., 2011). The control gains of the robot can be experimentally determined because the CMG controller uses observed disturbance with the inclusion of each sensor noises. Preferable is to set D gain small as it can improve the sound of sensors.

It is observed that the application of a CMG module improves the performance of the robot. However, it is difficult to remove the effects of disturbances entirely. This is because the CMG module generates an insufficient amount of torque based on its hardware specifications. However, through simulation, there is a possibility of verifying the effects of an improved CMG module (Dai et al., 2015). The two wheels balancing robots usually have to change their position to maintain their balance when an external force is applied. This is not an issue for the robot if only the stability of the system is considered. However, if the robot is moving in a narrow space, the movement, which is caused by the disturbance, can cause a real issue (An & Li, 2014). To solve this issue, the best option is to use the CMG module because, through experimentations, it is confirmed that the

CMG module is capable of generating a sufficient amount of torque for decreasing the effect of a disturbance.

Furthermore, the CMG module is capable of distributing the burden of the wheel motors. Although the CMG module reduces the movement and the tilt of the body of an experimental robot when the disturbance is applied, the performance is not perfect (Gonzalez et al., 2017). Through simulation, it is verified that the performance of the robot is improved by using the CMG module that generates a higher amount of torque.

The two wheels balancing robot that uses a Proportional–Derivative (PD) fuzzy control method can be designed and analysed in multivariable, higher-order, strong coupling, nonlinear and unstable systems (Wu et al., 2012). The existing research that is present regarding the fuzzy reasoning is divided into three different categories, i.e., the first is the fuzzy reasoning methodology and its analysis, secondly, fuzzy reasoning logical foundation, and thirdly, the fuzzy reasoning applications. Different fuzzy reasoning methods are proposed, mainly basing upon three different ideas. The first idea is about the composition that leads towards the Zadeh Compositional Rule of Inference (CRI) method, along with its variants (Lee & Jung, 2012). The second idea is about similarity and analogy, and the third idea is about the interpolation analysis in fuzzy reasoning methodologies, which are concerned with the different properties of interests like the fuzzy rules interpretability. Moreover, the consistency of the latest fuzzy consequences with the existing premises and the continuity of the fuzzy implications concerning the fuzzy relations and premises. Furthermore, according to Wu et al. (2012), various implication operators, along with the convections, can be adopted in the fuzzy reasoning methods that include an important class of processes that are concerned with the suitability of the particular techniques of fuzzy reasoning for the domain-specific applications.

Based on the system structure model, the researcher Wu, et al. (2012) constructed the kinetic equation, incorporating the Newtonian mechanics and dynamics. After performing several different simulation experiments, the researcher got the best Q and R, along with the state feedback matrices. After that, the researcher designed the PD fuzzy controller at which the speed and position of the robot were the inputs and angle rate along with the angle of the robot was controlled through the PD controller (Romlay et al., 2019). The real-time control platform was developed for two wheels balancing robot capable of effectively controlling the robot after some parameter

debugging. The results of the experiments indicate that PD fuzzy control algorithm is successful in achieving the self-balancing control of the robot and is very useful in preventing the robot from falling.

Chapter 3: Theoretical Methodology

The concept and the working of the smart two wheels balancing robot are based on the inverted pendulum theory. To develop a capable and reliable control system for the robot, it is critical to understand the parameters that are within a robot system (Parcito, 2016). The presentation of those parameters is achieved through a mathematical model. The inverted pendulum theory is more traditionally regarded as the cart and pole theory, and even though the structure of the self-balancing robot does not compare directly with a cart and pole, similar principles are applied as both follow the phenomenon of an inverted pendulum. Within this model, wheels are represented by a cart, and the chassis of the robot serves as the pole (Prakash & Thomas, 2017). The equations are derived from Newton's second law of motion with consideration of corrections that gravitational acceleration is factually the positive value as the frictional force between the horizontal surface, and the wheel must be taken into the equation (Kankhunthod et al., 2019). The friction coefficients are not considered in the project as this robot is expected to traverse through various types of surfaces and terrains. If coefficients are considered for the implementation and design of the control systems, then the circuitry, power consumption, and additional sensors would have been needed for deriving these latest values while in operation (Kongratana et al., 2012). The resources, time, and effort required for the creation of this capability exceeds the benefits can be expected by including them.

Inherently, a self-balancing robot is unstable, and if it is not capable of self-balancing, it will roll around its wheels' rotational axis without any external control and will eventually fall. However, with the consideration of this inverted pendulum theory, the robot can retain an upright state if the motor driving occurs and the robot's wheels move in the same direction of the fall. These robots are based on a special electromechanical system as these robots are designed in such a way that they have to self-balance on the pair of wheels while standing tall. Highly critical in this robot system is the base of the robot on which it stands. If it is not stable or balanced, the robot will tend to fall from its vertical axis; otherwise, it will stand tall. That is why the IMU, which includes the gyroscope, accelerometer, and magnetometer is used for providing the PID controller information about the angular position of the robot system's base (Majczak & Wawrzynski, 2015). A simulation to verify the model and tune the PID gains is conducted before implementing the algorithm in the APM controller. Then, the algorithm of self-balancing is programmed and

integrated into the controller. The controller is used for driving the motors of the robot either in the anticlockwise or clockwise directions for balancing the robot through the pulse width modulation control signal. The robot must have the capability of working on any type of surface based on the two motors that are constructed with one wheel for each.

The most important thing to understand is that the self-balancing robot should balance on the pair of wheels that have the required amount of grip and provide a sufficient amount of friction to the tires. To maintain its vertical axis, the robot has to do two things. It should measure the inclination angle and control the robot's motors. It should either move forward or backward based on the direction of the inclination angle. According to the inverted pendulum concept, the robot's body has to maintain a 0° angle, and any deviation from this angle means that the robot is not in an upright position or is unbalanced. For the measurement of the angle of the robot's body, two sensors are used, i.e., a gyroscope and an accelerometer. The accelerometer mainly senses the dynamic or static forces of acceleration, providing information about the linear velocity, and the gyroscope measures the angular velocity of the robot (Warren et al.,2011). The outputs that are provided by the sensors are mainly fused with the use of the complementary filter. However, it is important to understand that the output from the sensors is taken as an input from the PID algorithm, and the output from the motors is taken as the output by the PID algorithm. The main reason for that is that the self-balancing robot works on the concept of action and reaction. The motors of the robot move in the act of self-balancing only when there is a tilt in the robot's body either in the backward or forward direction. The sensors of the robot measure the process output that is subtracted by the PID algorithm from the reference set-point value for producing the error. This error value is then fed to the PID, where this error is managed in three ways. After processing the error in the PID algorithm, a control signal is produced by the controller. The signal PID is fed in a process under control. The control signal mainly tries to move the process towards a pitch angle set-point in the perpendicular direction by driving the motors of the robot in such a manner that it ends up in the set-point angle.

This project generalises the effects of right and left wheels and integrates them at a single combination term wheel. Since it helps in the simplification of the calculation, both wheels in this robot work in unison to maintain stability (Azar et al., 2018). For the determination of the specific requirements of forces (torques) for every individual wheel, the value of the wheels can be halved

for getting a single wheel approximate value. The concept is considered effective as surfaces and terrains vary among the wheels in certain terrains (Jiang et al., 2016).

The primary purpose of adopting the inverted pendulum theory is to keep the wheels of this robot beneath the centre of mass of the chassis. If the robot tilts forward to maintain stability and balance, the wheels will move in the forward direction to return to beneath the mass of chassis (Yuan et al., 2016). According to the inverted pendulum theory, if balance is not maintained, this robot will fall over.

Before the hardware implementation, it is better to start with the simulation. The simulation is conducted with the Matlab / Simulink software to ensure that the system is controllable. The model or plant of the robot is needed so that it has to formulated to a state-space form, and it must be incorporated with a closed-loop PID controller. The PID controller has three gains, such as Kp, Ki, and Kd, that can be tuned using Matlab / Simulink according to the desired signal response. The following sections represent the mathematical model formulation and the controller design.

3.1 Mathematical Model Formulation

In this section, the mathematical model of the robot is derived until the state-space with matrix representation is obtained. The six-state variables used are denoted by x vector, which contain the robot linear displacement (x), robot linear velocity ($\dot{x}$), pitch angle displacement (ϕ), pitch angular velocity ($\dot{\phi}$), yaw angle displacement (ψ), and yaw angular velocity (ψ). The control input is the voltage applied to the DC motor of the left wheel ($V_{a,L}$) and right wheel ($V_{a,R}$). The mathematical model is divided into the following three parts.

3.1.1 DC Motor Model

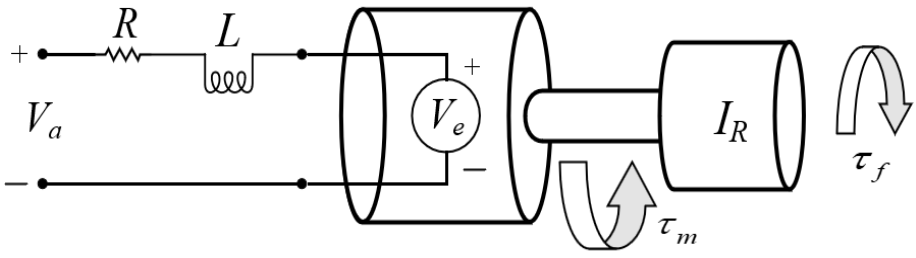


Figure 1: DC motor circuit [30]

By applying Kirchoff's law, the electric circuit equation can be obtained, which is represented in Equation 1:

$$V_a = Ri + L\, di/dt + V_e \quad \text{Equation 1}$$

V_a and V_e are the voltage applied to DC motor and the back ElectroMotive Force (EMF) voltage, respectively, while R is the resistance, and L is the inductance. The electrical current is denoted by i with τ_m is the motor torque without load, and τ_f is the friction torque. To simplify the model, the friction torque is neglected so that the total torque is represented in Equation 2 with k_m being the motor torque constant. i_{total} is the total current obtained from the voltage difference divided by the equivalent resistance, so that the total torque can be represented as in Equation 3:

$$\tau_{total} = \tau_m = i_{total} k_m \quad \text{Equation 2}$$

$$\tau_{total} = \frac{V_a - V_e}{R} k_m \quad \text{Equation 3}$$

The back EMF voltage is a linear function with motor angular velocity (ω), which is shown in Equation 4, with k_e, being the back EMF constant:

$$V_e = k_e \omega \quad \text{Equation 4}$$

By substituting Equation 4 to 3, Equation 5 and 6 are obtained with I_W being the moment inertia of the wheel:

$$\tau_{total} = I_W\, \dot{\omega} = \frac{V_a - V_e}{R} k_m \quad \text{Equation 5}$$

$$I_W\, \dot{\omega} = \frac{k_m}{R} V_a - \frac{k_m k_e}{R} \omega \quad \text{Equation 6}$$

3.1.2 Wheel Model

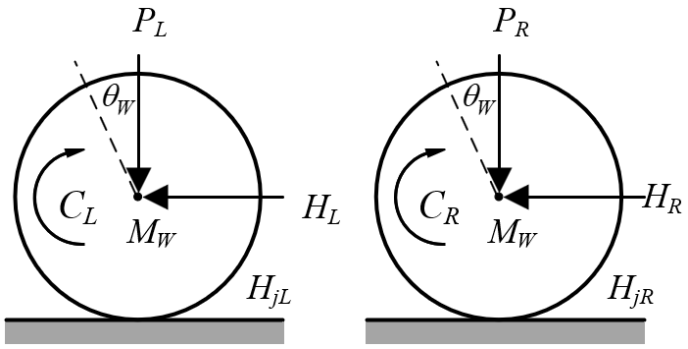


Figure 2: Robot's wheels free body diagram [30]

Using Newton's 2nd law of motion with Equation 6, the left and right wheel model equation is represented in Equation 7 and 8, considering that $x = \omega\, r$ with r is the radius of the wheel:

$$M_W \ddot{x} = \frac{k_m}{Rr} V_{a,L} - \frac{k_m k_e}{Rr^2} \dot{x} - \frac{I_W}{r^2} \ddot{x} - H_L \qquad \text{Equation 7}$$

$$M_W \ddot{x} = \frac{k_m}{Rr} V_{a,R} - \frac{k_m k_e}{Rr^2} \dot{x} - \frac{I_W}{r^2} \ddot{x} - H_R \qquad \text{Equation 8}$$

H_L and H_R are the frictional force of the left and right wheel, respectively. By adding Equation 7 and 8, results to:

$$2(M_W + \frac{I_W}{r^2})\ddot{x} = \frac{k_m}{Rr}(V_{a,L} + V_{a,R}) - \frac{2k_m k_e}{Rr^2} \dot{x} - (H_L + H_R) \qquad \text{Equation 9}$$

3.1.3 Pendulum/Body Model

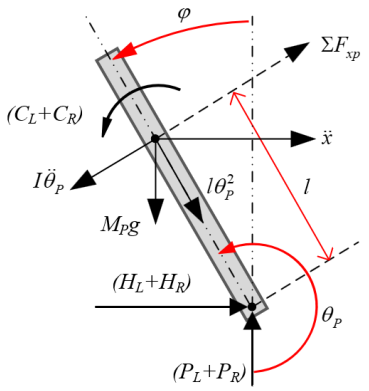


Figure 3: Robot's pendulum free body diagram [30]

The equations related to the angular and linear movement are represented in Equation 10 and 11:

$$(I_P + l^2 M_P)\ddot{\theta}_P - \frac{2\dot{k_m} k_e}{Rr}\dot{x} + \frac{k_m}{R}\left(V_{a,L} + V_{a,R}\right) + M_P g l \sin\theta_P = -M_P l \ddot{x}\cos\theta_P \quad \text{Equation 10}$$

$$\frac{k_m}{Rr}\left(V_{a,L} + V_{a,R}\right) = \left(2M_W + \frac{2I_W}{r^2} + M_P\right)\ddot{x} + \frac{2k_m k_e}{Rr}\dot{x} + M_P l\,\ddot{\theta}_P \cos\theta_P - M_P l \theta_P^2 \cos\theta_P \quad \text{Equation 11}$$

The yaw/heading state equation is represented in Equation 12, in which the derivation can be seen in (Asali et al., 2017).

$$\ddot{\psi} = \frac{k_m w}{Rr(I_\psi + \left(\frac{I_W}{r^2} + M_W\right)w^2}\left(V_{a,L} - V_{a,R}\right) \quad \text{Equation 12}$$

To obtain the linear state-space matrix, Equation 10 and 11 should be linearised by approximating $\cos\theta_P = \cos(\pi + \phi) \approx -1$, $\sin\theta_P = \sin(\pi + \phi) \approx -\phi$, and $\dot{\theta}_P^2 \approx 0$. After the linearisation process, Equation 10 and 11 become:

$$\ddot{\phi} = \frac{M_P l}{(I_P + l^2 M_P)}\ddot{x} + \frac{2k_m k_e}{Rr(I_P + l^2 M_P)}\dot{x} + \frac{M_P g l}{(I_P + l^2 M_P)}\phi - \frac{2k_m}{R(I_P + l^2 M_P)}\left(V_{a,L} + V_{a,R}\right) \quad \text{Equation 13}$$

$$\ddot{x} = \frac{M_P l}{\left(\frac{2I_W}{r^2} + M_P + 2M_W\right)}\ddot{\phi} - \frac{2k_m k_e}{Rr^2\left(\frac{2I_W}{r^2} + M_P + 2M_W\right)}\dot{x} + \frac{2k_m}{R\left(\frac{2I_W}{r^2} + M_P + 2M_W\right)}\left(V_{a,L} + V_{a,R}\right)$$

Equation 14

From equation 10, 11, 12, with $\beta = \frac{2I_W}{r^2} + M_P + 2M_W$, and $\alpha = I_P\beta + l^2 M_P\left(M_W + \frac{I_W}{r^2}\right)$. The state-space representation can be created, and it is shown in Equation 15 and 16 (Jamil et al., 2014).

$$\begin{bmatrix}\dot{x}\\ \ddot{x}\\ \dot{\phi}\\ \ddot{\phi}\\ \dot{\psi}\\ \ddot{\psi}\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\\ 0 & \frac{2k_m k_e(M_P lr - I_P - M_P l^2)}{Rr^2\alpha} & \frac{M_P g l^2}{\alpha} & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\\ 0 & \frac{2k_m k_e(r\beta - M_P l)}{Rr^2\alpha} & \frac{M_P g l\beta}{\alpha} & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \phi\\ \dot{\phi}\\ \psi\\ \dot{\psi}\end{bmatrix} + \begin{bmatrix}0 & 0\\ \frac{k_m(I_P - M_P lr + M_P l^2)}{Rr\alpha} & \frac{k_m(I_P - M_P lr + M_P l^2)}{Rr\alpha}\\ 0 & 0\\ \frac{k_m(-r\beta + M_P l)}{Rr\alpha} & \frac{k_m(-r\beta + M_P l)}{Rr\alpha}\\ 0 & 0\\ \frac{k_m w}{Rr(I_\psi + \left(\frac{I_W}{r^2} + M_W\right)w^2} & -\frac{k_m w}{Rr(I_\psi + \left(\frac{I_W}{r^2} + M_W\right)w^2}\end{bmatrix}\begin{bmatrix}V_{a,L}\\ V_{a,R}\end{bmatrix}$$

Equation 15

$$y = \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \phi\\ \dot{\phi}\\ \psi\\ \dot{\psi}\end{bmatrix}$$

Equation 16

3.2 PID Balancing Algorithm

Recently, a considerable amount of work has been done in the study of self-balancing robots. With consideration of the inverted pendulum theory, the self-balancing concept starts with balancing an inverted pendulum. In this project, the model of the robot is designed using the integral, proportional, and derivate terms of the PID controller. A PID loop is integrated into the software to provide the balance (Binugroho et al., 2016). The proportional term integrates the angle error of this robot and delivers that scaled value at motors for keeping the wheels rolling in the direction of the tilt. An integral term is used similarly, but it is the total of all angle errors recorded over time. The derivative term is also critical because, without it, the acceleration of the robot cannot be controlled (Pratama et al., 2016). The PID algorithm is considered as an adequate method of making the control system. In the basic algorithm, error signals received are considered as an input. The equation below is applied when an error signal is produced.

$$U(t) = Kp * e(t) + Kd * d/DT(e(t)) + Ki * integral(e(t)) \quad \text{Equation 17}$$

With the consideration of the above equation, integral and derivative versions of an error signal are calculated and multiplied with the respective constants and are added alongside the constant Kp and multiplied with ($e(t)$). The output of this calculation is then given to the actuator that makes this system run (Ali & Aphiratsakun, 2016). The PID algorithm is divided into three parts:

i. a proportional part that reduces time rise and reduces the error of steady-state.
ii. a derivative part that decreases the settling and overshoot time.
iii. an integral part that eliminates the error of steady-state and reduces the increased time (Martins & Nunes, 2017).

Tuning is also part of the algorithm because a good control system has low rise time, settling time, steady-state error, and peak overshoot (Dai et al., 2012). In this system, the variables that must be controlled are the wheel velocity, the pitch angle, and the yaw angle. The wheel velocity control is needed to control the robot to the desired velocity towards the goal destination. The pitch angle control is very crucial as the robot must always be upright position. The yaw angle control is important for the robot turning left or right according to the heading towards the goal destination. Therefore, the system needs three PID controllers with three feedbacks to minimise

three errors. The block diagram of the system is shown in Figure 4. The system symbols are defined as seen in Figure 5.

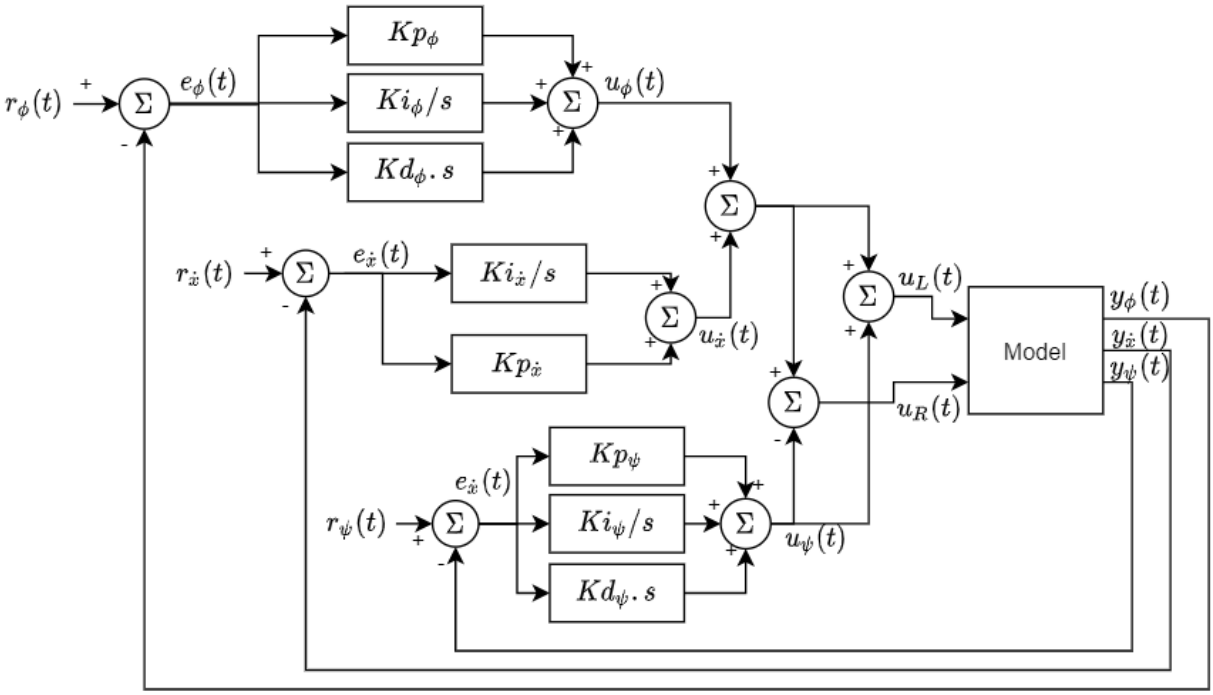


Figure 4: Control system block diagram

$y_\phi(t)$ = pitch measurement (from sensor IMU)
$y_{\dot{x}}(t)$ = wheel velocity measurement (from encoder)
$y_\psi(t)$ = yaw measurement (from sensor IMU)
$r_\phi(t)$ = desired pitch angle
$r_{\dot{x}}(t)$ = desired wheel velocity
$r_\psi(t)$ = desired yaw angle
$e_\phi(t)$ = error pitch angle
$e_{\dot{x}}(t)$ = error wheel velocity
$e_\psi(t)$ = error yaw angle

ϕ = pitch angle
$\dot{x}$ = wheel velocity
ψ = heading angle
$u_L(t)$ = control input to left wheel
$u_R(t)$ = control input to right wheel
$Kp_\phi, Ki_\phi, Kd_\phi$ = PID constants for pitch control
$Kp_{\dot{x}}, Ki_{\dot{x}}$ = PI constants for wheel velocity control
$Kp_\psi, Ki_\psi, Kd_\psi$ = PID constants for yaw control

Figure 5: System symbols definition

The controllers are used to optimally reduce the error ($e(t)$) between the measurement ($y(t)$) and the set-point reference ($r(t)$). The model needs two input control, $u_L(t)$ and $u_R(t)$, which are the voltage applied to the DC motor of the left wheel and right wheel, respectively. The system can simultaneously control the pitch angle and the velocity by adding the input control $u_\phi(t)$ and $u_{\dot{x}}(t)$ as they have the same direction. To control the robot to track the yaw angle set-point, intuitively, if the yaw error is positive, then the left wheel will go forward, and the right

wheel goes backward with the same magnitude. From the block diagram in Figure 4, system equations to calculate the input controls are represented from Equation 18 to 22.

$$u_{\phi}(t) = Kp_{\phi}e_{\phi}(t) + Ki_{\phi}\int_0^{\tau} e_{\phi}(\tau)\,d\tau + Kd_{\phi}\frac{de_{\phi}(t)}{dt}, e_{\phi} = r_{\phi} - y_{\phi} \quad \text{Equation 18}$$

$$u_{\dot{x}}(t) = Kp_{\dot{x}}e_{\dot{x}}(t) + Ki_{\dot{x}}\int_0^{\tau} e_{\dot{x}}(\tau)\,d\tau, e_{\dot{x}} = r_{\dot{x}} - y_{\dot{x}} \quad \text{Equation 19}$$

$$u_{\psi}(t) = Kp_{\psi}e_{\psi}(t) + Ki_{\psi}\int_0^{\tau} e_{\psi}(\tau)\,d\tau + Kd_{\psi}\frac{de_{\psi}(t)}{dt}, e_{\psi} = r_{\psi} - y_{\psi} \quad \text{Equation 20}$$

$$u_L(t) = u_{\phi}(t) + u_{\dot{x}}(t) - u_{\psi}(t) \quad \text{Equation 21}$$

$$u_R(t) = u_{\phi}(t) + u_{\dot{x}}(t) + u_{\psi}(t) \quad \text{Equation 22}$$

The PID controller is used to control the pitch and yaw angles, while only the Proportional Integral (PI) controller is used to control the velocity of the robot. The most crucial constant is the constant related to control the pitch angle because it sets the robot to balance. Kp_{ϕ} is the proportional constant for pitch angle error, which takes the error of the angle and sends it to the motors as a scaled value. Kd_{ϕ} is the derivative constant for pitch angle for controlling the rate of error. In theory, the derivative constant is used for compensating overshoot and improve stability. Ki_{ϕ} helped to cancel out centre of gravity issues.

3.3 Making the Robot Move

The primary robot leans in the direction of the movement. This is workable for a short period, but it falls over after constant acceleration, as shown in Figure 6 (A). If the robot tries righting its position, the forward motion is stopped. Instead, the robot needs to move in the forward direction while it is rolling vertically, as shown in Figure 6 (B). To make the robot move, the wheel velocity needs to be controlled by minimising the error between the desired wheel velocity and the measurement, as shown in Figure 4. The desired wheel velocity is defined by the distance between the current robot position and the destination position. However, it is constrained to the pitch angle, which must always be zero.

To achieve this goal, the wheels of the robot must rotate at an appropriate speed, and sufficient power must be left to keep the robot upright and balanced. Then, the velocity of the wheels is taken and is fed forward at the desired speed (Ruan & Li, 2014). It provides the robot with the ability to correct any rapid changes in an angle like an external force trying to push the robot over. The algorithm of this robot is developed by the observation of how people balance themselves when they are pushed, especially while standing straight (Chhotary et al., 2016). For example, a runner in a sprint race keeps running forward if unbalanced to keep his balance and prevent himself from falling. The two extra inputs are added alongside the algorithm of balancing, and the results are sent towards the motors, enabling the robot to gain stability, and move for a longer distance. This algorithm also considers the inverted pendulum theory as well as the pole and cart theory (Su et al., 2016). This method aims at ensuring that the robot maintains its balance while moving from one point to another. To prevent the robot from falling, it is programmed in such a way that its wheels move in the direction of the fall. For example, if the robot is falling forward, the wheels move in the forward direction to maintain balance. Alternatively, if the robot is falling backward, the wheels move backward to maintain the balance of the robot. There are two navigation modes; automatic and manual. In automatic mode, the robot automatically follows the waypoints based on GPS while self-balancing with zero pitch angle set-point without any RC interruption. In manual mode, the RC transmitter and receiver are used for balancing the robot when it is in motion.

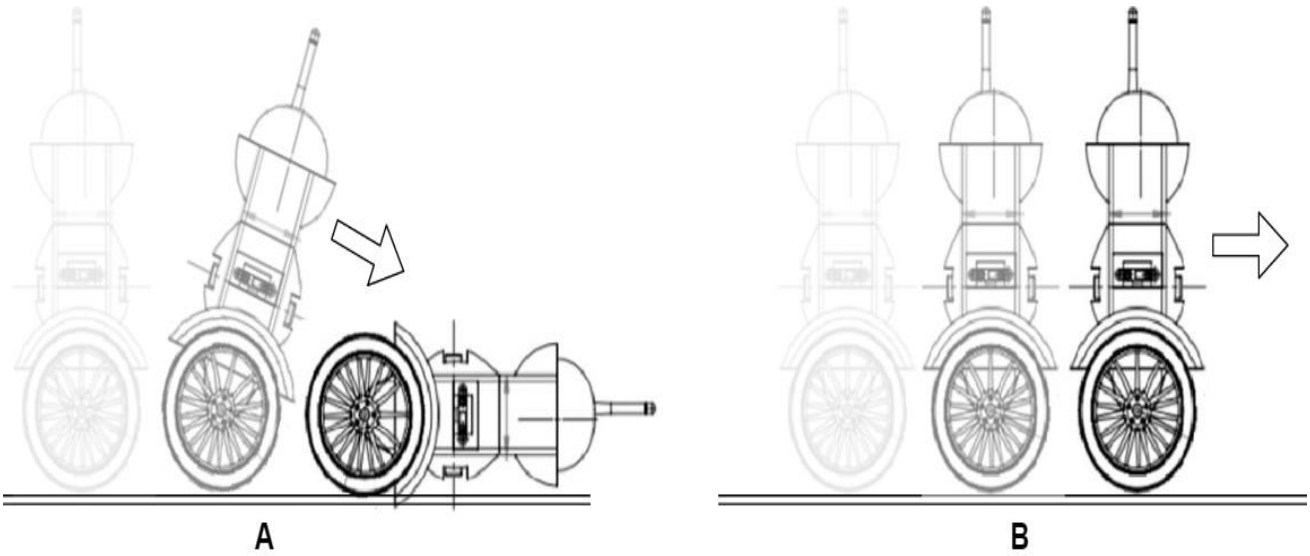


Figure 6: The smart two wheels balancing robot movement

3.4 Navigating the Robot

An important goal in this project is to navigate the robot with the help of a GPS, compass, and wheel encoders using Mission Planner software. To navigate this robot, it is essential to know precisely where the robot is and where it is going, this means that the current location of the robot and the destination of the robot has been known to ensure proper working of this system. GPS is workable, but its accuracy is limited to a few meters. To ensure that this system works as expected, far higher accuracy is required approximately down to the centimetre scale (Rahman et al., 2018). Wheel encoders are used in this robot as they allow accuracy in millimetres and are perfect for the GPS. In most similar systems, encoders are mostly located directly behind each motor. The primary function of each wheel encoder is counting the number of times the motor has rotated right or left. This system is very effective in robotics because it can be used in calculating the distance a robot has covered, and its speed (Wang et al., 2016). Each wheel encoder consists of two parts, the Hall Effect Sensor (that measures the strength of the magnetic field), and the Ring Magnet (that looks like a metal washer attached with the motor shaft). When the motor of the robot rotates the wheels, it also rotates the ring magnet. The Hall Effect Sensor, which is positioned near the ring, detects the changes in the magnetic field as the ring rotates. This is how the sensors of the robot count how many times the motor has rotated to calculate the speed and distance. The motors that are sold by Pololu are integrated with 48 CPR quadrature encoder on the motor shaft, and they can provide up to 1632.67 counts per revolution, which mainly work as the bicycle computers, i.e., having little magnets that rotate past sensors, providing data regarding the bicycle wheels speed (Acevedo & Alejo, 2014). Similar wheel encoders will be used in this robot, and they will provide accurate information regarding the speed of the wheels.

Furthermore, the Arduino Pro Mini will be used to read the pulses per second sent by encoders. The data is relayed towards the APM board through the I2C interface. As wheels' diameter is measured, the robot will know exactly how much and how fast it has moved (Pratama et al., 2015). The robot will know its heading angle and will be able to plot its precise position in the 2D space (Han et al., 2015). When available, the GPS of the robot is used for gently nudging this solution overtime for ensuring that the wheel spillage, rough terrain, and other issues do not emerge and move the robot off its course. The GPS value will correct the pose estimation calculated by the wheel encoder if there is a significant distance error by using a certain threshold (Hoang et al.,

2014). If, in any case, the robot becomes stuck when in motion, the APM will know when the wheels stop. The robot will try to reverse its direction and will keep trying again if it is not maintaining its balance (Wardoyo et al., 2015). In this part of the project, Mission Planner software is used to control the robot for autonomous navigation. Figure 7 shows the robot's location (Adelaide, Ascot Park) and data while controlling in Mission Planner.

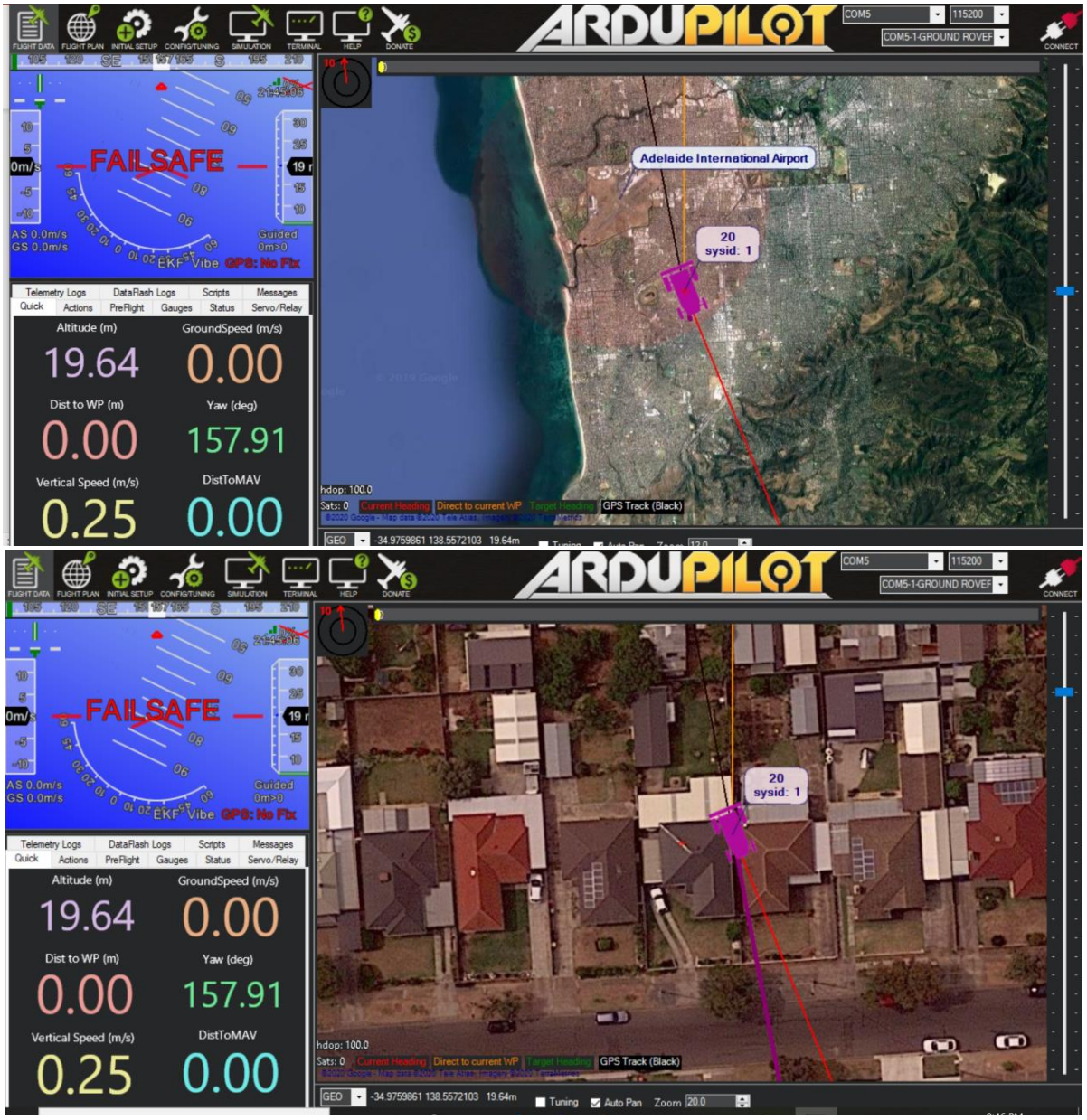


Figure 7: Robot in Mission Planner

Chapter 4: Results

4.1 Structure of the Robot

After completion of the last design concepts of the robot, the primary circuit for the robot was developed. Figure 8 illustrates the circuit diagram of the robot, including all component connections. The developed PID algorithm was integrated into the system, robot balancing, making the robot move, i.e., move while self-balancing manually with the RC transmitter and receiver or autonomously using GPS and other on-board sensors to reach all of the desired waypoints. Additionally, all components of the project are explained in depth in the discussion chapter of this report.

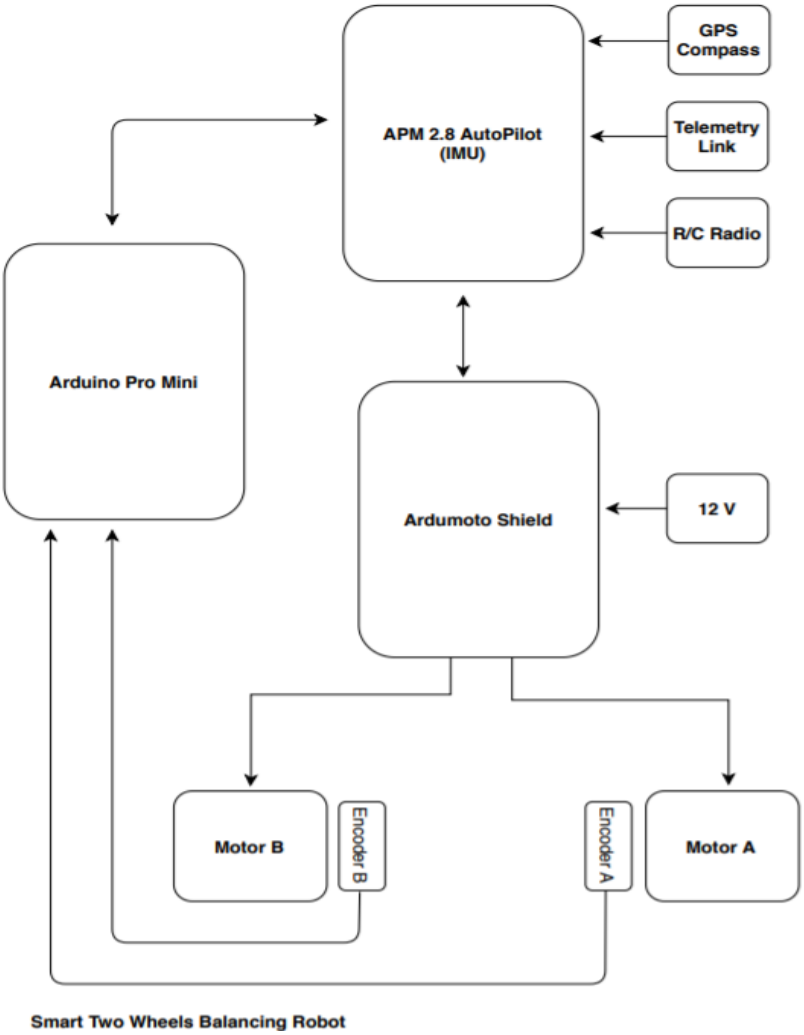


Figure 8: Circuit diagram of the robot

Figure 9 illustrates the rough design of the final robot structure. The electronic board can be inserted inside the main body along with the motors, wheels, and sensors. The RC receiver and telemetry radio lie outside the robot's body. The body is constructed in such a manner that it has the capability of fully integrating the board and allows smooth connection among the three main components of the robot, i.e., board, motors, and wheels.

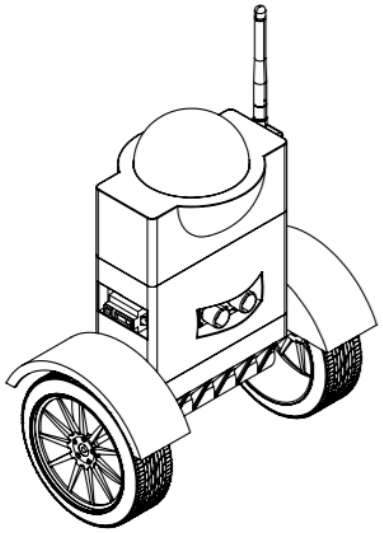


Figure 9: Robot structure

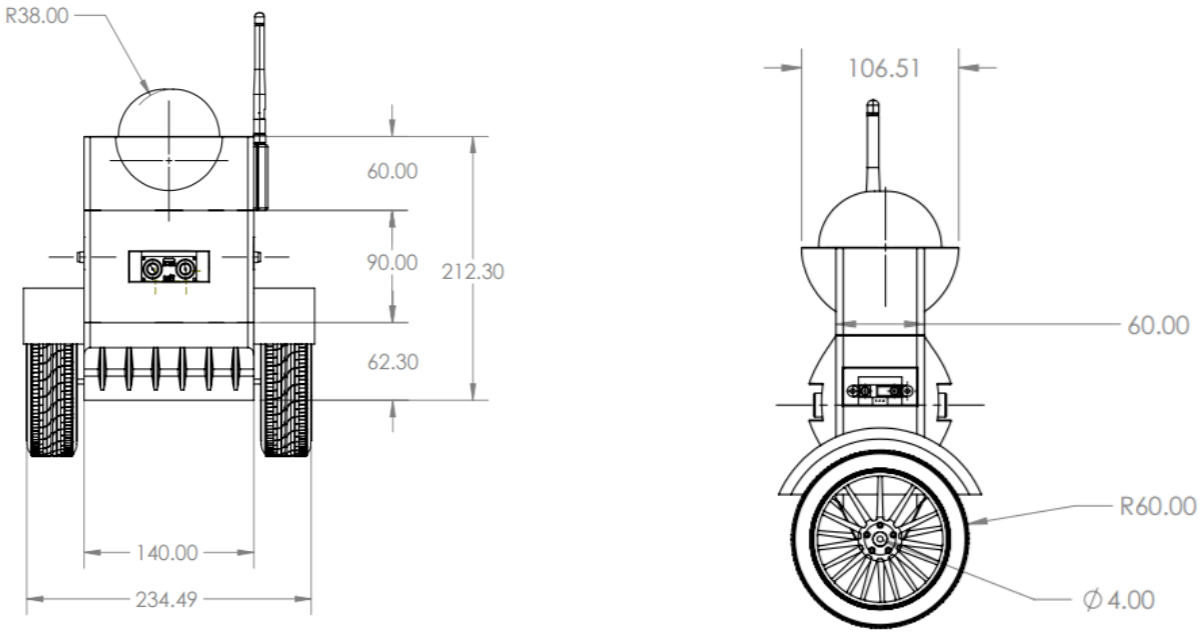


Figure 10: Robot dimensions

Figure 10 shows all the dimensions of the robot's body from top to bottom. Taking the dimensions of the robot before going ahead with the 3D modelling and design of the robot's body was essential because it helped to ensure that each component of the robot fits perfectly and that the robot's body is capable of integrating the board easily. After taking the accurate dimensions required for the robot's body, 3D physical models were printed. Figures 11, 12, and 13 represent the 3D upper, middle, and lower parts of the robot.

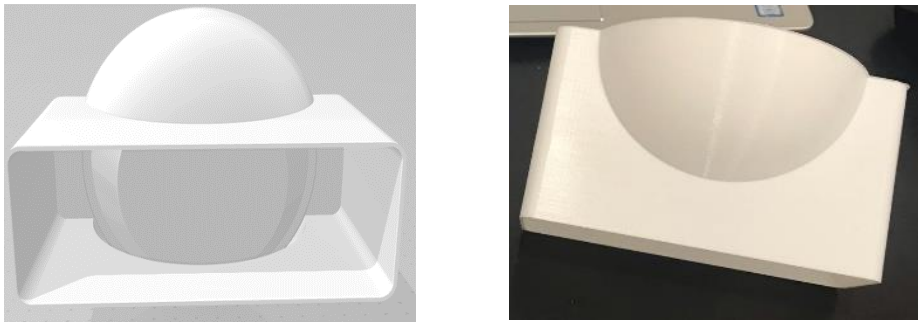


Figure 11: 3D Upper part model

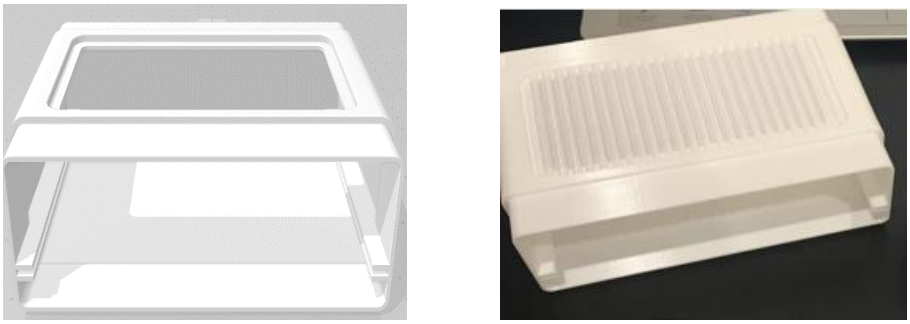


Figure 12: 3D Middle part model

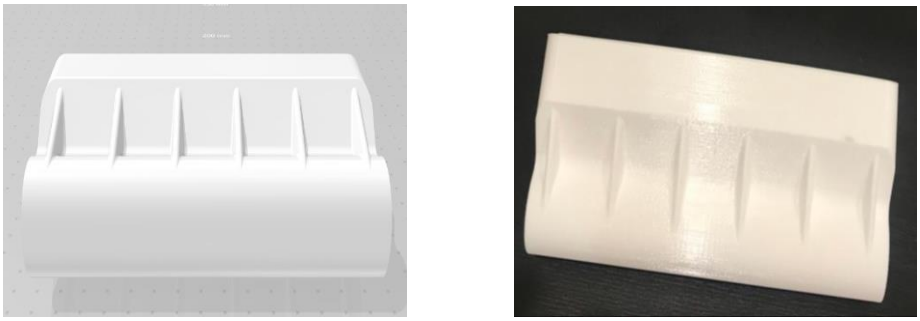


Figure 13: 3D Lower part model

After printing the 3D parts of the robot, it can be seen that the final physical models resemble the designs closely. Each part of the robot is designed to fit perfectly with the other parts. After the insertion of the boards along with the motors in the robot, the body is closed, allowing only space for the wheels to connect with the motors. The design of the robot is kept simple to assure minimal complications in the integration and connection of all the components of the robot with the main body. The reason for using plastic instead of any other material was to ensure that the material of the body helps the robot in balancing and self-controlling because the material is light and efficient. Plastic is much lighter than other materials that are usually used for building robot body parts such as steel and aluminium. This ensures easy movement and higher speeds of the robot.

4.2 Matlab / Simulink Simulation

Before experimenting on the robot, the simulation using Matlab and Simulink to verify the model and tune the PID is mandatory. The following are the simulation setup and the analysis of the simulation results.

4.2.1 Simulation Setup

According to the controller design in Figure 4, the block diagram in Simulink software is created, as seen in Figure 14. The system needs to control three of six state variables to a certain desired value, which are the wheel velocity, pitch angle, and yaw angle. In Simulink, the signal response of all state variables can be shown, but it must modify the matrix C to get all state variables as the outputs, i.e., C equal to 6x6 identity matrix. The saturation block is used to limit the input control because the voltage that can be applied to the DC motor is constrained. The linearised mathematical model is scripted in a Matlab function represented in Appendix A with the parameters according to the mechanical and electronic design in Table 1. The voltage input to the left and right DC motor is constrained to -12 to 12V.

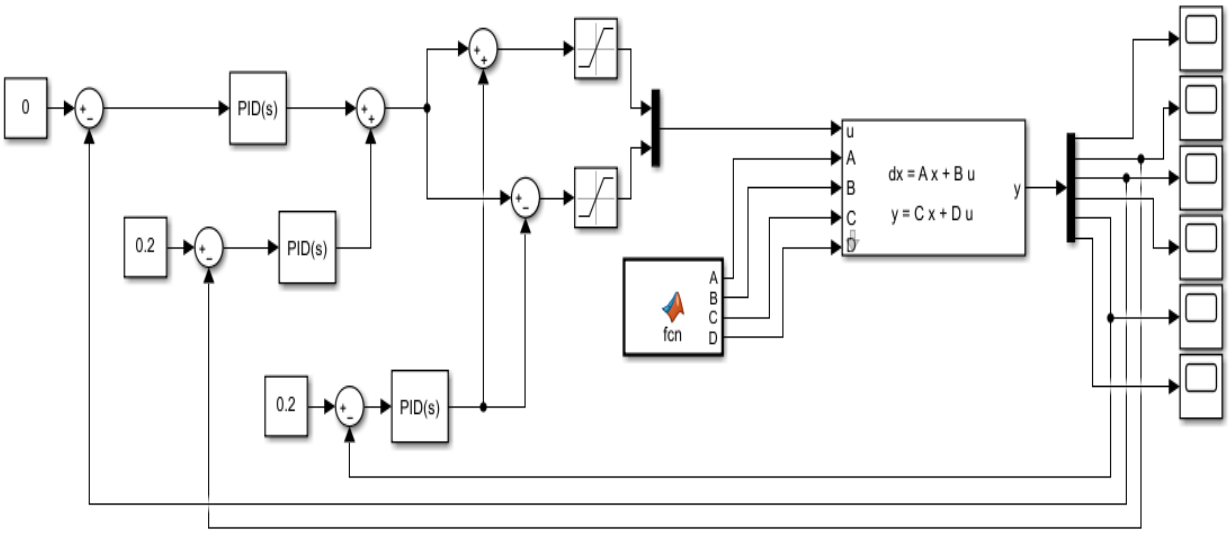


Figure 14: Control block diagram in Simulink

Parameter Name	Value	Unit
Gravity constant ($\boldsymbol{g}$)	9.81	m/s^2
The radius of a wheel ($\boldsymbol{r}$)	0.06	m
Mass of wheel ($\boldsymbol{M_W}$)	0.15	m
Mass of body ($\boldsymbol{M_P}$)	0.9	m
Height of body ($\boldsymbol{h}$)	0.22	m
Length to body's centre of gravity ($\boldsymbol{l}$)	0.165	m
Depth of body ($\boldsymbol{d}$)	0.05	m
Wheel's moment of inertia ($\boldsymbol{I_W}$)	2.7×10^{-4}	$kg\ m^2$
Body's moment of inertia in pitch direction ($\boldsymbol{I_P}$)	51×10^{-4}	$kg\ m^2$
Body's moment of inertia in yaw direction ($\boldsymbol{I_\psi}$)	17×10^{-4}	$kg\ m^2$
DC motor torque constant ($\boldsymbol{k_m}$)	0.1284	Nm/A
DC motor back EMF constant ($\boldsymbol{k_e}$)	0.3472	$V/(rad/s)$
DC motor equivalent resistance ($\boldsymbol{R}$)	2.7273	Ω

Table 1: Parameters according to the system design

Equation 23 to 25 are the formulas to calculate the moment inertia of the wheel, moment inertia of the robot's body in pitch, and yaw direction, respectively.

$$I_W = \frac{1}{2} M_W r^2 \quad \text{Equation 23}$$

$$I_P = \frac{1}{12}(h^2 + w^2) \quad \text{Equation 24}$$

$$I_\psi = \frac{1}{12}(d^2 + w^2) \quad \text{Equation 25}$$

4.2.2 Simulation Result and Analysis

In this section, some simulation scenarios are represented and explained as the following:

I. Testing the robot self-balancing

In the simulation, the signal response according to the robot's balancing ability is illustrated. The optimal PID constant is tuned, so it produces Kp_ϕ=1201, Ki_ϕ=10872, and Kd_ϕ=31. The robot is tested with an initial pitch angle of 0.1 rad, and the pitch angle reference is 0 rad, which is upright. Moreover, results are shown in Figures 15 and 16.

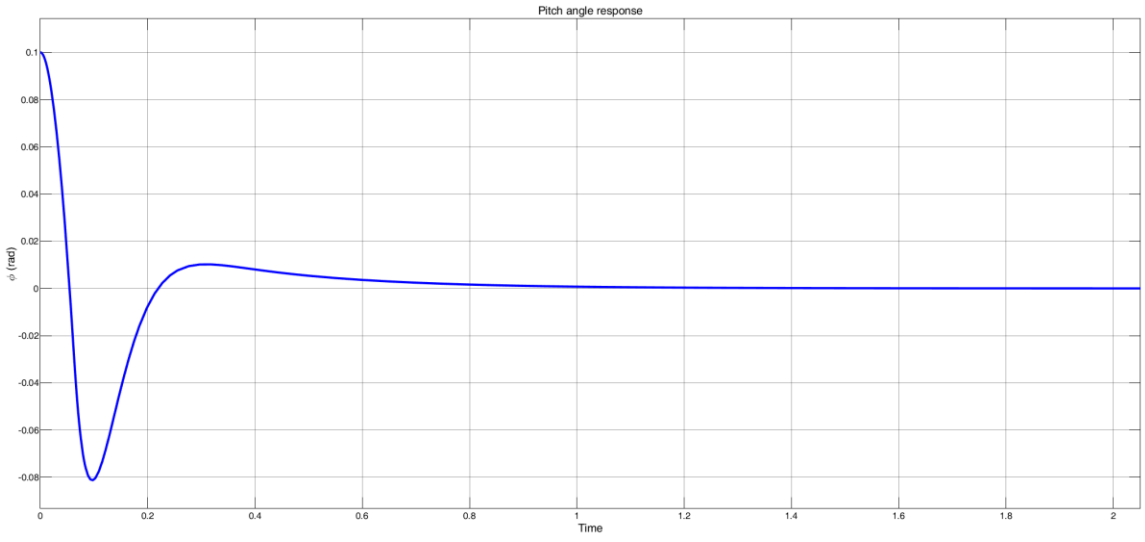


Figure 15: Signal response of pitch angle with an initial pitch angle 0.1 rad &pitch angle reference 0 rad

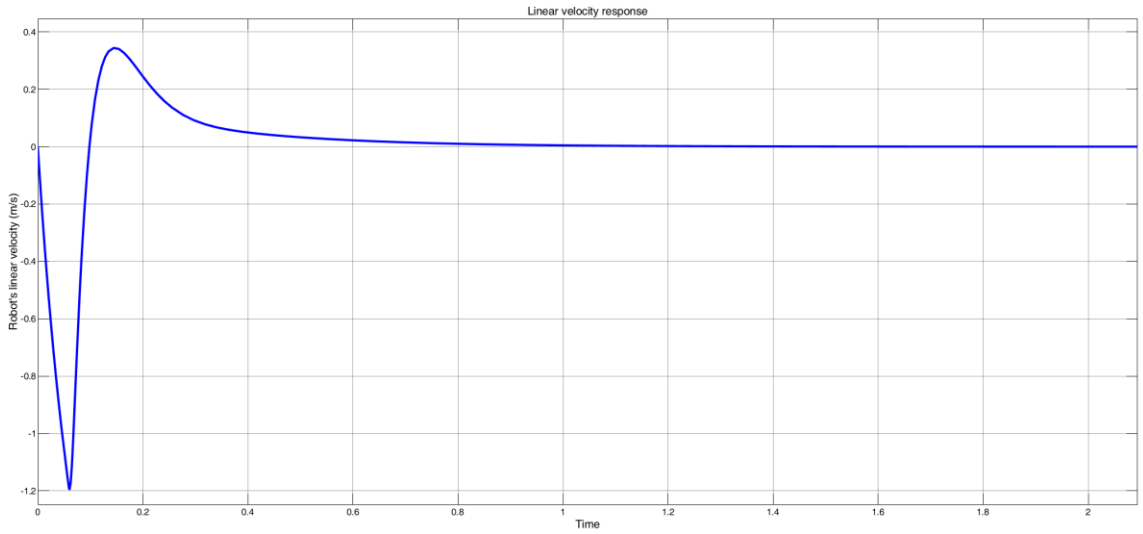


Figure 16: Signal response of the robot's velocity with an initial pitch angle 0.1 rad & pitch angle 0 rad

From the graph in Figures 15 and 16, if there is a disturbance that makes the robot pitch angle change to 0.1 rad, then it will be recovered for 0.8 secs with an overshoot of up to 80%. Consequently, to maintain balance, the robot needs to go backward for 0.1 sec and go forward for around 0.05 secs until the robot doesn't move.

II. Testing the robot's velocity control

The velocity control is needed when the robot is intended to go to a destination point while the pitch angle must always be upright. Therefore, in this scenario, the robot's velocity set-point is set to 0.1 m/s, while the pitch angle set-point is set to zero. The results are illustrated in Figures 17 and 18, where the system can track the desired velocity to 0.1 m/s, but it initially goes reverse up to 0.07 m/s, which is 70% until it reaches the steady-state for around one second. Another consequence is the pitch angle is slightly change around -0.01 rad, which is shown in Figure 18. The used PI constants are $Kp_{\dot{x}}$ = 45 and $Ki_{\dot{x}}$ = 1236.

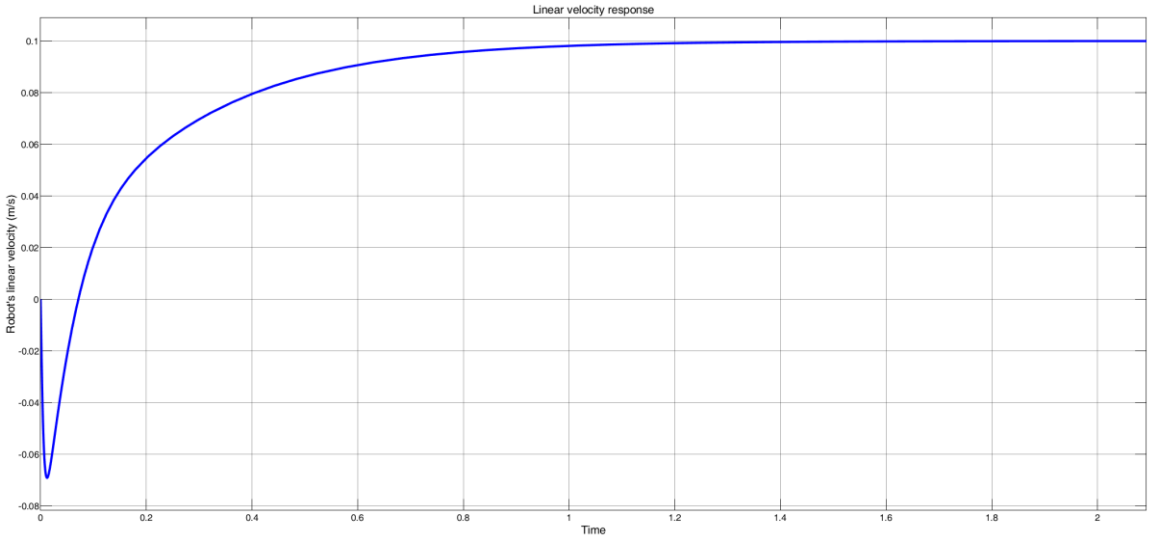


Figure 17: Signal response of the robot's velocity with the initial pitch angle and linear velocity is zero, and the robot's velocity reference is 0.1 m/s

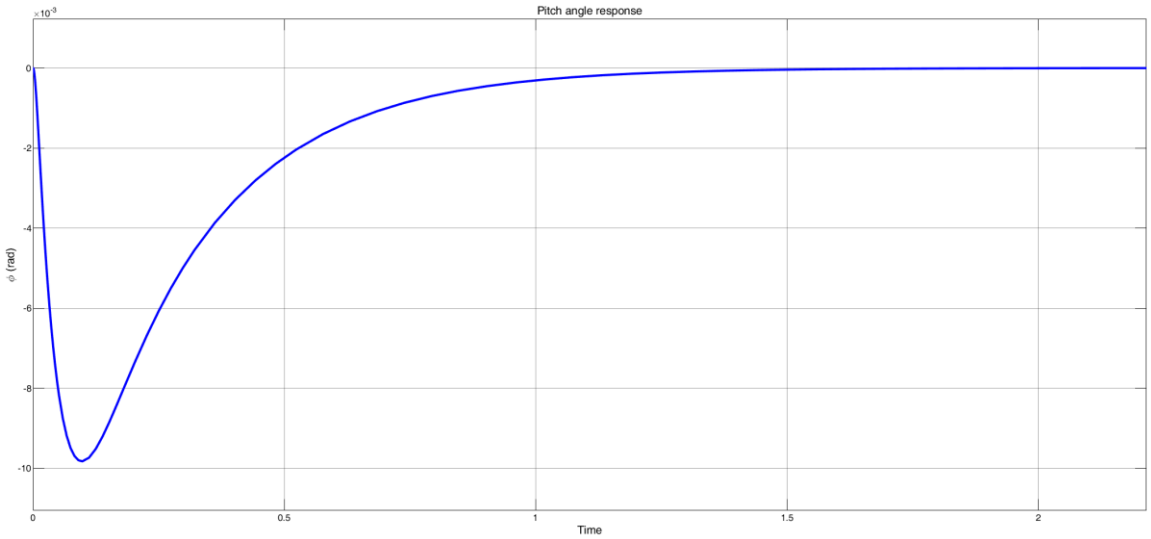


Figure 18: Signal response of pitch angle with an initial pitch angle and linear velocity at zero, and the robot's velocity reference is 0.1 m/s

III. Testing the robot's yaw control

The yaw angle is easier to control; i.e., the PID constant is easy to tune because the yaw direction doesn't affect the balancing function. The yaw angle response with 1 rad set-point is illustrated in Figures 19 and 20.

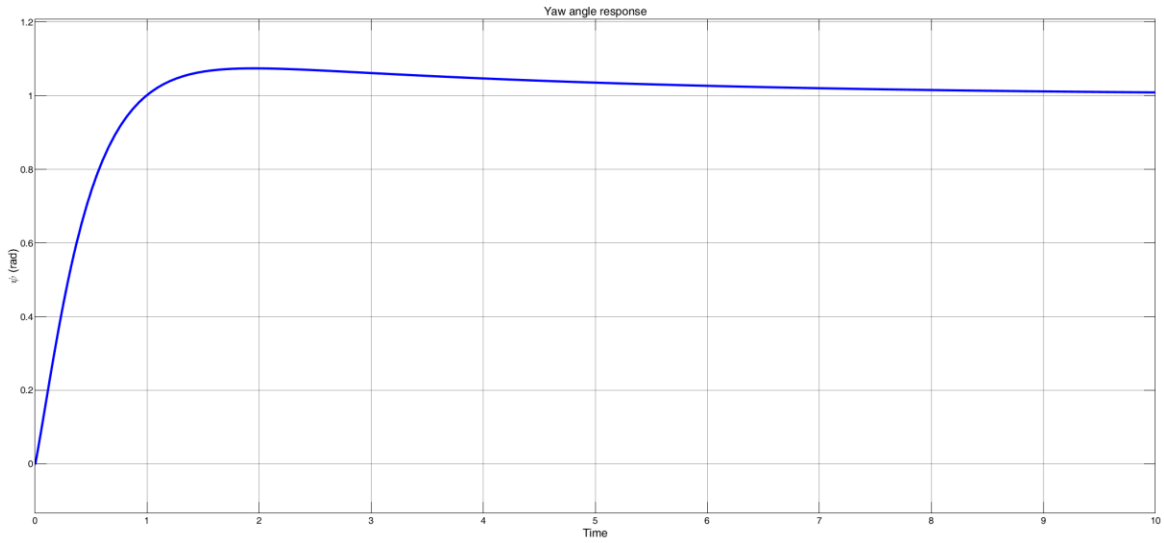


Figure 19: Signal response of yaw angle with 1 rad set-point (Kp_ψ=0.57)

The results in Figure 19 are obtained with PID constants: Kp_ψ = 0.57, Ki_ψ = 0.14, and Kd_ψ = 0.27. With the low value of Kp and Ki, the response is slow, but the overshoot is low. The result with greater Kp value is 1.57, which gives more overshoot but faster response as shown in Figure 20. Both increasing and decreasing the Kd_ψ is not an option because increasing Kd_ψ will reduce the overshoot, but increase the steady-state error while decreasing Kd_ψ will reduce stability, i.e., increase the overshoot.

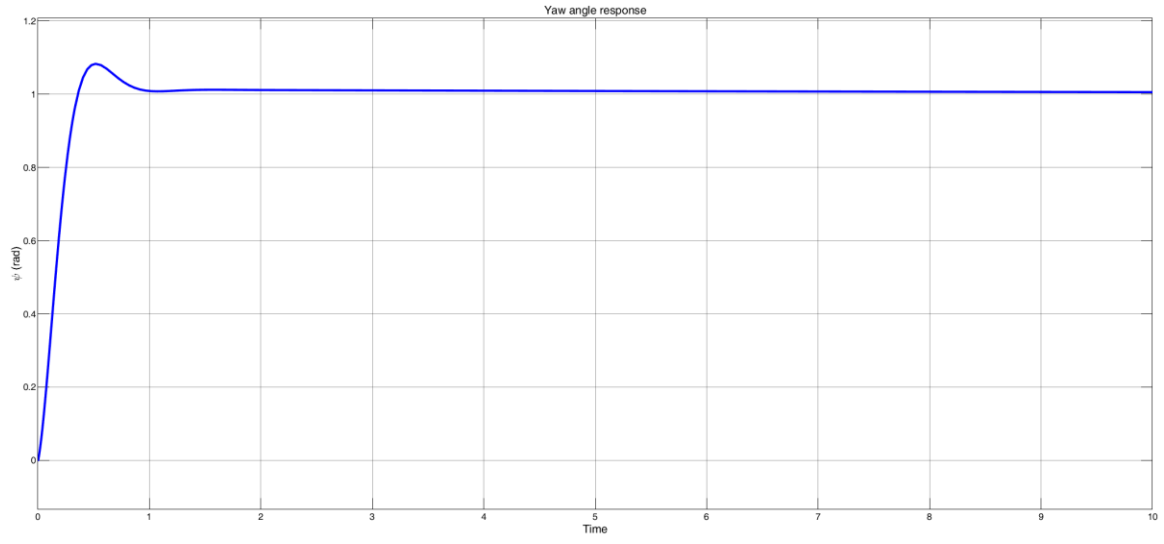


Figure 20: Signal response of yaw angle with 1 rad set-point (Kp_ψ=1.57)

From the concept of the two wheels balancing model, it is better if the robot has a longer length to the body's centre of gravity and larger wheel torque constant. However, from the above results, the mechanical and electrical parts of the robot are verified, and the robot is good enough to be tested in the real environment as it can balance by controlling the pitch angle, and it can move forward or backward by controlling the wheel velocity and can turn left or right by controlling the yaw angle. The tuned PID constants for pitch, wheel velocity, and yaw control can be used as the parameter constants into the APM firmware. The result is also reasonable, i.e., if the robot wants to go forward, it will go backward to do a reverse moment because it must always balance.

The simulation results depicted with the signal response are good, but it is simulated with an approximated mathematical model neglecting some aspects like wheel-slip, friction, and wind disturbance. In this project, the PID controller is used because it can be computed faster than the complex non-linear or neural network/AI-based controller, which needs sophisticated CPU/GPU processing. By using the PID controller, the model/plant also needs to be linearised. Therefore, there will be a slight difference response between the simulation and the implementation in the real environment.

4.3 Practical Results

After the structure and simulation of the robot have been done successfully, all electronics components including APM, Arduino Pro Mini 328 board, telemetry radio, R/C transmitter and receiver, GPS model, Ardumoto motor driver shield, DC motors with encoders, and the wheels are connected on one board according to the schematic, as seen in the Appendix F. The electronic board is inserted inside the main body of the robot, as seen in Figure 21. Also, Figure 22 shows the final assembly of the robot.

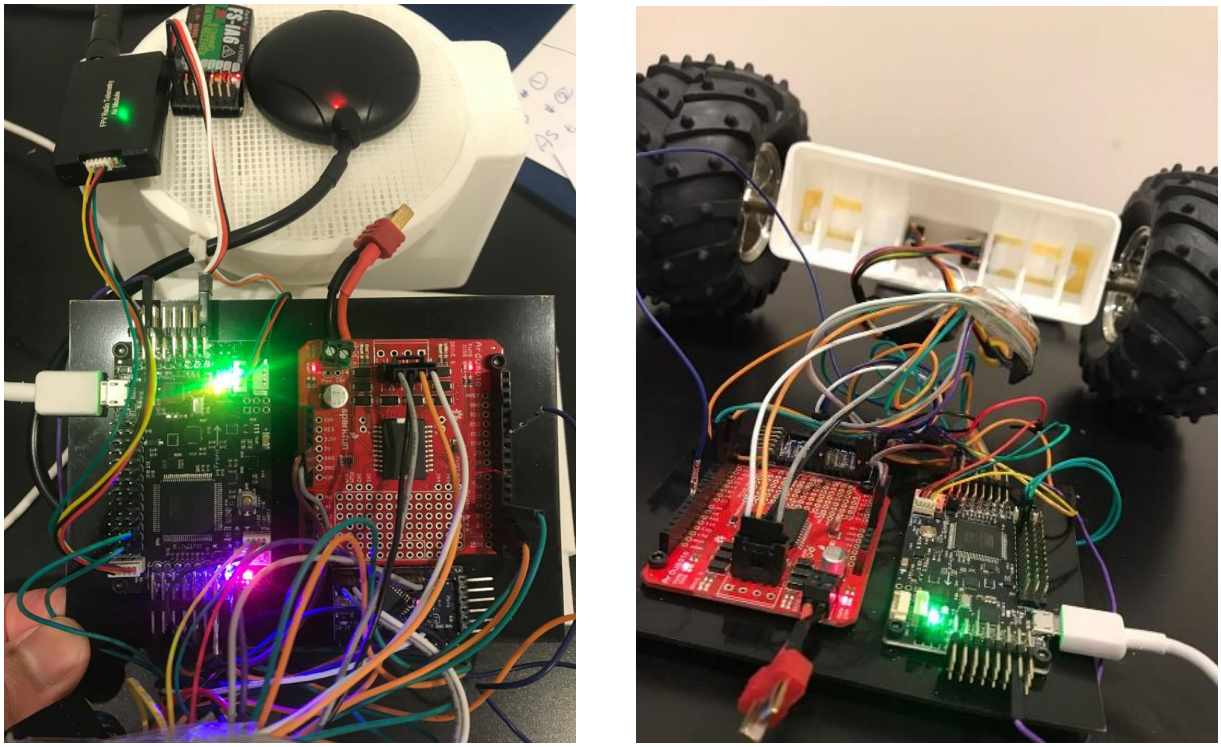


Figure 21: Electronics board connection of the robot

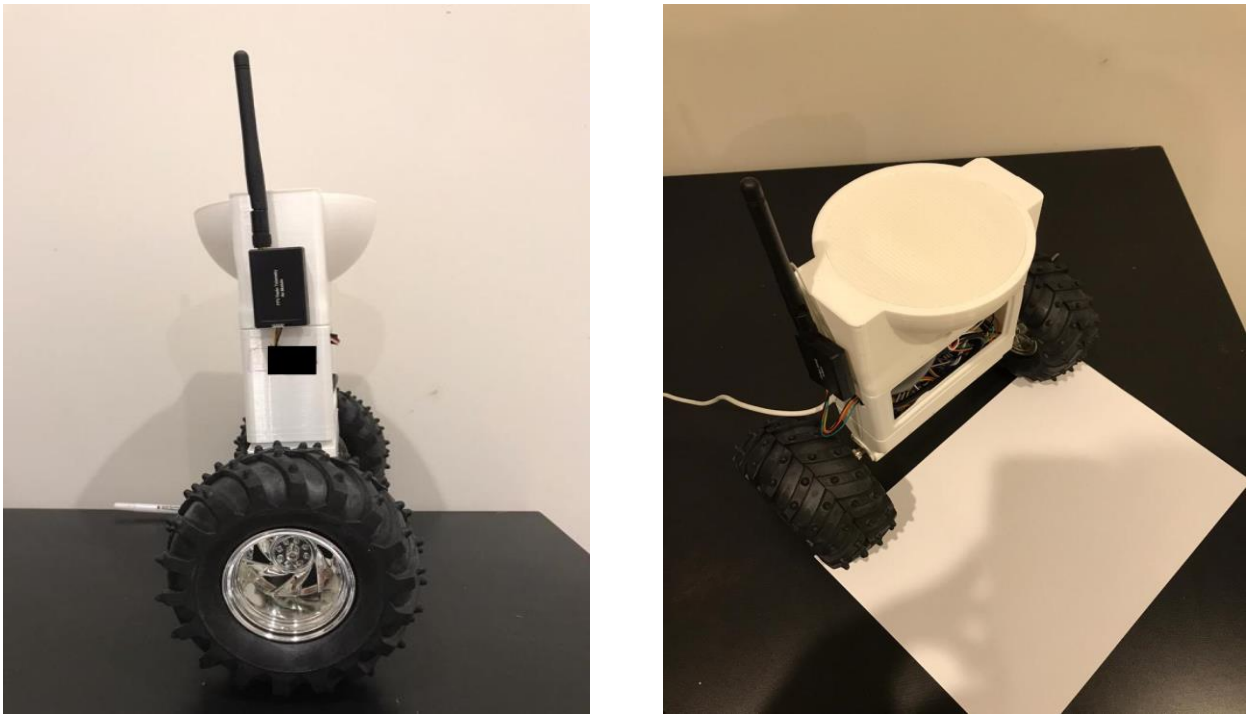


Figure 22: Final assembly of the robot

After completing the designing of the robot and testing all the stages, the robot has the capability of balancing the load of the robot chassis, i.e., the body of the robot (load) through its wheels instead of dragging the weight around like a regular robot after the body falls. This robot is programmed in such a way that the wheels ensure self-balancing by moving in the direction of the robot's tilt. For example, if the body of the robot is tilting forward, then the wheels move forward to balance it. If the body of the robot tilts backward, then the wheels move backward to ensure that the robot self-balances. It is essential to point out the fact that the wheels of this robot only move in two directions, i.e., backward and forward. After the design and assembly phases of the robot, it was tested multiple times across various terrains and obstacles to check whether it has the capability of not only moving from one place to another and self-balance while moving. The APM system provided the ability to turn the wheels of the robot forward and backward in a fully autonomous way and allowed the robot to perform programmed GPS tasks or missions with waypoints provided electronically through the controller using Mission Planner program, as seen in Figure 23.

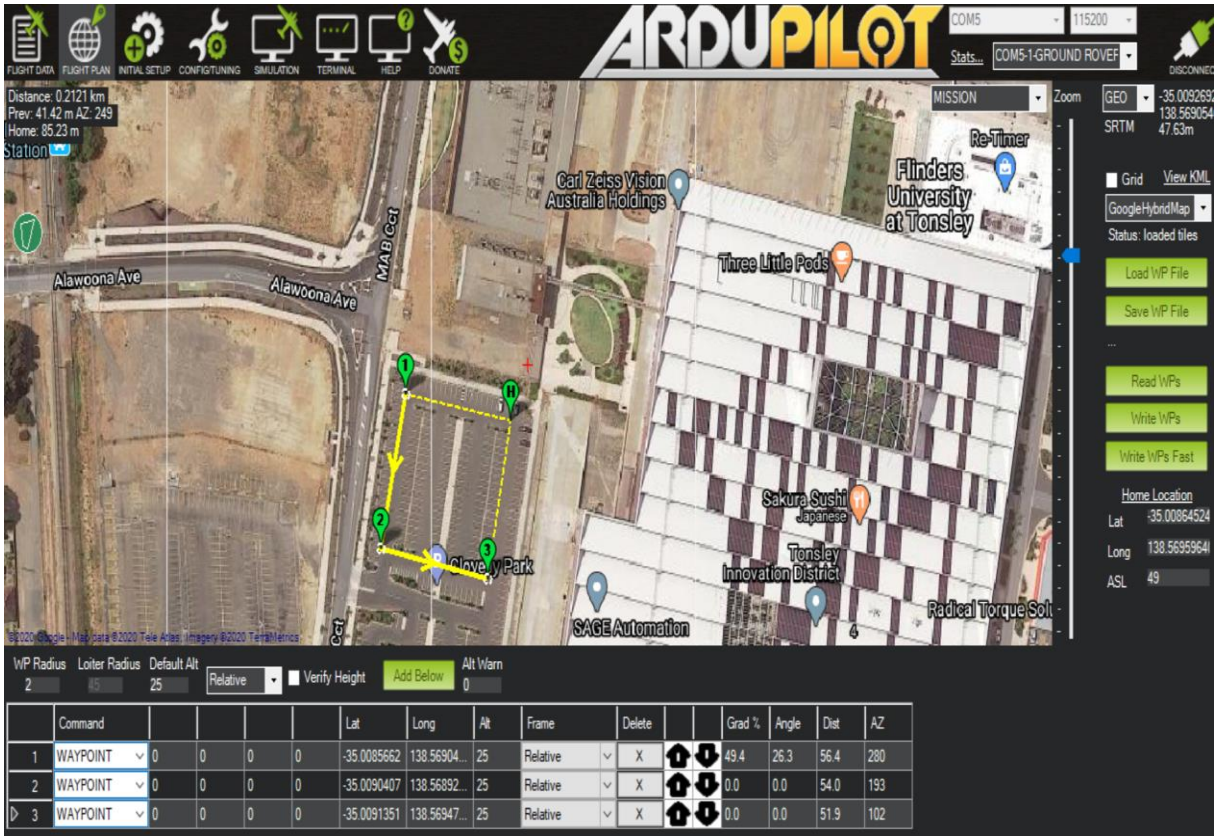


Figure 23: Robot's plan for autonomous navigation (waypoints) at Flinders University in Tonsley

The on-board compass on the APM was not very effective for this robot as the placement of the compass was very close to the motors and power sources of the robot due to its smaller size. The magnetic interference could not be avoided. However, as this controller is designed for its utilisation with the uBlox GPS with a compass, this unit was mounted on the robot to allow the use of GPS. As the GPS is a requirement for the APM to achieve full autonomy, the use of GPS with compass ensured proper working of the APM, allowing full autonomous navigation of the robot (Ateov et al., 2017).

The first goal of this project of making the robot balanced using the PID algorithm was achieved after the PID algorithms for the APM controller was developed using ArduPilot-Arduino as this algorithm was based on the inverted pendulum theory that was previously designed using Matlab/Simulink simulation. This algorithm has programmed the robot to move in the direction of tilt. With the help of the IMU sensors, the APM detects the movement of the robot body or tilt in the forward or backward directions. After detecting the motion of the tilt, the APM is programmed by the PID algorithm, as shown in Appendices B and C, to autonomously move the motor and the robot's wheels in the tilt's direction. This PID algorithm ensures the achievement of the first goal of this project of a self-balancing robot by automatically moving the robot in the direction of the tilt to ensure the maintenance of the robot's balance in motion. The second and third goals of this project were to make the robot move while self-balancing by using the RC transmitter and receiver and making it navigate using GPS, compass, and wheel encoders using Mission Planner program. The code for the Arduino Pro Mini microcontroller is shown in Appendix E. Both goals are successfully achieved by programming the Arduino Pro Mini microcontroller to control the wheel encoders, which allows the robot to navigate precisely. Finally, the robot is fully controlled using the Mission Planner program for autonomous navigation.

Chapter 5: Discussion

5.1 Concept of Self-Balancing Robot

The smart two wheels balancing robot, as mentioned earlier in this report, is following the inverted pendulum theory. Control theory requires keeping some of the variables steady, including the robot's position in this case, for which a special type of controller known as PID is required. The parameters particularly have gains known as Kp, Kd, and Ki (Kim & Kwon, 2015). The requirement of the PID was essential as it gives correction among the desired values (input) and actual values (output). The difference between the output and input values is known as error. The reason as to why the PID controller and algorithms are used is that their main purpose is to reduce the error between input and output values to the minimum by the continual adjustment of the output (Kim et al., 2011). In the self-balancing robot, the input (desired tilt in terms of degrees) is settled by the software. The APM reads the robot's current tilt and gives it to the PID algorithm that performs the required calculations for controlling the motor of the robot and ensuring that through continued movement, the robot maintains its upright position. However, it is necessary for the PID that the gains in Kp, Ki, and Kd are tuned to the optimal values. Most engineers use software like Matlab for computing values automatically. However, the autonomous capabilities of APM allow us to develop a PID algorithm that ensures the robot motor rotates in the direction of the tilt after detection and stabilises the robot in the upright position after the tilt is corrected.

The concept of this system is of balancing the robot by ensuring that the motors of the robot counteract the fall in either forward or backward direction. This action mainly requires the robot to perform feedback and correct actions at the same time. That is why the APM system was used with this robot system so that through the PID algorithm, the robot is programmed to move in the direction of the tilt for maintaining its balance and preventing the robot from falling (Memarbashi & Chang, 2011). The Arduino board is also used in this system, which was also programmed for the wheel encoders. The Arduino Pro Mini board uses the information provided by the APM to understand the latest orientation of the robot. Corrective action is performed by the combined action of the motor and wheels. The APM controller is used for this robot system mainly because this type of system is easy to use and provides GPS with a compass that helps in the navigation of a robot system (Ateov et al., 2017). Furthermore, APM can be programmed through the PID

algorithm, which allows us to ensure that the robot not only has the capability of moving from one point to another but is also capable of maintaining its balance or upright position (Ferdinando et al., 2011). The brushed DC motors and with CPR 48 encoders are the best choice of motor for this project because it gives the lower power 6V that is combined with the 34.104:1 super metal gearbox to ensure good integration with the board. After all the components were selected, the final design of the robot was prepared, after which the assembly phase began in which all these components were joined to resemble the robot's final design.

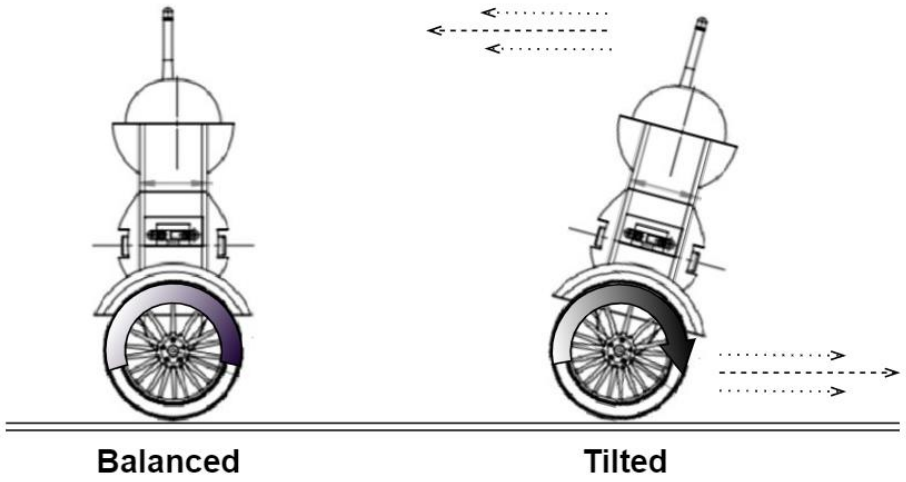


Figure 24: Functioning of the robot

Figure 24 shows the functioning of the smart two wheels balancing robot system. The motors and robot wheels rotate in the direction of the tilt to ensure that the robot maintains its balance and prevents the body from falling to the ground. All the components of the system were connected, as illustrated in the circuit diagram in Figure 8. Next, the two main programs for the Arduino Pro Mini and the APM were written. The code for APM is for controlling movement and balancing the robot while the code for Arduino Pro Mini is for acquiring the wheel velocity from the encoders. It was ensured before testing the robot that there is a good connection between the board and the controller. The concept of the inverted pendulum is essential for understanding the working of this robot system. According to this concept, the robot can be balanced much more adequately if the centre of mass is greater than that of the wheel axles. A higher centre of mass means a greater moment of inertia that corresponds to the lowered angular acceleration, i.e., a slower fall. That is why the battery is placed in the middle of the robot's body, as shown in Figure 22. The width and

height of the robot were measured beforehand to ensure that it is fully capable of integrating the board.

5.2 Construction of the System

The decision of 3D printing the chassis of the robot was made, and for that purpose, a 3D design of the robot was made using AutoCAD software. Figure 10 shows the 3D models of the robot chassis that were developed with AutoCAD software along with the 3D printed parts of the robot's body. Once the body parts of the robot were printed using a 3D printer, the board and the motors were integrated into the robot chassis, and the wheels were attached. The development of the board was done very simply, as shown in Figure 21. The APM was interfaced with the Arduino board, and the motors were connected through the motor driver module.

Furthermore, the 12V battery was chosen as the primary power supplier to the robot system, and each of these components of the robot was integrated into the robot chassis. After performing the selection of the components and joining the parts of the robot for the final design, the next step was the development of the PID algorithms for self-balancing the robot and for moving the robot. The system has to check if the robot is leaning backward or forward, and after detection, the wheels of the robot will rotate in the leaning direction to maintain balance (Esmaeili et al., 2017). At the same time, the goal was to control the speed of the robot wheels to ensure that the body does not fall over due to fast rotation. This is because, in this robot, the design is such that if the robot gets slightly disorientated from its centre position or the centre of gravity, the wheels have to slow down. However, the robot is programmed in such a way that the speed of the wheels increases as its body moves away from its central position. Hence, the PID algorithms were used in this project to set the centre position as the set-point.

To know the latest position of the robot, this system uses GPS with a compass that associated with APM. GPS allows not only the detection of the latest position of the robot but also its destination once it is in motion. The wheel encoders are used to get a reliable value of the speed of the robot and how much distance it covers, but it will be corrected by the GPS if the distance error more than a certain threshold. It has to be corrected because the wheel slippage and rough terrain make the pose estimation worse over time. Complete PID algorithms or codes for self-

balancing the robot and moving it from one place to another are illustrated in Appendices B and C, and the parameters are in Appendix D.

Firstly, in the development of the codes, all the relevant libraries were included that are required for this program to work properly (Ghaffari et al., 2016). For example, the built-in I2C library for acquiring the wheel encoder sensor, the APM sensor library for receiving data from the IMU, and the GPS were included in the code for performing various functions and calculations. After that, three cascaded controllers based on PID were implemented, compensating the error between the set-point and measurement sensor value controlling the tilt/pitch angle, yaw/heading angle, and wheel velocity. The first goal was to build the robot with an adequate centre of gravity and to ensure that the components of the robot are symmetrically arranged, which in many cases of self-balancing robots, is difficult to achieve. That is why, after ensuring that the robot design provides a good centre of gravity, the value of the set-point in the PID algorithm was set (Muhammed, et al., 2011). Also, the logger contains the pitch angle, yaw angle, and wheel velocity can be used to analyse the signal response. Although the PID gains have been tuned by Matlab/Simulink simulation, the response in the real experiment will not be optimal as in the simulation because the simulation process is approximated. Theoretically, the PID gains provided by the simulation can be used, but improving it again based on the data in the logger will give a better result.

The whole system task can be divided into five tasks, which are PID control according to the mode, encoder reading & robot pose estimation, data logging, auxiliary task (compass, battery, RC, LED, GCS), and waypoint APM & obstacle avoidance.

The most crucial task that must have the priority is the control of the pitch angle, yaw angle, and robot's velocity towards the goal destination, as shown in the flowchart as per Figure 25. For every loop, the first thing to do is to take the RC signal and the measurement of pitch and yaw angle through the IMU. After that, if the robot is not falling, it will calculate the input control using the PID controller and then send the PWM signal to the motor driver.

There are three modes for controlling the yaw and pitch angles, i.e., look at next waypoint mode, hold/stable mode, and manual/acro mode. The first mode is to look at the next waypoint, then the relative angle between the current robot pose and the destination pose should be calculated and become the yaw/heading angle reference. The velocity set-point is determined based on the

distance between the current robot pose and the goal pose, while the pitch angle set-point is set to zero to keep the robot balanced (upright). The second mode is position hold, whereas the robot has already arrived at the destination point. If the system meets this mode, then the robot must be steady, so that all of the references must be zero. The third mode is the manual/acro mode, which needs control from the RC. Nevertheless, if there is no RC input signal, then balance the robot. The PID controller is always running, whether the mode is automatic or manual, so that the robot maintains balance.

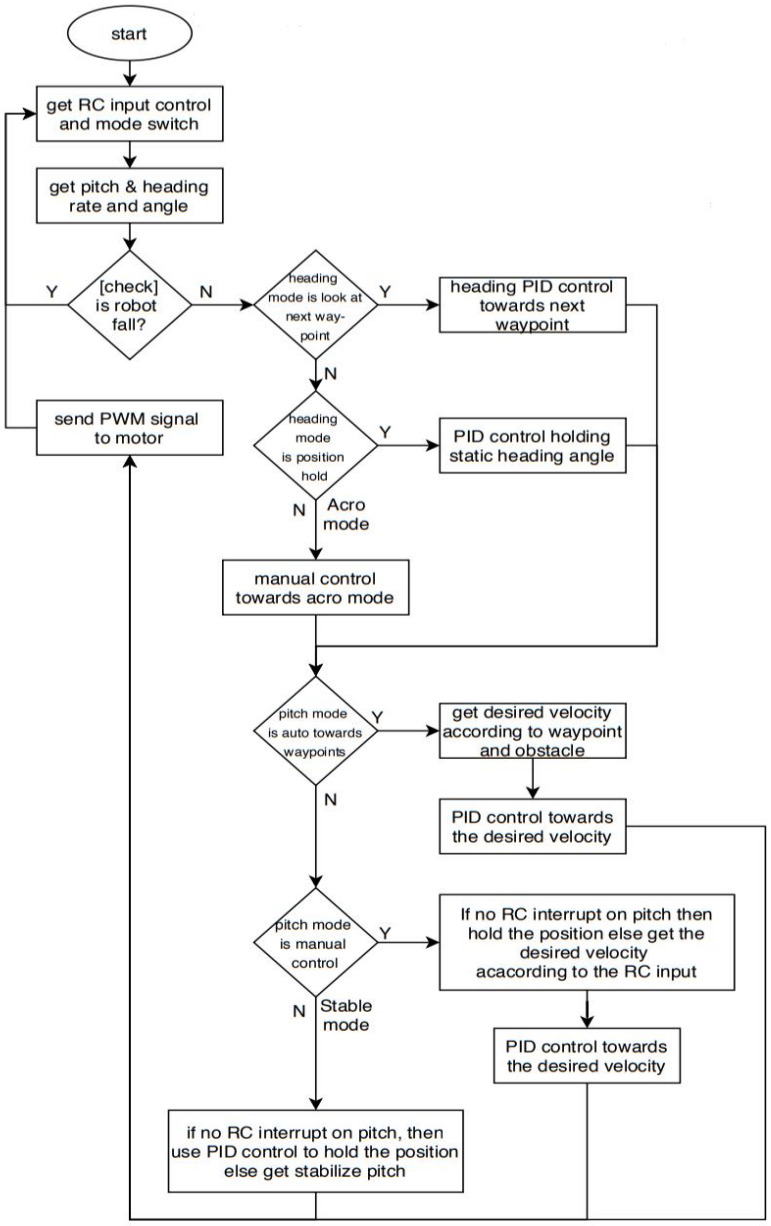


Figure 25: Flowchart of the PID controller in various modes

The wheel velocity measurement is important, as the feedback is compared with the velocity reference. However, it is not as crucial as the pitch angle measurement, so that the task that gets the velocity measurement has a lower priority than the pitch angle measurement. Moreover, if the wheel velocity measurement is also processed in the main task, then it will interfere because it needs to acquire the i2c from the Arduino Pro Mini that gets the measurement directly from the encoder. The velocity is measured from the wheel encoder then the displacement or robot position is estimated using an integral manner, which is commonly called "dead reckoning". The position estimation using the wheel encoder only is not good enough because of the disturbance, i.e., the wheel may slip or when the wheel is on the rough terrain. Therefore, GPS is used to correct the pose estimation if the estimation error between the dead reckoning method and GPS greater than a certain threshold. The flowchart is shown in Figure 26, and the complete code for the Arduino Pro Mini is illustrated in Appendix E.

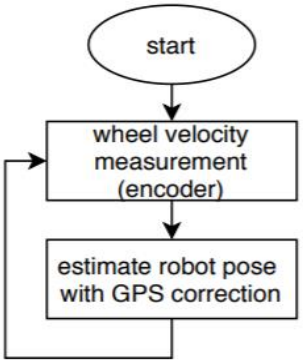


Figure 26: Flowchart of robot pose estimation

To analyse the control system, the state variables need to be saved. It is like a black box in the aeroplane; all variables are logged into a memory interfaced with the microcontroller. The data are the compass which is the heading angle, IMU contains pitch and yaw angle, GPS contains latitude, longitude, and altitude, navigation based encoder contains the wheel velocity and poses estimation, variable related to waypoint and obstacle contains the list of waypoint and obstacle status and the variables such as active mode and the performance of the multi-tasking. Figure 27 illustrates the flowchart for data logging.

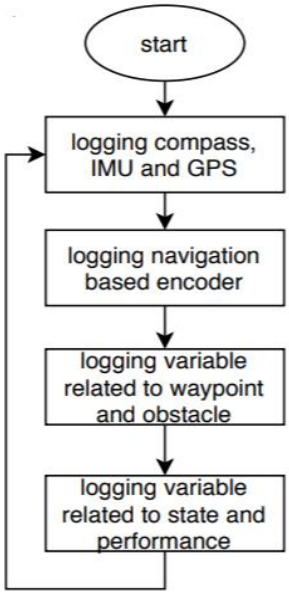


Figure 27: Flowchart of data logging

Another task involves doing some auxiliary tasks that have the lowest priority. The compass reading does not need to be done quickly because the IMU also measures the heading, but the compass is useful for correcting it. Furthermore, auxiliary performs standard procedures such as reading the battery status to see whether the robot should be returned to home or not, trim and tuning the remote control and read the input signal associated with the mode switch, read the signal from the GCS such as the waypoint or return to home command and toggle the LED-based on every device status. The flowchart of the auxiliary is illustrated in Figure 28.

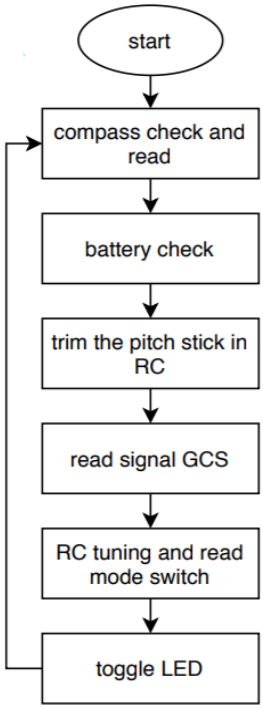


Figure 28: Flowchart of auxiliary task

The last process is to guide the robot based on the desired waypoints, as shown in Figure 29. The robot needs to get the position, which is the longitude and altitude to calculate the distance between the robot pose and the goal pose. The robot also needs to get the current heading and go-ahead to the goal pose. After that, to avoid an obstacle, do the sonar measurement, and if it is less than the threshold, then do an action, which is to go backward at a certain distance then continue to the next waypoint. There are three modes regarding the waypoint tracking, which is Return To Home (RTH), autonomous navigation, and guided using the remote control. If the mode is RTH, then the distance and heading to the home need to be calculated. If the mode is autonomous navigation, then the robot has to go to all of the waypoints sequentially until it reaches the home. The last mode is guided mode, whereas the robot only needs to wait for the input signal from the remote control.

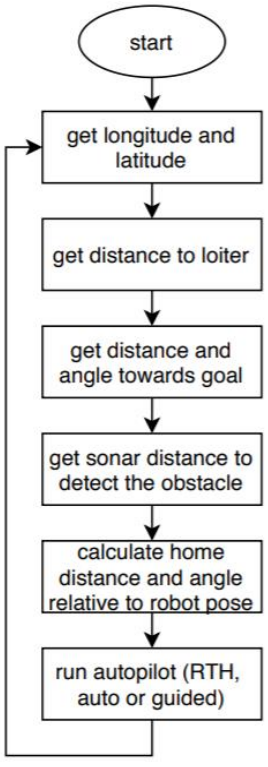


Figure 29: Flowchart of waypoint navigation

The four main programming software that was used for this project are as follows.

I. Matlab/Simulink software was used for the simulation part of the project.
II. ArduPilot-Arduino was used to program the APM. Through the help of this software, the PID algorithm for the APM controller was made that allowed the robot to balance and movement from one location to another. The theory of the inverted pendulum was followed in the development of the algorithm, which meant that the robot was programmed to rotate the motors in the direction of the tilt to ensure autonomous self-balancing, while the robot still goes forward and turning to the destination point.
III. The Arduino IDE software was used for programming the Arduino Pro Mini. It was used for the wheels encoder's data (pulses and speed) acquisition and relayed that data at the APM board through the I2C interface.
IV. Mission planner software was used in this project for the autonomous navigation of the robot system, including set home and several waypoints, send RTH command and see the robot status (pose, battery, etc.). This software did not require any coding, just setting up, and simulation.

5.3 Components of the Robot

- **ArduPilot Mega (APM 2.8)**

The ArduPilot Mega is an advanced quality IMU autopilot, which is based on the Arduino Mega platform, and APM 2.8 is considered the new autopilot (Open Source Drone Software. Versatile, Trusted, Open. ArduPilot., 2020). Sensors that are used in this system are the same as the APM 2.6 autopilot. This controller has an option to use the internal or external compass with the jumper. The APM is ideal for its use with robots, rovers, and multi-copters. The APM is a complete autopilot open-source system and is the bestselling software and has won the UAV challenge competition. Moreover, this system allows the user to change any fixed, multi-rotor vehicle or rotary-wing in a fully autonomous vehicle that has the capability of executing GPS-programmed missions with waypoints (Ateov et al., 2017). Furthermore, this board contains the optional compass on-board that is specifically designed for the vehicles where the placement of the compass is a requirement to be placed farther from motor and power sources as much as possible for avoiding magnetic interference. This controller is designed for its utilisation with GPS

having a compass to ensure that the compass/GPS unit gets mounted farther from the source of the noise APM itself. GPS is a requirement for APM since its fully autonomous; hence, extras like the GPS and compass are added to the APM in this project, as seen in Figure 30.

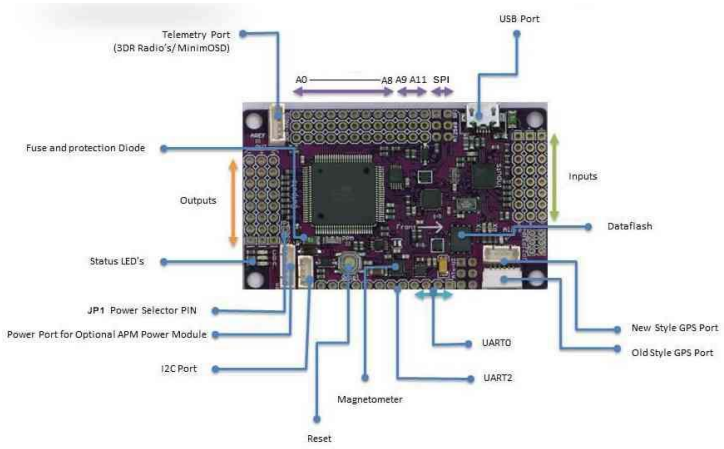


Figure 30: ArduPilot Mega (APM) Board

- **Arduino Pro Mini 328-Microcontroller**

The Arduino Pro Mini 328-microcontroller board is the Arduino 5V running at 16MHz of the bootloader, as seen in Figure 31. The purpose of using this system for this project is that this board is limited to a 5V, ensures a lacking the connectors, and allows off-board USB. Moreover, the board is very cost-effective as compared to other similar boards in the market, since it uses SMD components in a double layer. The board has the capability of connecting directly at the basic breakout FTDI board and provides support for auto-reset (Balung et al., 2017). The board works with FTDI; however, the FTDI cord does not provide a DTR pin. The auto-reset might not work. Another critical reason for choosing this board for this robot is that the voltage regulator on-board that allows the board to accept voltage approximately up to 12V DC allows the voltage to exceed above the normal limit of 5V. However, the important caution with these types of boards is that if unregulated power is supplied, the RAW pin has to be connected with the board and not VCC.

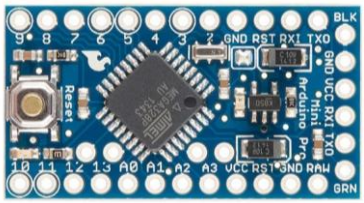


Figure 31: Arduino Pro Mini 328-microcontroller

The pin connection specifications for the Arduino Pro Mini, encoders, SDA, SCL of the APM signals can be found in Table 2.

Device (Component)	Pin
Encoder A output (A – yellow)	3
Encoder A output (B – white)	2
Encoder B output (A – yellow)	8
Encoder B output (B – white)	9
APM - SDA	A4
APM - SCL	A5

Table 2: Pin connection specifications for the Arduino Pro Mini

- **Telemetry Radios, 915MHz**

The telemetry radio is an inexpensive, small, and light radio open-source platform that allows different ranges that are more advantageous at 300m, as seen in Figure 32. The range of the telemetry radio can be extended to several more kilometres with the utilisation of the patch antenna on the ground. The radio mainly uses open-source firmware that is designed specifically for working with the packets of MAVLink and to get integrated with Copter, Mission Planner, Plane, and the Rover (Samson et al., 2011). The radios can be either 433 MHz or 915 MHz. The 915 MHz model is chosen for this project because it will allow higher transmission of signals for the robot.

Figure 32: Telemetry radio kit

- **R/C Receiver and Transmitter**

FlySky FS-i6 2.4G 6CH AFHDS radio transmitter with FS-iA6 receiver is attached to the robot's board to ensure its connection with the remote controller that will be used for controlling the robot's movement wirelessly. This component of the board is essential because it will ensure that this robot is controlled from a fair distance without loss of connection, just like any wireless device (Hsu & Sheen, 2011). The model of the used transmitter and receiver is, as shown in Figure 33.

Figure 33: RC Transmitter & Receiver

- **uBlox GPS Model with Compass**

The uBlox GPS module with a compass is the most commonly used GPS for the ArduPilot Mega Boards. The reason why this uBlox GPS module is used for this project is that APM has the capability of automatically configuring the GPS soon after startup, which ensures that there is no need for any GPS related calibrations with the system (Ateov et al., 2017). However, the compass integrated into this system has to be calibrated before use to avoid any errors in measurement.

Figure 34: uBlox GPS model

- **Ardumoto Motor Driver Shield**

The Ardumoto motor driver shield is the controller (dual-motor) for the Arduino and is based on H-bridge L298. The SparkFun Ardumoto has the capability of driving approximately two DC motors at 2A per channel, as seen in Figure 35. It is used in this project in combination with the Arduino because Ardumoto makes an effective controller platform, particularly for RC robots or vehicles, when it is in connection with the Arduino (Cancharoen, et al., 2015).

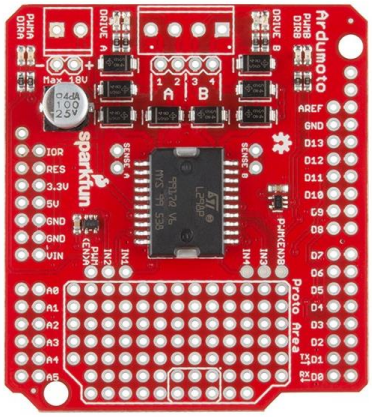


Figure 35: Ardumoto motor driver shield

This board mainly takes the powers from a similar Vin line like the Arduino board, and it incorporates yellow and blue LEDs that indicate the active direction. Also, driver lines on this board are mainly diode protected from the back EMF. The pin connection specifications for the Ardumoto motor driver shield, motors A, motor B, and APM signals can be found in Table 3.

Device (Component)	Pin
APM (Direction – Motor A)	2
APM (Direction – Motor B)	4
APM (PWM - Motor A)	3
APM (PWM - Motor B)	11
Motor A	A1
Motor A	A2
Motor B	B3
Motor B	B4

Table 3: Pin connection specifications for the Ardumoto motor driver shield

- **Brushed DC Motor with 48 CPR Encoder**

The gear motors 34:1 gear ratio with the 48 CPR rotary encoder is a gear motor that consists of the lower power 6 V DC brushed motor that is combined with the 34.104:1 super metal gearbox, as seen in Figure 36. It is also integrated with a 48 CPR quadrate rotary encoder that is present on the motor shaft providing 1632.67 counts per revolution of the output shaft of the gearbox (Acevedo & Alejo, 2014). The gear motor is cylindrical, having a diameter that is under 25 mm and the output shaft (D-shaped), which is 4 mm in diameter, and extends 12.5 mm from the gearbox's faceplate. The 34:1 gear reduction is chosen for increasing the torque (Madhira et al., 2016). Table 4 illustrates the specification of the chosen DC motor.

Figure 36: DC motor

Gear Ratio	34:1
No-Load Speed at 6 V	285 RPM
Stall Torque at 6V	60 oz-in
Stall Current at 6 V	6 A

Table 4: Specification of the DC motor

A two-channel Hall effect encoder is attached to the rear of the motor shaft to measure the rotation of the magnetic disk. When counting the edges of both channels, the quadrature encoder can provide a resolution of 48 counts per revolution of the motor shaft. The wire functions of the encoder used can be found in Table 5.

Colour	Function
Red	Motor Power (VCC)
Black	Motor Power (GND)
Blue	Encoder (VCC)
Green	Encoder (GND)
Yellow	Encoder output (A)
White	Encoder output (B)

Table 5: Encoder wire functions

In the quadrature encoder, the meaning of quadrature comes from the word "quarter", which means that 90° out of phase between the signal as seen in Figure (37). The A and B outputs of the encoder are a square wave starting from 0 to Vcc, and they are 90° of phase. The frequency of the transition can measure the speed of the motor, and the direction of the motor can be obtained by order of the transition (rising edge or falling edge). The following oscilloscope represents the A (yellow) and B (aqua) outputs of the encoder using a motor voltage of 6V and a Hall sensor Vcc of 5V, as seen in Figure (38).

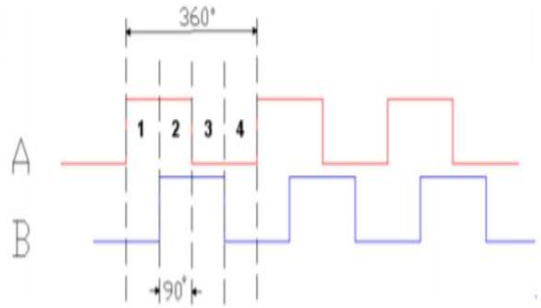


Figure 37: Theoretical signal of the quadrature encoder

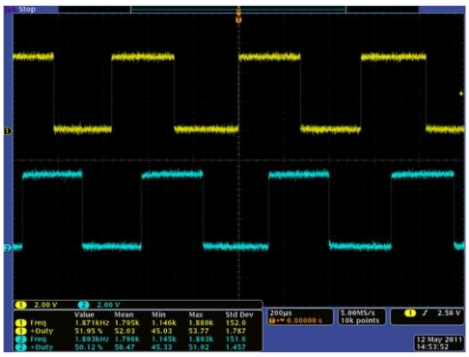


Figure 38: Oscilloscope signal of the quadrature encoder[70]

- **Wheels**

In this robot, two Pololu soft and rubber tires are chosen as wheels, as seen in Figure 39. The size of the wheel is 120x60mm with 4mm shaft adapters. The reason why soft and rubber tires are chosen for this robot because they will allow the wheels to have increased traction and be soft on

bumps making it more capable of traversing through various terrains easily and maintaining its balance for a longer duration compared to other lower quality.

Figure 39: Pololu robot wheel

All components used in this project are listed in Table 6, with the total cost of the project at the time of building it.

Items	Qty	Distributor	Price (AUD)
APM 2.8 ArduPilot	1	Amazon	$98
Arduino Pro Mini 328	1	eBay	$18.01
uBlox GPS with Compass	1	Amazon	$72.93
Telemetry radio-915MHz	2	Amazon	$42.58
Ultrasonic Sensor Module	1	Core Electronics	$11.5
Ardumoto - Motor Driver Shield (DEV-14129)	1	Core Electronics	$33.57
Level Translator Breakout (PCA9306)	1	Core Electronics	$9.83
FTDI FT232RL USB to TTL Serial Converter	1	eBay	$8.14
FlySky FS-i6 2.4G 6CH Transmitter	1	Banggood	$65.42
FS-iA6 Receiver	1	Banggood	$14.99

34:1 Metal Gearmotor 25Dx52L mm	2	Core Electronics	$48.53
POLOLU Wheel	2	Core Electronics	$23.55
FLOUREON RC LiPo Battery	2	eBay	$62.99
Screw Terminals 3.5mm Pitch	4	Core electronics	$4.99
4-Pin to Female Socket Cable, battery connectors, switches and cables	2	Core electronics	$30
Total			**$617**

Table 6: Project components cost

5.4 Analysis of Results

It is a challenging task to balance the inverted pendulum because, as described earlier, this mechanism is naturally unstable. A little disturbance or error from the equilibrium position of the robot, which is, in this project, the set-point angle of the robot chassis, i.e., 0°, causes the robot to lose its equilibrium position and balance. Eventually, the body of the robot falls to the ground. Therefore, to maintain the robot's balance, it is required that the motors and wheels of the robot rotate in the anticlockwise or clockwise direction. This robot system considers the tilt or deviation of the robot's body from the 0° angle as an error. It keeps rotating the motors and wheels of the robot in the direction of the tilt until the robot regains its balance and upright position, i.e., gets back to its set-point angle (Acevedo & Alejo, 2014). To ensure that this robot system is capable of working on the mechanism of an inverted pendulum, the PID controller is adopted that mainly uses the tilt/pitch feedback to control the motor’s torque and to keep the robot balanced. The PID controller continuously measures the process variable, calculates the error value, i.e., deviation from the set-point angle of 0°, and transmits it to the controller to move the motors and wheels of the robot in the direction of the tilt to get back the robot in a balanced position. The PID controller tries to minimise this type of error with time by continuously adjusting the control variable, i.e., the motor torque.

In Equation 17, U(t) represents the control variable, the variable e(t) represents the latest error in process variable, and Kp, Ki, and Kd are the parameters that have to be tuned to achieve the desired behaviour of the robot. To control the robot, both the gyroscope and accelerometer of the IMU are used. The angle information is obtained from the gyroscope. Moreover, the angular velocity and the direction that the robot is falling to are obtained from the accelerometer. When the angle information is not in the straight standing position, the APM will send a signal to the motor drivers to adjust the robot according to the opposite of the gyroscope's reading until the APM reads the set-point which is 0°. The accelerometer sensor mainly measures all forces that act upon an object, i.e., the robot's body, and sees more than g-force (gravity vector). Every force that is acting upon the object, for example, the frictional force, alters with the measurements to a large extent. While performing on the actuated system, forces that drive this system are easily visible on the sensor. The data of the accelerometer is reliable for a longer-term project, but to get that, low pass filters are needed (Yuan et al., 2016). In the case of the gyroscope, it is a lot easier to obtain accurate data, as it is not affected by external forces. However, due to integration over time, the measurements of sensors tend to drift and not come back to level zero, especially when this system returns towards its base position. However, it is important to understand that the sensor data is reliable only for a shorter period because it illustrates drift over the longer-term. A complementary filter used in this system provides data for both the long term and short term from the gyroscope and the accelerometer. The Arduino Pro Mini is used in this robot system to acquire data along with the filter.

Considering the motor's speed, it is made capable of adjusting by the PWM through the adjustment of the voltage and duty cycle that is given to the motor. With the use of the PWM, the expense of acquiring the digital to analogue converter is saved. Furthermore, another important benefit of using the PWM is that in this system, the signal stays digital, and no digital to analogue conversion is required. Also, by doing this, the noise effect is minimised. The changing of the PID controller's set-point individually for the two motors that are used in this system can help in controlling the translational motion of the robot system (Riattama et al., 2017). The motor power mainly rises by proportional term as this system leans further ahead and decreases the power of the motor when the system reaches the upright position, which is the required position for the robot. The gain factor Kp is responsible for determining how much power is needed to be applied to the motor for any given tilt or lean of the robot's body, as seen in Equation 26.

$$Proportional\ term = Kp * error$$ Equation 26

In the PID algorithm, the deferential term acts as the damper that reduces oscillation. Furthermore, another gain factor Kd determines the rate of power that is to be applied to the motor in accordance with Equation 27.

$$Differential\ term = Kd * (error - previous\ error)$$ Equation 27

Finally, neither the differential term nor the proportional term of the algorithm removes all of the tilt or lean because both of these terms get to zero as the orientation of this inverted pendulum system settles at the vertical position. Moreover, the accumulated errors are summed up by the integral term and try to drive the tilt or lean towards zero, as seen in Equation 28. Finally, this provides the PID controller's output, as seen in Equation 29.

$$Integral\ term\ (output) = Ki * (sum\ of\ errors)$$ Equation 28

$$PWM = integral\ term + porpotional\ term + differential\ term$$ Equation 29

Equation 29 represents the output of the motor's PWM, and this output is used as the motor's set-point, which is approximately at a 0° angle of the robot's body. Similarly, the motor's speed is also calculated as the sum of these where instead of the error of the tilt, the error of the motor speed is taken in all of the above equations. The PID algorithm controls the yaw angle, but it gives a positive input control to the left wheel and gives negative input control to the right wheel. In the tuning of the motor speed's PID control, the values of Ki, Kd, and Kp are calculated by Matlab/Simulink simulation. They can then be slightly improved in a real experiment. The H-bridge amplifier mainly amplifies the signal of the pulse-width modulator channel for the production of voltage, which is sufficient in driving the motor. Under the command of the software used in this system, the motor terminals are swapped by the H-bridge for driving the motor of the robot in a different direction. The use of the H-bridge saves expenses of using two voltage sources to get bidirectional control of the motor (Pratama et al., 2016). There are mainly two logic inputs in the H-bridge for the directional control of the motor. For the analysis of the experimental setup

response, the real-time data of the sensors is imported to Matlab at various conditions. The single PID computation for both motors is performed rather than performing separate computations for pitch angle and wheel velocity control. This single computation of both motors resolves the issue of delay in time due to the serial communication that has a critical effect upon the response of this setup, but the yaw angle control is separated between both motors. However, this type of implementation does not enable any sort of rotational movement along the vertical axis but allows the balancing of the robot in one translational direction.

To determine the maximum angle of tilt, beyond which the system is incapable of coming back to a stable position, the impulsive external forces were provided to the robot various times. The result of this experiment shows that the robot is capable of balancing itself many times. Still, it is incapable of balancing to the set-point angle of 0° if an external force pushes the body. However, according to experiments, the maximum angle of tilt is approximately 10° away from the set-point 0°. Thus, if the robot's body tilts farther than 10° or -10°, then the robot will be incapable of self-balancing itself (Riattama et al., 2017). To quantifying the position drift while this setup tries to balance itself, RPM is sent to the motors in the PWM form, which is acquired in the computer software and integrated into the controller. However, an inherent assumption is that the PWM signals sent to the motors of the robot are directly proportional to the motor's RPM. However, the position drift of this robot system varies whenever the experiment is repeated. The results of the experiments suggest that this robot system has the capability of self-balancing itself if the tilt is not beyond 10°.

5.5 Limitations

With the consideration of the three main goals of this project, the limitation of the project is that in the autonomous mission, the robot could not handle more than ten waypoints in navigation. Also, if the tilt of the robot's body is higher than 10°, then it is difficult for it to self-balance as the body loses its stability, and with higher gravitational force being applied on the body, it is difficult to stop the body from falling. However, this limitation does not heavily affect the primary goal of this project, which was to ensure that this robot is fully capable of having the self-balancing feature, which worked effectively in the experiments.

Chapter 6: Conclusion

This project provides all the valuable information about design, development, and testing of the smart two wheels balancing robot. At firstly, for the project, a literature review analysis was performed to find relevant data about the working and design of the robot. The goal of the literature review was to identify various ways by which proper development, design, and the functioning of the robot are ensured. To do this, a lot of relevant conference proceedings, journal articles, and web articles are sourced. The results of the literature review analysis illustrated that through the incorporation of adequate control systems in the smart two wheels balancing robot, the stability of this system can be ensured. Also, as the robot is moving across different terrains and obstacles, the goal of the project was to ensure that it is capable of balancing while moving through various obstacles. After performing the literature review, a lot of critical information about the development, design, and working of the robot was collected. The second phase in the development of this system was the acquisition of all the relevant components for the robot system. After the acquisition of all these components, the next phase in this project was to assemble these components before designing the body of the robot. A circuit of the robot system was made, and each component was connected accurately. The next phase was developing the physical design of the robot system. The final 3D design and body parts are shown in this report. After designing the robot, the next phase in this project was the assembly. The board, motor, chassis, and wheels of the robot were assembled to provide the final shape of the robot.

After that, the simulation using Matlab/Simulink to ensure the model is controllable or not and tuning the PID gains must be done before implementing the PID algorithms in the APM for self-balancing and navigating the robot, and the Arduino Pro Mini for collecting data from the wheel encoders were developed and integrated into the Arduino board and controller. The final stage in this project was testing the performance of the robot and find out whether the robot was capable of achieving its intended goals. The three main goals of this project were making the robot balance using the PID algorithm, making the robot move while balancing using the RC transmitter and receiver, and making it navigate using GPS, compass, and wheels encoders associated with Mission Planner software. The three goals, according to the results of the experiments, were achieved.

Appendices

Appendix A: Matlab Simulation Code

```
function [A,B,C,D] = fcn()

g = 9.81; % gravity constant
r = 0.06; % wheel radius
Mw = 0.15; % wheel mass
Mp = 0.9; % body mass
h = 0.22; % body height
l = 0.75*0.22; % distance from base of body to cog point
% approximately 75% of body height
w = 0.14; % width of body
d = 0.05; % depth of body
Iw = 0.5*Mw*r*r; % moment inertia of wheel
Ip = 1/12*Mp*((h*h)+(w*w)); % moment inertia of body (pitch direction)
Iy = 1/12*Mp*((d*d)+(w*w)); % moment inertia of body (yaw direction)
no_load_speed_rpm = 165; % motor speed without load in RPM
no_load_speed_rad_per_s = no_load_speed_rpm*pi/30.0; % motor speed without
load in rad/s
voltage = 6; % motor DC stall voltage
stall_current = 2.2; % motor DC stall current
stall_torque_oz_in = 40; % motor DC stall torque in oz-in
stall_torque_N_m = stall_torque_oz_in*0.00706155; %motor DC stall torque in
Nm
km = stall_torque_N_m/stall_current; % DC motor torque constaant (Nm/A)
ke = voltage/no_load_speed_rad_per_s; % Back EMF constant (V/(rad/s))
R = voltage/stall_current; % DC motor equivalent resistance

beta = 2*Mw+2*Iw/r/r + Mp;
alpha = Ip*beta+Mp*l*l*(Mw+Iw/r/r);

a22 =2*km*ke*(Mp*l*r-Ip-Mp*l*l)/(R*r*r*alpha);
a23 =Mp*g*l*l/alpha;
a42 =2*km*ke*(r*beta-Mp*l)/(R*r*r*alpha);
a43 =Mp*g*l*beta/alpha;

b2 =2*km*(-Mp*l*r+Ip+Mp*l*l)/(R*r*alpha);
b4 =2*km*(-r*beta+Mp*l)/(R*r*alpha);
b6 =w*km/(R*r*(Iy+(Iw/r/r+Mw)*w*w));

A = [0 1 0 0 0 0;
     0 a22 a23 0 0 0;
     0 0 0 1 0 0;
     0 a42 a43 0 0 0;
     0 0 0 0 0 1;
     0 0 0 0 0 0];
B = [0 0;
    b2/2 b2/2;
    0 0;
    b4/2 b4/2;
    0 0;
    b6 -b6];
C = eye(6);
D = [0 0;0 0;0 0;0 0;0 0;0 0];
```

Appendix B: Main Code (APM)

```
#define THISFIRMWARE "two_wheeled_robot"
// This project is created based on ardupilot library
// The most important constant in config.h (Appendix D):
// 1. wheel diameter in centimeters: WHEEL_DIAMETER_CM
// 2. PID constants for pitch control: BALANCE_P, BALANCE_I, BALANCE_D
// 3. PID constants for heading/yaw control: YAW_P, YAW_I, YAW_D
// 4. Proportional constant for wheel: WHEEL_P

#include <AP_Common.h> // these are mandatory library for running APM firmware
#include <AP_Progmem.h>
#include <AP_Menu.h>
#include <AP_Param.h>
#include <AP_HAL.h>
#include <AP_HAL_AVR.h>

#include <GCS_MAVLink.h> // for MAVLink integration
#include <AP_GPS.h> // for defining all supported GPS classes
#include <DataFlash.h> // for dataflash Log library
#include <AP_ADC.h> // analogue to digital converter
#include <AP_Compass.h> // for defining all supported compass classes
#include <AP_Math.h> // for defining math calculation (matrix/vector)
#include <AP_InertialSensor.h> // for measuring gyro and accel
#include <AP_AHRS.h> // support for IMU reading
#include <AC_PID.h> // PID controller
#include <RC_Channel.h> // for defining remote control channel
#include <AP_RangeFinder.h> // for sonar
#include <Filter.h> // for using filter
#include <AP_Buffer.h> // for using fifo buffer template class
#include <AP_Airspeed.h> // for getting air velocity_magnitude
#include <AR_EncoderNav.h> // for encoder reading and robot pose estimation
#include <AP_Declination.h> // for approximating magnetic declination using a lookup table
#include <memcheck.h> // for checking the memory limit
#include <AP_Scheduler.h> // for scheduling some tasks

#include "compat.h" // for converting from HAL to Arduino
#include "defines.h" // defining constant and enumeration
#include "config.h" // defining configuration (important: WHEEL_DIAMETER_CM and PID constants)

#include "Parameters.h" // library related to all parameters used
#include "GCS.h" // library related to Ground Control Station communication

#define ENCODER_ADDRESS 0x29 // I2C address (must be the same as in Arduino mini pro)

const AP_HAL::HAL& hal_object = AP_HAL_BOARD_DRIVER; // hardware abstraction layer (HAL)
object
```

```
static AP_HAL::BetterStream* serial_command; // for debugging through USB
static DataFlash_APM2 flash_logger; // logging to flash memory
static AP_Scheduler thread_scheduler; // define the thread thread_scheduler
static Parameters global_var; // define the parameters as a global variable
static GPS *gps; // GPS object
AP_GPS_Auto g_gps_driver(&gps); // gps driver
static AP_Compass_HMC5843 compass; // compass object
static AP_Int8 *mission_mode = &global_var.mission_mode1; // mode of flight
static AP_InertialSensor_MPU6000 imu; // IMU object
static const AP_InertialSensor::Sample_rate ins_sample_rate = AP_InertialSensor::RATE_200HZ; //
IMU rate
static AP_AHRS_DCM ahrs(&imu, gps); // direction cosine matrix for imu and gps (AHRS: attitude and
heading reference system)
ModeFilterInt16_Size3 filter_sonar(1); // sonar filter
static AP_HAL::AnalogSource *sonar_source; // sonar hardware source
static AP_RangeFinder_MaxsonarXL *sonar; // sonar object

static GCS_MAVLINK gcs_0; // initialize the GCS (Ground Control Station)
static GCS_MAVLINK gcs_3; // initialize the GCS (Ground Control Station)

static byte state_ctrl = STABILIZE; // initial control state is stabilising the robot
static unsigned byte prev_switch_pose; // previous switch poses on RC
static unsigned byte receiver_rssi; // RSSI: Receive Signal Strength Indication

static Vector3f attitude_rate; // rate of roll, pitch and yaw
double ch6_tuning_val; // value of tuned channel 6

static unsigned byte stat_led = NORMAL_LEDS; // initial led status
static unsigned byte led_indicator_gps; // led indicator of gps
static unsigned byte led_indicator_motor; // led indicator of motor
static byte led_indicator_navigation; // led indicator of navigation

static const double ten_pow_7 = 1.0e7f; // for scaling gps value
static double long_scale_up = 1; // for scaling gps value
static double long_scale_dn = 1; // for scaling gps value
static const double earth_rad = 6378100; // radius of earth in meters

static int heading_waypoint; // heading of robot relative to next waypoint
static int heading_waypoint_ori; // heading of robot relative to the first waypoint
static int heading_to_home; // heading of robot relative to home
static int distance_to_home; // distance between robot and home
unsigned int distance_to_next_waypoint; // distance between robot and next waypoint
double distance_loitering; // distance for loitering
static unsigned byte mode_navigation; // navigation mode (avoid back, avoid turn, loiter, waypoint)
static double cos_heading = 1; // variable for dcm (calculating the heading)
static double sin_heading = 1; // variable for dcm (calculating the heading)

static short distance_sonar; // distance sonar to obstacle
```

```
static unsigned byte is_sonar_healthy; // true if we can trust the altitude from the sonar

static unsigned byte heading_mode; // mode of heading (waypoint, hold, and acro)
static unsigned byte pitch_mode; // mode of pitch (auto, guided, and stabilise)

static struct Location current_loc; // robot current location
static int heading_navigation; // the heading navigation
static double delta_time_controller = 0.01; // delta time for integrator in PID controller
static AR_EncoderNav encoder_navigation(&gps); // encoder navigation with a checking from gps

static short pwm_motor_outputs[2]; // This is the array of PWM values being sent to the motors
static double cog_offset; // centre of gravity offset

static short pitch_output; // the output of pitch control (input for plant/model)
static short heading_output; // the output of heading control (input for plant/model)

static double velocity_reference; // reference of robot velocity
static double encoder_ticks_reference; // reference of encoder ticks
static double distance_obstacle_navigation; // distance to avoiding the obstacle

AP_HAL::Semaphore* semaphore_i2c; // semaphore for the i2c object
short failed_i2c; // to check the i2c is failed or not

Vector3f goal; // goal position of robot (equal to the next waypoint position)
bool arrive_at_goal; // true if the robot arive at the goal

static unsigned int last_main_thread_timer; // timer in the main loop
static byte counter_start_delay; // counter for motor starting delay
static unsigned int last_update_gps_timer; // timer for gps update
static unsigned byte counter_pitch_auto_trim; // auto trim for pitch rc control

static AP_HAL::AnalogSource* source_rssi; // rssi (connection to gcs)
static AP_HAL::AnalogSource* source_voltage_battery; // related to battery voltage monitoring
static AP_HAL::AnalogSource* source_current_battery; // related to battery current monitoring
static AP_HAL::AnalogSource* source_board_vcc; // related to vcc board monitoring

AP_Param param_loader(var_info, WP_START_BYTE); // load the parameter
static union {
  struct {
    unsigned byte is_armed : 1; // is the robot armed?
    unsigned byte is_rc_overwritten : 1; // is the rc have overwritten?
    unsigned byte is_home_set : 1; // is the home position (original position of robot) set?
    unsigned byte is_pose_hold : 1; // is the robot state is hold
    unsigned byte is_batt_low : 1; // is the battery low?
    unsigned byte stat_obs : 1; // show the obstacle status
    unsigned byte stat_gps : 1; // show the gps status
    unsigned byte stat_compass : 1; // show the compass status
  };
```

```
   unsigned short val;
} twr;

static struct AP_System{
   unsigned byte stat_channel_7 : 1; // show the status of channel 7
   unsigned byte frame_radio_new : 1; // show the new radio frame
   unsigned byte is_usb_connect : 1; // show if the usb connected or not
   unsigned byte stat_gps_led : 1; // show the led of GPS status
   unsigned byte stat_motor_led : 1; // show the led of motor status
} twr_sys;

static struct { // wheel velocity and distance
            short left_distance;
            short right_distance;
            short left_velocity;
            short right_velocity;
   short left_velocity_output;
   short right_velocity_output;
            double velocity;
} wheel;

static union { // i2c buffer (for receiving signal from Arduino pro mini)
   int value_long;
   short value_int;
   unsigned byte array_byte[];
} bytes_union;

static const AP_Scheduler::Task list_thread[] PROGMEM =
{ // list of thread {name_of_thread, interval ticks, max time in micros}
   {thread_gps_update, 2, 900},
   {thread_auxiliary, 2, 900},
   {thread_encoder_read_pose_estimation, 2, 950},
   {thread_navigation, 10, 800},
   {thread_logger, 10, 500},
   {thread_gcs_input_checking, 2, 700},
   {thread_gcs_heartbeat_sending, 100, 700},
   {thread_gcs_send_data_streaming, 2, 1500},
   {thread_gcs_deferred_sending, 2, 1200},
};

void setup() {
   // first initialization
   memcheck_init(); // initialize memory
   serial_command = hal_object.console; // initialize the serial command for debugging
   AP_Param::setup_sketch_defaults(); // load default values for all scalars in a sketch

   sonar_source = hal_object.analogin->channel(CONFIG_SONAR_SOURCE_ANALOG_PIN); // initialize
the analog pin for the sonar hardware
```

```
   sonar = new AP_RangeFinder_MaxsonarXL(sonar_source,&filter_sonar); // initialize sonar object
with a filter

   // gcs initialization
   source_rssi = hal_object.analogin->channel(global_var.rssi_pin, 0.25);
   source_voltage_battery = hal_object.analogin->channel(global_var.battery_volt_pin);
   source_current_battery = hal_object.analogin->channel(global_var.battery_curr_pin);
   source_board_vcc = hal_object.analogin->channel(ANALOG_INPUT_BOARD_VCC);

   init_ardupilot(); // intialize the ardupilot firmware

   thread_scheduler.init(&list_thread[0], sizeof(list_thread)/sizeof(list_thread[0])); // all thread start
}

void loop() // main thread (fastest thread for control system)
{
   unsigned int timer = micros(); // get time stamp in microseconds
   if (imu.num_samples_available() >= 2) // read IMU if the number of sample more than 2
   {
      delta_time_controller = (double)(timer - last_main_thread_timer)/1.0e6f; // for PI and PID
controller
      last_main_thread_timer = timer;
      read_rc_input(); // get rc input for pitch, wheel and heading control
      read_rc_mode_switch(); // get rc input for mode switch
      read_imu(); // get pitch and heading rate
      get_heading(); // get heading angle
      if(is_robot_not_falling()) // check whether the robot is falling or not
      {
         heading_control(); // controlling heading with PID controller according to the mode
         pitch_control(); // controlling pitch with PID controller according to the mode
      }
      write_pwm_motor(); // write the pwm signal to the motors
      thread_scheduler.tick(); // give tick to other threads
   }
   else // do nothing (delay 10 ms using thread scheduler) -> give other threads some space
   {
      unsigned short dt = timer - last_main_thread_timer;
      if (dt < 10000)
         thread_scheduler.run(10000 - dt);
   }
}

static void thread_auxiliary() // thread for compass, rc, led indicator
{
   // create a loading for the motor to spin
   if(counter_start_delay == -1 || counter_start_delay > 100)
      counter_start_delay = -1;
   else
```

```
      counter_start_delay++;

   // check and read the compass
   set_compass_healthy(compass.healthy);
   if(global_var.compass_enabled)
   {
      if(state_ctrl == FBW)
         compass.save_offsets();
      if (compass.read())
         compass.null_offsets();
      compass.accumulate();
   }

   if (global_var.battery_monitoring != 0) // check the battery
      read_battery();

   pitch_auto_trim(); // slightly adjusts the ahrs.pitch_trim towards the current stick positions
   read_signal_gcs(); // read signal from ground control station

   if(global_var.radio_tuning > 0) // tuning the rc
     remote_control_tuning();
   read_trim_switch(); // read mode sitch (channel 7)

   update_gps_motor_led(); // update GPS and motor led
   update_apm_leds(); // update APM led
}

static void thread_encoder_read_pose_estimation() // thread related to encoder (measure speed and
estimate position of the robot)
{
   read_encoder_sensor(); // get data encoder to update wheel velocity
   estimate_robot_pose(); // estimate robot pose from data encoder
}

static void thread_logger() // thread for logging datas to  flash memory
{
   static unsigned char log_counter_inav = 0;

   if(twr.is_armed)
   {
      if (global_var.log_bitmask & MASK_LOG_COMPASS)
         Log_Write_Compass(); // log compass value

      if (global_var.log_bitmask & MASK_LOG_IMU)
         flash_logger.Log_Write_IMU(&imu); // log pitch and heading angle and rate

      if (global_var.log_bitmask & MASK_LOG_GPS)
         flash_logger.Log_Write_GPS(gps, current_loc.alt); // log longitude, latitude, altitude
```

```
      if(global_var.log_bitmask & MASK_LOG_INAV) // log encoder navigation
      {
         log_counter_inav++;
         if(log_counter_inav >= 10)
         {
            log_counter_inav = 0;
            Log_Write_INAV();
         }
      }

      Log_Write_NTUN(); // log variable related to waypoint and obstacle
   }

   if (global_var.log_bitmask != 0)
      Log_Write_Data(DATA_AP_STATE, twr.val); // log variable related to state

   if (global_var.log_bitmask & MASK_LOG_PM)
      Log_Write_Performance(); // log performance info
}

static void thread_gps_update(void) // get GPS data
{
   gps->update(); // read the GPS (long, lat and alt)
   update_GPS_light(); // control the GPS led
   set_gps_healthy(gps->status() >= GPS::GPS_OK_FIX_3D); // set the GPS health to fix 3D
   if (gps->new_data && last_update_gps_timer != gps->time && gps->status() >=
GPS::GPS_OK_FIX_2D) // are we have a new gps data?
   {
      gps->new_data = false;
      last_update_gps_timer = gps->time;
   }
}

void heading_control(void) // control the heading, rc channel 1 for heading control
{
   static int target_heading = 0;
   static byte heading_counter = 0;
   if(heading_mode == HEADING_LOOK_NEXT_WAYPOINT) // goes to the next waypoint location
   {
      if(mode_navigation == NAV_AVOID_TURN) // turn the robot 90 degree because of obstacle
      {
         heading_navigation = limit_360_centi_degree(heading_waypoint + 9000); // 9000 centi degree
= 90 degree (change the heading goal by 90 degree)
         target_heading = limit_heading_rate(target_heading, heading_navigation,
AUTO_YAW_SLEW_RATE+80); // limit the heading rate to the maximum heading rate
         heading_output = get_stable_heading_pid_control(target_heading); // get input control u(t)
from target heading reference with PID controller to robot
```

```
      }
      else
      {
         if(global_var.sonar_enabled) // avoid the obstacle if sonar is active
         {
            heading_navigation = avoid_obstacle(heading_waypoint); // change the heading due to the
presence of obstacle
         }
         else
         {
            heading_navigation = heading_waypoint; // the heading goal is the heading at the next
waypoint
            heading_navigation = waypoint_trajectory_following(heading_navigation); // for keeping the
robot path on the waypoint path
         }
         target_heading = limit_heading_rate(target_heading, heading_navigation,
AUTO_YAW_SLEW_RATE); // limit the heading rate to the maximum heading rate
         heading_output = get_stable_heading_pid_control(target_heading); // get input control u(t)
from target heading reference with PID controller to robot
      }
   }
   else if(heading_mode == HEADING_HOLD) // hold the heading angle
   {
      target_heading = ahrs.yaw_sensor;  // get the heading measurement
      if(global_var.rc_1.control_in != 0) // there is a heading control from the rc
      {
         heading_output = global_var.rc_1.control_in; // straightly give the value from the rc input to
heading output
         heading_counter = 100; // give 1 sec to decelerate (the main loop rate is 100 Hz)
      }
      else // there is no control from the rc
      {
         if(heading_counter > 0) // counter below 1 sec
         {
            heading_counter--; // keep decrement the counter until it goes to zero
            if(heading_counter == 0) // it has already 1 sec
            {
               heading_navigation = ahrs.yaw_sensor; // hold the heading to current heading after 1 sec
            }
            heading_output = 0; // initialize the heading output to zero before it is controlled with PID
         }
         else // after 1 sec decelerating, stabilise the heading
         {
            heading_output = get_stable_heading_pid_control(heading_navigation); // get input control
u(t) from target heading reference with PID controller to robot
         }
      }
   }
```

```
   else if(heading_mode == HEADING_ACRO) // acro: release the sticks and the robot will maintain its current pitch and heading and will not return to level
   {
      heading_output = global_var.rc_1.control_in/2.0; // reduce the rc value 50%
      target_heading = ahrs.yaw_sensor; // get the heading measurement
   }
}

void pitch_control(void) // control the pitch angle (stabilise the robot)
{
   if(twr_sys.frame_radio_new) // if new radio frame received then toggle it
      twr_sys.frame_radio_new = false;

   if(pitch_mode == PITCH_AUTO) // the mode is auto (based on the waypoint)
   {
      if (mode_navigation == NAV_AVOID_BACK) // go back because of obstacle
      {
         velocity_reference = limit_acceleration(VELOCITY_REFERENCE_AVOID_BACK, ACCELERATION_LIMIT_AUTO_NAV_AVOID_BACK); // limit the acceleration
         clear_obstacle_counter(); // clear the obstacle counter
         if(distance_obstacle_navigation > 100) // go back 1 meter
         {
            mode_navigation = NAV_AVOID_TURN; // after go back 1 meter, then do turning
            distance_obstacle_navigation = 0; // reset the distance_obstacle_navigation
         }
      }
      else if (mode_navigation == NAV_AVOID_TURN) // turning because of obstacle
      {
         velocity_reference = limit_acceleration(VELOCITY_REFERENCE_AVOID_TURN, ACCELERATION_LIMIT_AUTO_NAV_AVOID_TURN); // limit the acceleration
         clear_obstacle_counter(); // clear the obstacle counter
         if(distance_obstacle_navigation > 100) // if the robot have turned 1 meter then change the mode to waypoint navigation
            mode_navigation = NAV_WP; // change the mode to waypoint navigation
      }
      else if (mode_navigation == NAV_LOITER) // keep the robot in loiter position
      {
         velocity_reference = limit_acceleration(get_loiter_speed(), ACCELERATION_LIMIT_AUTO_NAV_LOITER); // limit the acceleration
         twr.is_pose_hold = true; // make the pose hold because it will loiter
         clear_obstacle_counter(); // clear the obstacle counter
      }
      else if(mode_navigation == NAV_WP) // the robot follows the waypoints
      {
         velocity_reference = get_waypoint_velocity_reference(); // get velocity reference from a set of waypoint
         velocity_reference = limit_acceleration(velocity_reference, ACCELERATION_LIMIT_AUTO_NAV_WP); // limit the acceleration
```

```
        twr.is_pose_hold = false; // make the pose not hold
        check_obstacle(); // check the presence of obstacle
      }
      calculate_pitch_output_pid_controller(velocity_reference); // calculate the input pitch control
u(t)
    }
    else if(pitch_mode == PITCH_GUIDED) // the mode is guided/manual control from the rc
    {
      if(global_var.rc_2.control_in == 0) // no rc interrupt
      {
        if(!twr.is_pose_hold) // previously the state is not hold
        {
          twr.is_pose_hold = true; // make it hold
          set_goal(encoder_navigation.get_position()); // set the goal according to the encoder
          arrive_at_goal = true; // assume the robot arrives at the goal location
        }
        velocity_reference = limit_acceleration(get_loiter_speed(), ACCELERATION_LIMIT_HOLD); //
limit the acceleration
      }
      else // rc interrupt
      {
        twr.is_pose_hold = false; // the pose is not hold
        velocity_reference = ((double)global_var.rc_2.control_in / (double)MAX_INPUT_PITCH_ANGLE)
* -global_var.waypoint_speed; // get velocity from rc input channel 2
        velocity_reference   = limit_acceleration(velocity_reference, ACCELERATION_LIMIT_GUIDED);
// limit the acceleration
      }
      calculate_pitch_output_pid_controller(velocity_reference); // calculate the input pitch control
u(t)
    }
    else if(pitch_mode == PITCH_STABILIZE) // stabilise the pitch angle
    {
      if(global_var.rc_2.control_in == 0) // no interruption from the rc
      {
        if(!twr.is_pose_hold) // previously the state is not hold
        {
          twr.is_pose_hold = true; // make it hold
          set_goal(encoder_navigation.get_position()); // set the goal according to the encoder
          arrive_at_goal = true; // assume the robot arrives at the goal location
        }
        velocity_reference = limit_acceleration(get_loiter_speed(), ACCELERATION_LIMIT_HOLD); //
limit the acceleration
        calculate_pitch_output_pid_controller(velocity_reference); // calculate the input pitch control
u(t)
      }
      else // interruption from the rc
      {
```

```
        pitch_output = (get_stable_pitch_pid_control(global_var.rc_2.control_in) +
get_velocity_pitch_from_wheel()); // get input pitch control from pid controller (add output from PID
control and wheel velocity)
      }
   }
}

// get roll, pitch, heading rate but we only need the pitch and heading
static void read_imu(void)
{
   ahrs.update();
   attitude_rate = imu.get_gyro();
}

// get heading with Direction Cosine Matrix: http://www.starlino.com/dcm_tutorial.html
static void get_heading(void)
{
   const Matrix3f &dcm_matrix = ahrs.get_dcm_matrix(); // initialize the matrix
   Vector2f vector_heading; // initialize the vector heading

   vector_heading.x = dcm_matrix.a.x; // get cos value from dcm matrix
   vector_heading.y = dcm_matrix.b.x; // get sin value from dcm matrix
   vector_heading.normalize(); // vector normalization

   cos_heading = constrain(vector_heading.x, -1.0, 1.0); // limit the value between -1 and 1
   sin_heading = constrain(vector_heading.y, -1.0, 1.0); // limit the value between -1 and 1
}

static void remote_control_tuning() // tune the RC
{
   ch6_tuning_val = (double)global_var.rc_6.control_in / 1000.0f; // get the value from the rc channel
6
   global_var.rc_6.set_range(global_var.radio_tuning_low,global_var.radio_tuning_high); // set range
for rc channel 6 value
}

// autopilot navigation thread
static void thread_navigation(void)
{
   current_loc.lng = encoder_navigation.get_longitude(); // get longitude from gps
   current_loc.lat = encoder_navigation.get_latitude(); // get latitude from gps

   distance_loitering = distance_to_loiter(); // get distance to loiter
   distance_to_next_waypoint = distance_to_goal(); // get distance to next waypoint
   heading_waypoint = heading_to_goal(); // get heading to next waypoint

   distance_sonar += read_sonar(); // get distance between the sonar and obstacle
   distance_sonar >>= 1;
```

```
   if(twr.is_home_set) // if the home is set
   {
      Vector3f current_position = encoder_navigation.get_position(); // get current position
      heading_to_home = angle_two_vectors(current_position, Vector3f(0, 0, 0)); // get heading to
home
      distance_to_home = pythagorous2(current_position.x, current_position.y); // get distance to
home
   }
   else // if it is not set then initialize it to zero
   {
      distance_to_home = 0;
      heading_to_home = 0;
   }

   if(arrive_at_goal == false) // if the robot have not arrived at the goal yet
      // if the distance to the waypoint less than the waypoint radius or the waypoint is missing then
      //make the robot status is arrive at the goal
      arrive_at_goal = (distance_to_next_waypoint < global_var.waypoint_radius) ||
check_missed_wp();

   if(state_ctrl == RETURN_TO_HOME) // return to home state
   {
      if(arrive_at_goal) // if the robot has arrived then change the mode to fly by wire
         set_mode(FBW);
   }
   else if(state_ctrl == AUTO) // autopilot from ardupilot library
   {
      update_commands();
      verify_commands();
   }
   else // this must be guided mode
   {
      // do nothing
   }
}

AP_HAL_MAIN(); // mandatory for APM firmware
```

Appendix C: Control System Code

```
/// -*- tab-width: 4; Mode: C++; c-basic-offset: 4; indent-tabs-mode: nil -*-

static double wheel_ratio; // used to convert ticks into 1000 t/s for 1 wheel rotation
static unsigned int balance_timer; // to get the time of pitchover

static bool is_robot_not_falling()
{
   unsigned int current_time = millis();

   if(labs(ahrs.pitch_sensor) > 4000) // if the pitch angle more than 40 degree or less than -40 degree
   {
      balance_timer = current_time;
      if(twr.is_armed) // if the motor is armed
      {
         heading_navigation = ahrs.yaw_sensor; // hold the heading
         heading_output = 0; // give the heading output zero
         pitch_output = 0; // give the pitch output zero
         encoder_navigation.set_current_position(gps->longitude, gps->latitude); // set final position
         set_goal(encoder_navigation.get_position()); // set goal position
         clear_all_integrator(); // reset the integrator
         init_disarm_motors(); // disarm the motor
      }
      return false;
   }

          //no output until we have been upright for 3 seconds
   if((current_time - balance_timer) < 3000) // if the robot does not get up for 3 secs
   {
      // the robot does fall, nothing to do
      return false;
   }
   else
   {
            if(!twr.is_armed) // try to wake up
      {
         init_arm_motors(); // re-arm the motor
               set_goal(encoder_navigation.get_position()); // hold the position
            }
   }
   return true;
}

static short get_stable_pitch_pid_control(int pitch_reference) // pitch_reference = r(t)
{
   int pitch_error = wrap_180_cd(pitch_reference - ahrs.pitch_sensor);  // (e(t) = r(t) - y(t))
```

```
   int pitch_error_dot = 0 - (attitude_rate.y * DEGX100); // differential of pitch error is the pitch
rate*(-1)

   short error_proportional = global_var.pid_balance.kP() * (double)pitch_error;
   short error_integral = global_var.pid_balance.get_i(pitch_error, delta_time_controller); // integrate
the pitch error
   short error_differential = global_var.pid_balance.kD() * (double)pitch_error_dot;

   serial_command->printf_P(PSTR("a_pitch:%d, e_pitch:%d, e_P:%d, e_D:%d\n"),
(short)ahrs.pitch_sensor, pitch_error, error_proportional, error_proportional);

   return error_proportional + error_integral + error_differential;
}

static short get_velocity_pitch_from_wheel()
{
   serial_command->printf_P(PSTR("wheel_velocity:%d\n"), wheel.velocity);
   return global_var.p_vel*wheel.velocity; // Kp_wheel*wheel_velocity
}

static short get_stable_heading_pid_control(int heading_reference)
{
   int heading_error = wrap_180_cd(heading_reference - ahrs.yaw_sensor); // heading angle error
   heading_error = constrain(heading_error, -1500, 1500); // limit the heading error
   short heading_output = (double)global_var.pid_yaw.get_pid(heading_error,
delta_time_controller)/wheel_ratio; // heading output
   return heading_output;
}

static void clear_all_integrator(void) // reset all integrator
{
   cog_offset = 0;
   global_var.pid_balance.clear_integrator();
   global_var.pid_yaw.clear_integrator();
   reset_navigation_integrator();
}

static void reset_navigation_integrator() // reset navigation integrator only
{
   global_var.pid_wheel.clear_integrator();
}

static void init_control_system()
{
   gps->longitude = 0; // initialize longitude
   gps->latitude = 0; // initialize latitude
   wheel_ratio = 1000.0 / (double)global_var.wheel_encoder_speed; //convert ticks into 1000 t/s for
one rotation
```

```
    failed_i2c = 0; // sum of failed i2c transmission is initially zero
    semaphore_i2c = hal_object.i2c->get_semaphore(); // get the i2c semaphore from other threads
    if (!semaphore_i2c->take(HAL_SEMAPHORE_BLOCK_FOREVER))
        hal_object.thread_scheduler->panic(PSTR("Failed to get Encoder semaphore"));
    semaphore_i2c->give(); // give the i2c semaphore to other threads
    heading_navigation = ahrs.yaw_sensor; // hold the heading
    set_armed(false); // don't arm the motor now
}

static double convert_encoder_from_velocity(double wheel_velocity) // convert wheel velocity to
encoder ticks
{
    return wheel_velocity*1000.0/global_var.wheel_diameter;
}

static double convert_velocity_from_encoder(double encoder) // convert encoder ticks to wheel
velocity
{
    return encoder*global_var.wheel_diameter/1000.0;
}

static bool read_encoder_sensor() // reading the encoder through i2c
{
    unsigned char array_buffer[12]; // the buffer of i2c
    if (hal_object.i2c->read(ENCODER_ADDRESS, sizeof(array_buffer), array_buffer) != 0) // do not read
the encoder if the i2c failed
    {
        semaphore_i2c->give(); // give the i2c semaphore to other threads
        failed_i2c++; // increment the number of i2c communcation failure
        return false;
    }

    if (!semaphore_i2c->take(5)) // do not read the encoder if it does not takes 5
      return false;

    semaphore_i2c->give(); // give the i2c semaphore to other threads

    // initialize dynamic array
    memcpy(bytes_union.array_byte, &array_buffer[1], 2);
    memcpy(bytes_union.array_byte, &array_buffer[3], 2);
    memcpy(bytes_union.array_byte, &array_buffer[5], 2);
    memcpy(bytes_union.array_byte, &array_buffer[7], 2);

    // calculating distance and velocity of wheel left and right
    wheel.left_distance = bytes_union.value_int*WHEEL_ENCODER_DIR_LEFT;
    wheel.left_velocity = bytes_union.value_int*WHEEL_ENCODER_DIR_LEFT;
    wheel.right_distance = bytes_union.value_int*WHEEL_ENCODER_DIR_RIGHT;
    wheel.right_velocity = bytes_union.value_int*WHEEL_ENCODER_DIR_RIGHT;
```

```
   double velocity_1_rot_per_sec = ((double)(wheel.left_velocity + wheel.right_velocity) *
wheel_ratio)/2.0; // wheel velocity 1 rotation per sec divided by 2 because there are 2 wheels
   wheel.velocity = (wheel.velocity + velocity_1_rot_per_sec)/2.0; // average the wheel velocity
(prev_velocity+current_velocity)/2

   ground_velocity = convert_velocity_from_encoder(wheel.velocity); // convert to ground velocity
   distance_obstacle_navigation += fabs(ground_velocity)*0.02; // sum the distance related to
obstacle mode
   encoder_navigation.set_velocity(ground_velocity*sin_heading, ground_velocity*cos_heading); //
set the velocity
   return true;
}

static void estimate_robot_pose()
{
   static double last_GDT = 0;
   encoder_navigation.update(delta_time_controller + last_GDT); // update the encoder navigation
(position based encoder only)
   last_GDT = delta_time_controller; // record the last time for calculating the next delta time
}

// calculate_pitch_output_pid_controller
static void calculate_pitch_output_pid_controller(float velocity)
{
   // slow down in front of obstacles
   if(twr.stat_obs)
      velocity = velocity/2;

   int16_t bal_out = 0;
   int16_t vel_out = 0;
   int16_t nav_out = 0;
   float wheel_speed_error;
   int16_t ff_out;

   // switch units to encoder ticks
   encoder_ticks_reference   = convert_encoder_from_velocity(velocity);

   // grab the wheel velocity error
   wheel_speed_error = wheel.velocity - encoder_ticks_reference;

   // 4 components of stability and navigation
   bal_out  = get_stable_pitch_pid_control(0); // hold as vertical as possible
   vel_out  = get_velocity_pitch_from_wheel(); // wheel.velocity * 1.0
   ff_out   = (-encoder_ticks_reference) * global_var.throttle; // encoder_ticks_reference * 1.0

   if(twr.is_pose_hold)
   {
```

```
        global_var.pid_wheel.clear_integrator();
        nav_out = 0;
    }
    else
    {
        nav_out  = global_var.pid_wheel.get_i(wheel_speed_error, delta_time_controller);
    }

    // sum the output for u(t)
    pitch_output  = bal_out + vel_out + nav_out  + ff_out;
}
```

Appendix D: Parameters in config.h File

```
// PID constants for pitch control
#ifndef BALANCE_P
 # define BALANCE_P                1201
#endif
#ifndef BALANCE_I
 # define BALANCE_I                10872
#endif
#ifndef BALANCE_D
 # define BALANCE_D                31
#endif
#ifndef BALANCE_IMAX
 # define BALANCE_IMAX             200
#endif

// PID constants for heading/yaw control
#ifndef YAW_P
 # define YAW_P                    0.57
#endif
#ifndef YAW_I
 # define YAW_I                    0.14
#endif
#ifndef YAW_D
 # define YAW_D                    0.27
#endif
#ifndef YAW_IMAX
 # define YAW_IMAX                 400
#endif

// PID constants for wheel control
#ifndef WHEEL_P
 # define WHEEL_P                  45
#endif
#ifndef WHEEL_I
 # define WHEEL_I                  1236
#endif
#ifndef WHEEL_D
 # define WHEEL_D                  0.000
#endif
#ifndef WHEEL_IMAX
 # define WHEEL_IMAX               0
#endif
```

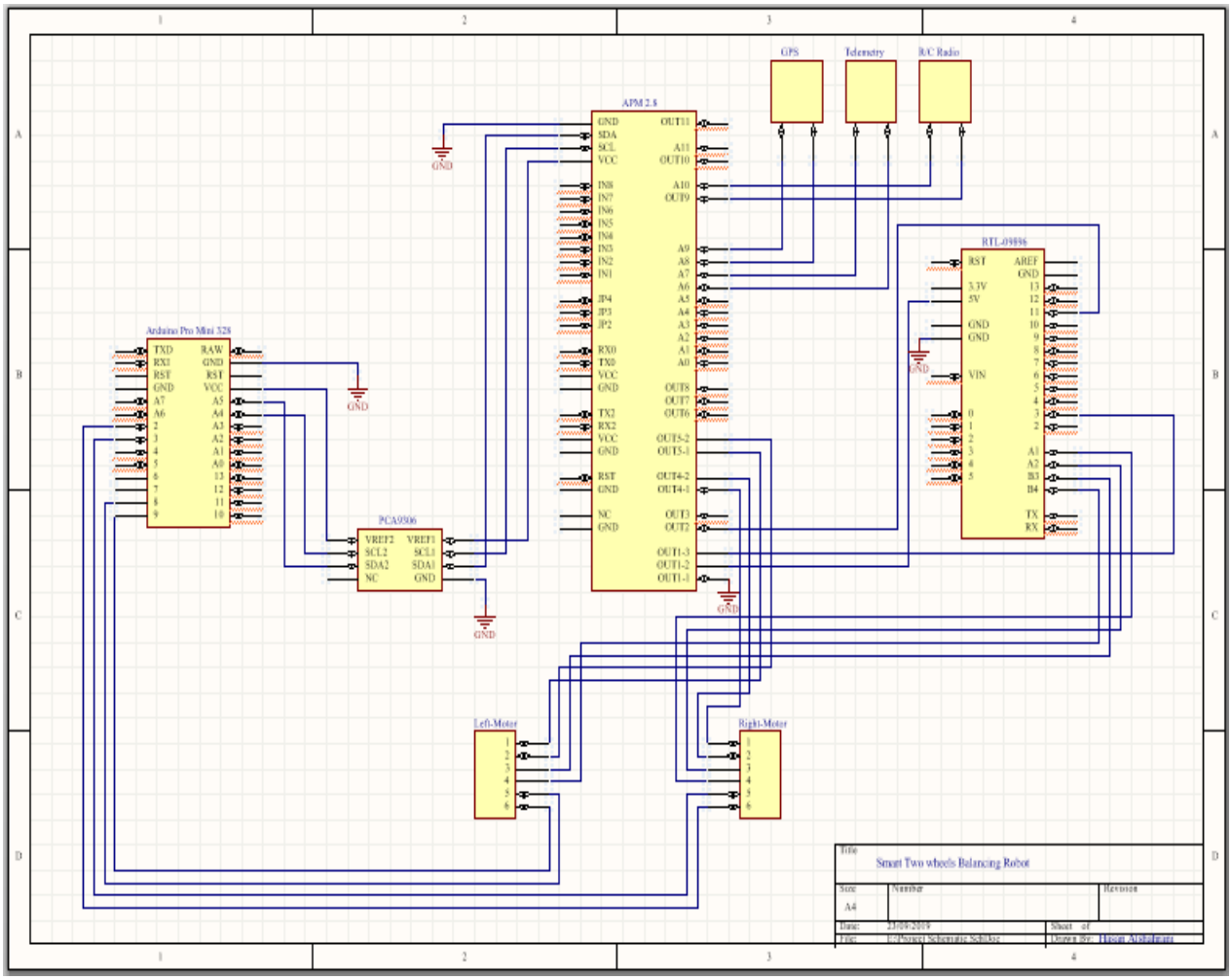
Appendix E: Wheel Encoders Code (Arduino Pro Mini)

```
#include <Wire.h>

#define ADDR_I2C 0x29 // the i2c address

static double wheel_left_velocity, wheel_right_velocity; // wheel velocity value
volatile int16_t wheel_left_ticks, wheel_right_ticks; // wheel encoder
static uint32_t prev_timer; // for calculating delta time

// this is the struct of data which must be sent through i2c to APM
struct{
        // the left and right wheel velocity
        int16_t wheel_left_velocity;
        int16_t wheel_right_velocity;
        // the left and right ticks number of encoder
        int16_t wheel_left_ticks;
        int16_t wheel_right_ticks;
        uint8_t bytes[]; // the buffer to be send to APM
}data;

void setup()
{
        pin_setup(); // setup pins to read the encoder
        // initialize i2c hardware
        Wire.begin(ADDR_I2C);
        // initialize the i2c event/callback
        Wire.onRequest(requestEvent);
        Wire.onReceive(receiveEvent);
}

void loop()
{
        // nothing to do, the loops are in ISR
}

void requestEvent()
{
   uint32_t timer = micros(); // read current time stamp

        data.wheel_left_ticks = wheel_left_ticks; // copy wheel left encoder ticks
        data.wheel_right_ticks = wheel_right_ticks; // copy wheel right encoder ticks
   wheel_left_ticks = 0; // clear wheel left encoder ticks
   wheel_right_ticks = 0; // clear wheel left encoder ticks

        double delta_time = (double)(timer - prev_timer)/1.0e6f; // calculate delta time in secs
        delta_time = minimum_value(delta_time, 0.15); // limit the delta time
```

```
        prev_timer = timer; // save current timer to be previous timer

        // find velocity of rotation and averaging
        wheel_left_velocity = (wheel_left_velocity + ((double)data.wheel_left_ticks /
delta_time))/2.0;
        wheel_right_velocity = (wheel_right_velocity + ((double)data.wheel_right_ticks /
delta_time))/2.0;

        // assign the value to the i2c buffer data
        data.wheel_left_velocity = wheel_left_velocity;
        data.wheel_right_velocity = wheel_right_velocity;

        Wire.write(data.bytes, sizeof(data)); // write the data through i2c to APM
}

void receiveEvent(int recv_data)
{
        // no received i2c
}

void pin_setup() // setup pins to input for reading the encoder and activate two interrupts
{
        // connection to encoder wheel left
        pinMode(2, INPUT); // PIND2
        pinMode(3, INPUT); // PIND3
        // connection to encoder wheel right
        pinMode(8, INPUT); // PINB1
        pinMode(9, INPUT); // PINB2

        // initialize ISR interrupt
        PCMSK2          = _BV(PCINT18);
        PCMSK0          = _BV(PCINT0);
        PCICR   = _BV(PCIE0) | _BV(PCIE2);
}

ISR(PCINT0_vect) // the interrupt trigger/callback for i2c which connects to the right wheel
{
        // increase or decrease the ticks according to the input in PINB1 and PINB2
        if(((PINB & 0x01) == 0 && (PINB & 0x02) == 0) || ((PINB & 0x01) == 1 && (PINB & 0x02) > 1))
                wheel_right_ticks++;
        else
                wheel_right_ticks--;
}

ISR(PCINT2_vect) // the interrupt trigger/callback for i2c which connects to the right wheel
{
        // increase or decrease the ticks according to the input in PIND2 and PIND3
        if(((PIND & 0x04) == 0 && (PIND & 0x08) == 0) || ((PIND & 0x04) > 1 && (PIND & 0x08) > 1))
```

```
                wheel_left_ticks++;
        else
                wheel_left_ticks--;
}

double minimum_value(double a, double b) // calculate the minimum value of two variables
{
        if(a < b)
                return a;
        else
                return b;
}
```

Appendix F: Schematic Circuit



References

1. Alarfaj, M. & Kantor, G., 2011. *Centrifugal force compensation of a two-wheeled balancing robot.* Singapore, IEEE.

2. Almeshal, A. M., Goher, K. M. & Tokhi, M. O., 2013. Dynamic modelling and stabilization of a new configuration of two-wheeled machines. *Robotics and Autonomous Systems,* 61(5), pp. 443-472.

3. An, W. & Li, Y., 2014. *Simulation and control of a two-wheeled self-balancing robot.* Shenzhen, IEEE.

4. Chan, R., Stol, K. A. & Halkyard, R., 2013. Review of modelling and control of two-wheeled robots. *Annual Reviews in Control,* 37(1), pp. 89-103.

5. Chiu, C., Lin, Y. & Lin, C., 2011. Real-time control of a wheeled inverted pendulum based on an intelligent model-free controller. *Mechatronics,* 21(3), pp. 523-533.

6. Dai, F. et al., 2015. A two-wheeled inverted pendulum robot with friction compensation. *Mechatronics,* 30(1), pp. 116-125.

7. Gonzalez, C., Alvarado, I. & Pena, M., 2017. Low-cost two-wheels self-balancing robot for control education. *IFAC,* 50(1), pp. 9174-9179.

8. Juang, H. & Lurrr, K., 2013. *Design and control of a two-wheel self-balancing robot using the Arduino microcontroller board.* Hangzhou, IEEE.

9. Lee, H. & Jung, S., 2012. Balancing and navigation control of a mobile inverted pendulum robot using sensor fusion of low-cost sensors. *Mechatronics,* 22(1), pp. 95-105.

10. Lee, J. H., Shin, H. J., Lee, S. J. & Jung, S., 2013. Balancing control of a single-wheel inverted pendulum system using air blowers: Evolution of Mechatronics capstone design. *Mechatronics,* 23(8), pp. 926-932.

11. Miasa, S., Al-Mjali, M., Ibrahim, A. & Tutunji, T., 2010. *Fuzzy control of a two-wheel balancing robot using DSPIC.* Amman, IEEE.

12. Romlay, R., Ibrahim, A., Toha, S. & Rashid, M., 2019. Two-wheel Balancing Robot; Review on Control Methods and Experiments. *International Journal of Recent Technology and Engineering (IJRTE),* 7(6), pp. 106-112.

13. Su, K., Chen, Y. & Su, S., 2010. Design of neural-fuzzy-based controller for two autonomously driven wheeled robots. *Neurocomputing,* 73(13), pp. 2478-2488.

14. Sun, L. & Gan, J., 2010. *Researching of Two-Wheeled Self-Balancing Robot Base on LQR Combined with PID.* Wuhan, IEEE.

15. Thao, N., Nghia, D. & Phuc, N., 2010. *A PID backstepping controller for a two-wheeled self-balancing robot.* Ulsan, IEEE.

16. Unluturk, A., Aydogdu, O. & Guner, U., 2013. *Design and PID control of two-wheeled autonomous balance robot.* Ankara, IEEE.

17. Wu, J., Zhang, W. & Wang, S., 2012. A Two-Wheeled Self-Balancing Robot with the Fuzzy PD Control Method. *Mathematical Problems in Engineering,* 1(1), pp. 1-13.

18. Yau, H., Wang, C., Pai, N. & Jang, M., 2009. *Robust Control Method Applied in Self-Balancing Two-Wheeled Robot.* Wuhan, IEEE.

19. Yim, E., Lee, S., Lee, Y. & Kim, S., 2018. *Optimal Outer-Loop Position Controller for Two-Wheeled Mobile Balancing Robot Based-on Off-Line optimization Technique.* Daegwallyeong, IEEE.

20. Aguilar-Acevedo, F & Alejo, VG 2014, 'Using the open-source platform for trajectory control of DC motors', *IEEE International Autumn Meeting on Power Electronics and Computing (ROPEC) 2013*, pp. 1-5.

21. Ali, E & Aphiratsakun, N 2016, 'AU ball on plate balancing robot', *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2015, pp. 2031-2034.

22. Ateov, S, Kwon, K, Lee, S & Moon, K 2017, 'Data analysis of the MAVLink communication protocol', *2017 International Conference on Information Science and Communications Technologies (ICISCT)*, 2017, pp. 1-3.

23. Azar, AT, Ammar, HH, Barakat, MH, Saleh, MA & Abdelwahed, MA 2018, 'Self-balancing robot modelling and control using two degrees of freedom PID controller', *International Conference on Advanced Intelligent Systems and Informatics*, vol. 1, no. 1, pp. 64-76.

24. Azimi, MM & Koofiger, HR 2013, 'Model predictive control for a two-wheeled self-balancing robot', *2013 First RSI/ISM International Conference on Robotics and Mechatronics (ICRoM), Tehran*, pp. 152-157.

25. Balung, M, Spoljarnic, T & Vujicic, G 2017, 'A laboratory model of the elevator controlled by ARDUINO platform', *40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1562-1565.

26. Binugroho, E, Pratama, D, Syahputra, A & Pramadihanto, D 2016, 'Control for balancing line follower robot using discrete, cascaded PID algorithm on ADROIT V1 education robot', *2015 International Electronics Symposium (IES)*, pp. 245-250.

27. Cancharoen, R, Sripakagorn, A & Maneeratana, K 2015, 'An Arduino kit for learning mechatronics and its scalability in semester projects', *2014 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE), Wellington*, pp. 505-510.

28. Jamil, O., Jamil, M., Ayaz, Y & Ahmad, K 2014, *'Modeling, control of a two-wheeled self-balancing robot', 2014 International Conference on Robotics and Emerging Allied Technologies in Engineering (iCREATE), Islamabad*, pp. 191-199.

29. Asali, M, Hadary, F & Sanjaya, B 2017, 'Modeling, Simulation, and Optimal Control for Two-Wheeled Self-Balancing Robot', *International Journal of Electrical and Computer Engineering*, vol. 7, no. 4, viewed 2 March 2020, <https://www.researchgate.net/publication/320248792_Modeling_Simulation_and_Optimal_Control_for_Two-Wheeled_Self-Balancing_Robot>.

30. Kankhunthod, K, Kongratana, V, Numsomran, A, & Tipsuwanporn, V 2019, 'Self-balancing Robot Control Using Fractional-Order PID Controller ', *International MultiConference of Engineers and Computer Scientists*, vol. 7, no. 4, viewed 5 March 2020, < https://www.semanticscholar.org/paper/Self-balancing-Robot-Control-Using-Fractional-Order-Kankhunthod-Kongratana/afb8171cc6330428eeca44a9dba0981dea6b6e64 >.

31. Chhotary, A, Pradhan, MK, Pandey, KK & Parhi, DR 2016, 'Kinematic analysis of a two-wheeled self-balancing mobile robot', *Proceedings of the International Conference on Signal, Networks, Computing, and Systems*, vol. 1, no. 1, pp. 87-93.

32. Dai, F, Li, F, Bai, Y, Guo, W, Zong, C & Gao, X 2012, 'Development of a coaxial self-balancing robot based on sliding mode control', *2012 IEEE International Conference on Mechatronics and Automation, Chengdu*, pp. 1241-1246.

33. Esmaeili, N, Alfi, A & Khosravi, H 2017, 'Balancing and trajectory tracking of two-wheeled mobile robot using backstepping sliding mode control: Design and experiments', *Journal of Intelligent & Robotic Systems*, vol. 87, no. 1, pp. 601–613.

34. Ferdinando, H, Khoswanto, H & Tjokro, S 2011, 'Design and evaluation of two-wheeled balancing robot chassis', *2011 International Conference on Communications, Computing and Control Applications (CCCA), Hammamet*, pp. 1-6.

35. Ghaffari, A, Shariati, A & Shamekhi, AH 2016, 'A modified dynamical formulation for two-wheeled self-balancing robots', *Nonlinear Dynamics*, vol. 83, no. 1, pp. 217–230.

36. Ghani, M, Naim, F & Yon, T 2011, 'Two wheels balancing robot with line following capability', *International Journal of Mechanical and Mechatronics Engineering*, vol. 5, no. 7, pp. 1401-1405.

37. Han, HY, Han, T & Jo, H 2015, 'Development of omnidirectional self-balancing robot', *2014 IEEE International Symposium on Robotics and Manufacturing Automation (ROMA), Kuala Lumpur,* pp. 57-62.

38. Hao, Q, Li, P, Ze Chang, Y & Yang, F 2011, 'The fuzzy controller designing of the self-balancing robot', *Proceedings of 2011 International Conference on Electronics and Optoelectronics, Dalian, 2011,* pp. V3-16-V3-19.

39. Hoang, P, Drieberg, M & Nguyen, C 2014, 'Development of a vehicle tracking system using GPS and GSM modem', *2013 IEEE Conference on Open Systems (ICOS), Kuching,* pp. 89-94.

40. Hsu, C & Sheen, W 2011, 'Joint Calibration of Transmitter and Receiver Impairments in Direct-Conversion Radio Architecture', *IEEE Transactions on Wireless Communications*, vol. 11, no. 2, pp. 832-841.

41. Warren, J.D., Adams, J. and Molle, H., 2011. Arduino for robotics. In *Arduino robotics* (pp. 51-82). Apress, Berkeley, CA.

42. Jiang, L, Qiu, H, Wu, Z & He, J 2016, 'Active disturbance rejection control based on adaptive differential evolution for a two-wheeled self-balancing robot', *2016 Chinese Control and Decision Conference (CCDC), Yinchuan,* pp. 6761-6766.

43. Kim, S & Kwon, S 2015, 'Dynamic modelling of a two-wheeled inverted pendulum balancing mobile robot', *International Journal of Control, Automation, and Systems,* vol. 13, no. 1, pp. 926–933.

44. Kim, S, Seo, J & Kwon, S 2011, 'Development of a two-wheeled mobile tilting & balancing (MTB) robot', *2011 11th International Conference on Control, Automation and Systems, Gyeonggi-do,* pp. 1-6.

45. Kongratana, V, Gulphanich, S, Tipsuwanporn, V & Huantham, P 2012,'Servo state feedback control of the self-balancing robot using MATLAB', *12th International Conference on Control, Automation, and Systems, JeJu Island,* pp. 414-417.

46. Madhira, K, Gandhi, A & Gujral, 2016, 'Self-balancing robot using complementary filter: Implementation and analysis of complementary filter on SBR' *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT),* pp. 2950-2954.

47. Mahler, B & Haase, J 2013, 'Mathematical model and control strategy of a two-wheeled self-balancing robot', *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society, Vienna,* pp. 4198-4203.

48. Majczak, M & Wawrzynski, P 2015, 'Comparison of two efficient control strategies for two-wheeled balancing robot', *20th International Conference on Methods and Models in Automation and Robotics (MMAR), Miedzyzdroje, 2015*, pp. 744-749.

49. Martins, R & Nunes, F 2017, 'Control system for a self-balancing robot', *4th Experiment@International Conference (exp.at'17), Faro*, pp. 297-302.

50. Memarbashi, HR & Chang, JY 2011, 'Design and parametric control of co-axes driven two-wheeled balancing robot' *Microsystem Technologies*, vol. 17. no. 1, pp. 1215–1224.

51. Muhammed, M, Buyamin, S, Ahmad, MN & Nawawi, SW 2011 'Dynamic Modeling and Analysis of a Two-Wheeled Inverted Pendulum Robot', *2011 Third International Conference on Computational Intelligence, Modelling & Simulation*, Langkawi, pp. 159-164.

52. Parcito, B 2016, 'Ensemble Kalman filter and PID controller implementation on the self-balancing robot', *2015 International Electronics Symposium (IES), Surabaya*, pp. 105-109.

53. Peng, K, Ruan, X & Zuo, G 2012, 'Dynamic model and balancing control for the two-wheeled self-balancing mobile robot on the slopes', *Proceedings of the 10th World Congress on Intelligent Control and Automation, Beijing*, pp. 3681-3685.

54. Prakash, K & Thomas, K 2017, 'Study of controllers for a two-wheeled self-balancing robot', *2016 International Conference on Next Generation Intelligent Systems (ICNGIS)*, Kottayam, pp. 1-7.

55. Pratama, D, Ardilla, F, Binugroho, E & Pramadihanto, D 2015, 'Tilt set-point correction system for balancing robot using PID controller', *2015 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC), Bandung*, pp. 129-135.

56. Pratama, D, Bunigroho, E & Ardilla, F 2016, 'Movement control of two wheels balancing robot using a cascaded PID controller', *2015 International Electronics Symposium (IES), Surabaya*, pp. 94-99.

57. Rahman, M, Rashid, H & Hossain, M 2018, 'Implementation of Q learning and deep Q network for controlling a self-balancing robot model', *Robotics and Biomimetic*, 5(8), pp. 33-45.

58. Riattama, D., Binugroho, E., Dewanto, R & Pramadihanto, D 2017, 'PENS-wheel (one-wheeled self-balancing vehicle) balancing control using PID controller', *2016 International Electronics Symposium (IES), Denpasar*, pp. 31-36.

59. Ruan, X & Li, W 2014, 'Ultrasonic sensor based two-wheeled self-balancing robot obstacle avoidance control system', *2014 IEEE International Conference on Mechatronics and Automation, Tianjin*, pp. 896-900.

60. Sadeghian, R & Masoule, M 2016, 'An experimental study on the PID and Fuzzy-PID controllers on a designed two-wheeled self-balancing autonomous robot',

61. Samson, N, Dumont, S, Specq, M & Praud, J 2011, 'Radio telemetry devices to monitor breathing in non-sedated animals', *Respiratory Physiology & Neurobiology*, vol. 179, no. 3, pp. 111-118.

62. Sun, F., Yu, Z & Yang, H 2015, 'A design for a two-wheeled self-balancing robot based on Kalman filter and LQR', *2014 International Conference on Mechatronics and Control (ICMC), Jinzhou*, pp. 612-616.

63. Su, X., Wang, C., Su, W & Ding, Y 2016, 'Control of balancing mobile robot on a ball with fuzzy self-adjusting PID', *2016 Chinese Control and Decision Conference (CCDC), Yinchuan*, pp. 5258-5262.

64. Wang, X., Chen, S., Chen, T & Yang, B 2016, 'Study on control design of a two-wheeled self-balancing robot based on ADRC', *2016 35th Chinese Control Conference (CCC), Chengdu*, pp. 6227-6232.

65. Wardoyo, A. S. et al. 2015, 'An investigation on the application of fuzzy and PID algorithm in the two-wheeled robot with a self-balancing system using microcontroller', *2015 International Conference on Control, Automation and Robotics, Singapore*, pp. 64-68.

66. Wu, J., Liang, Y & Wang, Z 2011, 'A robust control method of two-wheeled self-balancing robot', *Proceedings of 2011 6th International Forum on Strategic Technology, Harbin, Heilongjiang*, pp. 1031-1035.

67. Wu, J & Wanying, Z 2011, *'research on Control Method of Two-wheeled Self-balancing Robot', 2011 Fourth International Conference on Intelligent Computation Technology and Automation, Shenzhen, Guangdong*, pp. 476-479.

68. Yuan, S., Lei, G & Bing, X 2016, '*Dynamic modelling and sliding mode controller design of a two-wheeled self-balancing robot', 2016 IEEE International Conference on Mechatronics and Automation, Harbin*, pp. 2437-2442.

69. "Rover Home — Rover documentation", Ardupilot.org, 2020. [Online]. Available: https://ardupilot.org/rover/index.html. [Accessed: 08- May- 2020].

70. Encoder, 3., 2020. 34:1 Metal Gearmotor 25Dx52l Mm HP With 48 CPR Encoder. [online] Core Electronics. Available at: < https://core-electronics.com.au/34-1-metal-gearmotor-25dx52l-mm-hp-with-48-cpr-encoder.html > [Accessed 13 May 2020].

71. "Open Source Drone Software. Versatile, Trusted, Open. ArduPilot.", Ardupilot.org, 2020. [Online]. Available: https://ardupilot.org/. [Accessed: 08- May- 2020].