Contents

1 Introduction

One of the main specifications for the project is that the robot designed should be able to navigate the laboratory autonomously. It has been decided that the robot will use ROS and SLAM to navigate the laboratory. Therefore, research has been conducted into the hardware used for the previous implementation of SLAM on other similar robotic systems. This hardware bridges the gap between the SLAM algorithm and the controlling of the motors. This chapter focuses on: The hardware selected to efficiently run ROS and allow for full implementation of SLAM, the installation of ROS specific hardware into the robot's power system, programming using the ROS API and the design of a motor control system. Additive manufacturing has been used to throughout the design and assembly of the robot's chassis. Setup parameters will have to be tuned to allow accurate movement of the robot in a lab environment. Real-world testing has been detailed to display the robot's response to a complete coverage solution.

2 Background literature

2.1 ROS

To understand how ROS works, which is required for hardware selection, the ROS Robot Programming manual has been studied [1]. This manual is an excellent guide for a beginner in ROS and has built a solid foundation of knowledge for this project. The ROS wiki webpage has been vital for finding officially supported hardware and tutorials for setting up the software.

2.2 Market Research

The first and most important piece of research was looking at other projects with similar applications and design specifications. The CARMA platform developed at the University of Manchester is the most similar platform to this application [2].

TurtleBot 3 would be a great platform to learn ROS, but it is expensive. Therefore, independently building a very similar platform but with design modifications to fit the application would be less expensive. This approach was taken by matop_m [3], where they built their own Turtlebot 3 Waffle for 1/6 of the price.

There are many examples of using ROS on a Raspberry Pi and Arduino. This approach has been used by Shaoul [4], where Hector-SLAM has been successfully implemented using a Raspberry Pi 3 and an Arduino Uno. It has been vital to find similar project examples that use SLAM and which hardware is used. This supports the purchasing of the same hardware for this project as there is evidence of its capability of running SLAM algorithms.

Market research helps to identify pre-existing problems with the hardware used by others and methods used to fix them. This is shown by Robin Baran [5], where they have used the Raspberry Pi to publish the odometry topic as the Arduino does not have enough RAM.

2.3 Additive Manufacturing

A considerable amount of time and research into additive manufacturing has been conducted before starting the project. Operating a 3D printer has enabled the lead time for prototyping to be reduced significantly. Knowledge of CAD modelling for additive manufacturing has been used to adapt designs before they were even printed. Experience in using fusion 360 enabled more complex designs to be created when they were not available online.

3 Work Conducted

3.1 ROS Specific Robot Hardware

3.1.1 Single board computer (SBC)

The robot requires a computer that is powerful enough to run SLAM calculations. This computer has to have a small form factor and a low power demand. An SBC is perfect for this application as it can have the features of a standard computer while also costing less. They can be passively cooled, reducing the amount of maintenance required.

An Ubuntu Linux 18.04 image has to be available for the selected SBC, allowing it to use ROS Melodic, the chosen version of ROS for this project. Even though ROS Melodic is not the newest version of the operating system, it supports the most ROS packages. All of the similar ROS robot projects use this version of ROS, which has enabled quick installation of the software. It has been available since 2018, meaning there is more community support for this version making it faster to debug any problems.

The SBC used for the robot is the Raspberry PI 4 [6]. This has similar processing power to equivalent SBCs on the market but at a fraction of the price [7]. Raspberry Pi's are the most popular SBC on the market, meaning there is lots of support for using them in ROS robots.

The newest version, Pi 4, comes with a Cortex-A72 processor, which is excellent at "high-frequency loop executions" for collision avoidance and "heavy computations" for route planning [8]. As it has a 64-bit architecture, there is the option of having up to 8GB RAM. There is also an Ubuntu Linux 18.04 image available for it. Only the 4GB version of Pi 4 is required for the robot; when running SLAM simulations on a desktop virtual machine, no more than 3GB of RAM was used. The Nvidia Jetson Nano has much greater graphical processing power than the Raspberry Pi 4. However, the graphical processing power of the Raspberry Pi 4 is sufficient as Visual SLAM is not being used for reasons outlined on page 42.

A Pi 3 was loaned from the University of Leeds. This allowed for the implementation of Ubuntu and ROS to be tested on the platform before purchasing the Pi 4. The Pi 4 has lots of online support and guides, enabling faster implementation of ROS onto the SBC. A 12V to 5V step-down buck converter, specifically designed to power the Pi, was used to connect it to the battery.

3.1.2 Microcontroller

As the Raspberry Pi 4 uses most of its computational power to run SLAM calculations, a separate microcontroller was used for low-level control of the robot. The Geiger counter outputs the audio signal as detailed on page 13. The Pi does not have any analogue inputs. Therefore, a separate microcontroller provided the required I/O pins for the project.

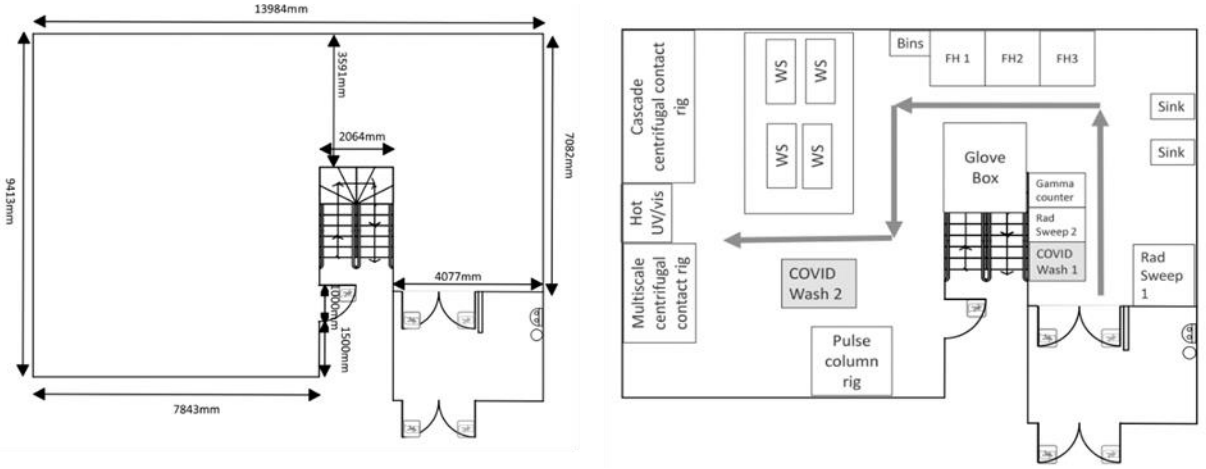
The Arduino Mega 2560 was the microcontroller used for this project [9]. This is the same microcontroller used by matop_m [3], Shaoul [4], and Baran [5] in their ROS robot projects, which enabled the fast setup of the hardware. The Arduino does not have an operating system. Therefore it is more stable than the Pi for running motor control programs. Similar to the Raspberry Pi, the Arduino Mega 2560 is one the most used microcontroller platforms. This means there were many guides to follow when setting up the ROS serial package for communication between Arduino and Pi. Serial communication is set up over a custom length USB cable which also powers the Arduino at the same time. This reduces the number of power cables required in the robot, simplifying the design of the power system. Even though a Teensy microcontroller may have been a better choice for this project, an Arduino Mega 2560 could be loaned from the University of Leeds. The Teensy has little online support, and purchasing it would have been a risk, as it was not guaranteed that ROS could be set up on it.

3.1.3 LIDAR Sensor

As detailed on page 42 the robot uses a LIDAR sensor to enable implementation of SLAM. As Visual SLAM is not being used for the reason's details on 42 the LIDAR sensor is the primary source of data for localising the robot and the only source of data object avoidance.

The LIDAR sensor used on the robot is RPLIDAR A1[10]. This LIDAR is supported with an official ROS package causing it to have a very stable interface compared to other LIDARs with unofficial community-built packages. This is the most popular LIDAR sensor used in other ROS robot projects meaning there was much guidance on implementing the LIDAR with a Raspberry Pi 4. The RPLIDAR A1 has a class 1 safety standard, so it is safe to use around humans. The lab environment that the robot is in has bright lights, and the RPLIDAR A1 is capable of working in these conditions. This also meant that during hardware setup and testing, the lights could remain on.

Figure 1: GM59 LNL Lab Floor Plans



: Provided by Jerry Lee

From looking at Figure 1, the maximum line of sight that the robot will have is 9.413 m. As the RPLIDAR has a range of 0.15m to 12 m larger than this distance, it is expected that the robot will better location results and be quicker at mapping [11]. This is because it always has data on the other side of the lab, no matter its position. As LIDAR has a minimum range of 0.15 m, any parts of the robot chassis will be ignored and not affect the scanning data. During testing, this was confirmed as the robot did not need to be driven as far to map the whole room, which saves battery life. The RPLIDAR also has an excellent sampling rate for its low price, which helps to improve the mapping accuracy [12]. The RPLIDAR also has an excellent distance accuracy of 1% at all ranges, which helps the mapping accuracy. This meant the maps produced of the lab had a high resolution enabling quick localisation. As the lab that the robot is scanning has lab benches with flat surfaces and no table legs, a clean map of the lab can be produced from the LIDAR data.

3.1.4 Motor Type for Differential Drive

The most common motor type used in ROS differential drive robots is a geared DC motor with encoders. Bowling [13] explained that geared DC motors are cheap and have high torque at low speed, which is what this project requires. They can also be found with quadrature encoders already attached to the motors housing. This means the motors output shaft can be measured by the encoder rather than the gear box's shaft. This allows for a more accurate measurement of the motor velocities by the encoders as the backlash in the gears does not affect the results. The motor velocities must be measured accurately as these are used to calculate the linear and angular velocities of the robot for differential drive. These velocities are then used to calculate the odometry data of the robot, which is an estimate of far the robot's position has changed with time. This information of the best motor type was then used to select the specific motors for the robot.

This robot is using a differential drive which is detailed on page 45. A four-wheel-drive system was considered to help stop the movement of the arm, causing the robot to rotate about its vertical axis.

To turn on the spot with the four-wheel-drive the robot would have to use skid steer. It is possible to implement skid steer using the ROS diff_drive_controller package. However, the skid steer part of the package does not allow direct skid steer commands and has many reported issues. It was also found by Üveges, Ďurovský, and Fedák, [13] that skid steer movements are not accurate enough and a robot would require visual odometry data to enable accurate localisation estimates. As the robot is not using Visual SLAM, a two-wheel drive would be used instead of a four-wheel drive as skid steer is not accurate enough. There are also no ROS project examples of low-cost skid steer robots, which would lead to a longer implementation time if this drive method was used.

3.1.5 Inertial Measurement Unit IMU

The robot uses the wheel velocities to work out how far it has moved and, therefore, positioned relative to its starting position. However, the wheel velocity data can be misinterpreted. If the wheels slip due to poor grip on some surfaces, the robot will not actually move as far as it thinks. This causes the robots localisation estimate to be very poor as the changing LIDAR data does not match the movement distance calculated using the wheel velocities. This was found especially when testing the robot on the carpet as the casters would catch as the robot rotated. Therefore to improve the accuracy of the localisation estimate, an IMU is used. An IMU can measure the actual linear acceleration, angular velocity and orientation of the robot. The data from the IMU is not affected by the wheels slipping. The IMU sensor data can be fused with the LIDAR sensor data and wheel velocities [14]. This helps to reduce the error in localisation estimate as there is another input source of data [15].

This robot uses a 9-axis IMU. A 6-axis IMU has an accelerometer and gyroscope. A 9-axis IMU adds a magnetometer to this, which enables the robot's orientation to measured using the earth's magnetic field [16]. The change in orientation of the robot can be calculated from the gyroscope data. However, using a magnetometer to calculate the orientation was found by Schön, Hol and Kok [17] to be more accurate and helps to reduce z-axis drift over time.

The IMU selected for the robot is the MPU9250 [18], which is on the MakerHawk breakout board. As this is a 9-axis IMU, it is more readily available for purchase when compared to 6-axis IMU's, which is now an older technology. The MPU9250 is recommended by many IMU example projects, which allowed quick implementation time as the guides could be followed. Another popular IMU used by the hobbyist community is the BNO055. Both the MPU9250 and BNO055 IMU's have supporting ROS packages and Arduino libraries. It was found by Aoyagi and Choi [19] that the MPU9250 has better overall performance than the BNO055, which is why the MPU9250 was selected for this robot.

3.1.6 User Interface

To make the robot more user-friendly, it has a user interface broken into physical and digital parts. The physical user interface consists of switches and a screen; the digital user interface is detailed on page 32. The screen is used to display the IP of the digital user interface and the number and the scanning status of the robot. The chosen screen used on the robot is the Nokia 5110 LCD**.** This screen was chosen because it integrates easily with an Arduino and it has a low power demand. It is also used in the ELEC 2645 module, so the Arduino library is very well known.

3.2 Additive Manufacturing

Additive manufacturing, also known as 3D printing, has been a vital tool during the project for designing and making any structural parts. In this chapter, additive manufacturing has mainly been used for creating cases for the ROS specific hardware that allows it to be mounted to the chassis. The cases also protect the hardware and allows it to be handled without the risk of electrostatic discharging destroying any electrical components.

PETG has been the chosen filament type for all the 3D prints in this project. PLA and ABS are other popular 3D printing filaments that could have been used and are easier to print with. However, PETG was chosen as it is much stronger than both PLA and ABS, and it also has a higher temperature resistance than PLA [20]. This means that PETG has a high impact resistance required as the robot may encoder impacts during testing, so parts do not break over time.

As the robot can contact radioactive material, one of the design specifications requires it to be cleanable with Decon90 agent. The wheels and motor mounts of the are the most likely areas of the robot that become contaminated as they are closest to the ground. Another advantage of PETG is that it has a high chemical resistance to chemical agents. Therefore, if any part of the robot is contaminated, it can be cleaned with the Decon90 agent without the risk of destroying the plastic. PETG is also hydrophobic; therefore, the parts will not absorb any radioactive solutions if they were to contact it.

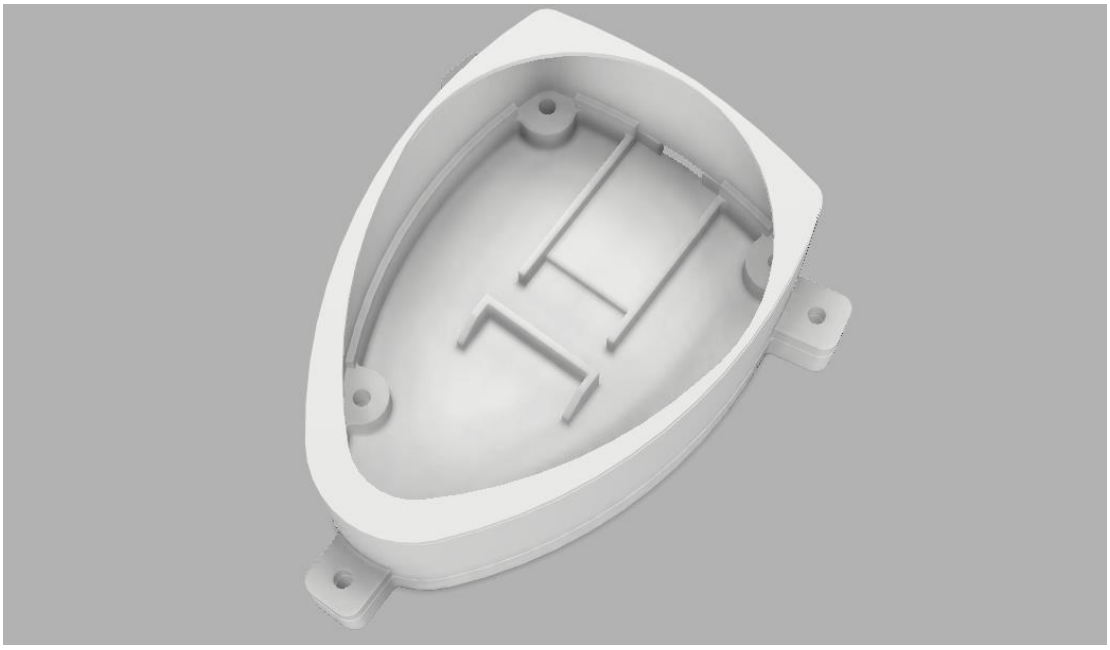
Finally, PETG is a familiar filament to print with, and the 3D printer is already set up with the correct settings to produce good quality prints. This meant parts could be printed as soon as the filament arrived, and not as much filament was wasted on prints failing due to print settings.

The ROS specific hardware selected is also the most popular hardware in the electronics hobbyist community. This meant many protective case designs already existed on online 3D printing sharing platforms. Which saved loads of time as the cases for the Pi designed by Ba [21], Arduino designed by 3DRay [22] and LCD case could be used instead of designing new ones. Using already designed cases also saves lots of filament as the designs are guaranteed fit and designs will not need changing then reprinting. This specific Pi case was chosen as it enables heatsinks and fan to attached to cool the pi if it gets too hot. It turned out that whilst running SLAM software, the Pi 4's CPU reached close to 80 °C after a few minutes. The Pi 4 thermal throttles at 80 °C, meaning it will turn down the clock speed of the CPU to cool down. This causes the Pi to slow down and causes the robot movement and command response to become very slow. The heatsinks and a small 5 V fan powered from the Pi have been added to the case. This keeps the Pi cool and stops it from thermal throttling. The cooling reduced CPU temperatures during testing to 35 °C**.**

Figure 2: RPLIDAR Case with LIDAR

An RPLIDAR case design could already be found online. However, the CAD designs were not free and would cost £4. Therefore a new case was modelled using Fusion360. The new case design was based on the PRLIDAR case design by HGROSS [23]. The final design for the RPLIDAR case can be seen in Figure 2. The case was designed around an accurate model of the RPLIDAR form NetBUG [24], shown in black. This meant the case only had printed once, and it fit the first time. The case makes it possible to mount the LIDAR sensor and also protects it from dust build-up.

Figure 3: Inside RPLIDAR Case



The inside of the case can be seen in Figure 3. This shows a mounting spot for the UART serial to USB adapter to sit. This keeps all the cables inside the case and out the way and prevents the risk ESD of damaging the circuit boards. The LIDAR connects to Pi using a micro-USB to USB A cable. There is a hole in the side of the case to enable the micro-USB cable to be plugged into the adapter. There are risers inside the case for the LIDAR legs to sit on as they are not long enough to make enough clearance for the serial cable.

The hardest part of the robot to 3D print was the plastic coupler shown on page 25. It had to be redesigned multiple times for it the fit the wheel and new metal coupler. Owning a 3D printer made it possible for all the design changes to be made within one day. It also saved much time whilst prototyping the chassis design. As the University's 3D printers were used, there would have been a two-day lead on each print. It was found that a plastic coupler had to be printed on its side, which increased print time. This was because the locking nuts of the metal coupler would split the 3D printed layers apart when tightening them. Therefore, the layers were printed perpendicular to the locking nut, so the layers were compressed together rather than split apart.

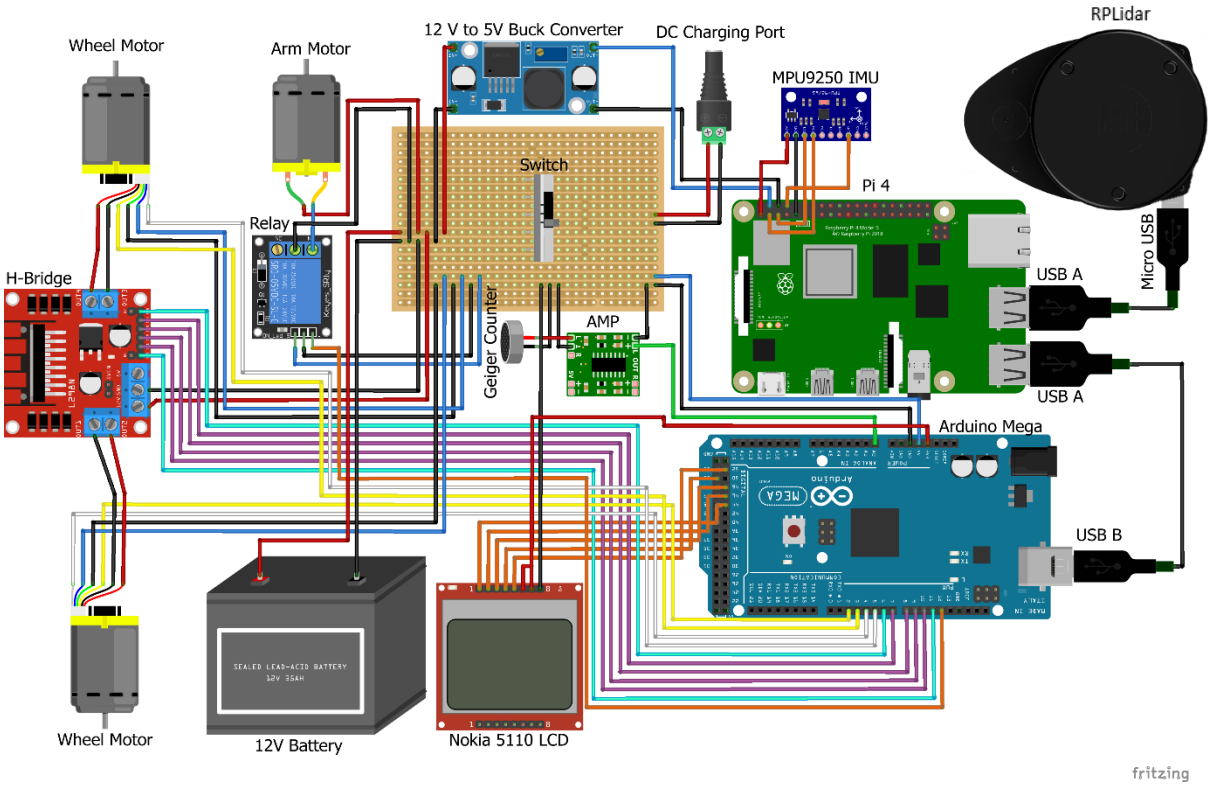
3.3 Hardware Setup

As a custom robot platform was built, which could run ROS, a lot of time was spent connecting all the hardware and testing it. Designing the platform instead of buying one meant it was much quicker to integrate the arm and Geiger counter hardware into the robot. As the robot could be designed around these parts and a better understanding of the system is obtained.

3.3.1 Schematic

Before purchasing any hardware, a circuit schematic was modelled using Circuito to check that all the parts were compatible, and that the Arduino mega had enough I/O ports. However, as there were many design changes and new parts were added to the robot, the circuit schematic was updated. The updated circuit schematic for the robot can be found in Figure 4 and was designed using Fritzing. Fritzing is a more robust circuit schematic drawer than Circuito. It has many more parts in its catalogue, and it allows people to design their own custom parts. This was very useful as many of the parts in Figure 4 were design by community members as they did not exist in the catalogue. Fritzing also enables the use of a Pi and Arduino in the same schematic, along with any USB connections. A more accurate circuit schematic could be designed as it had the exact parts used in the robot. All the selected hardware is popular within the electronic community, which meant there were many guides and videos to follow on how to wire it.

Figure 4: Robot Hardware Schematic



Wire key: **Black** = Ground | **Red** = 12 V | **Blue** = 5 V |**Yellow** = Encoder Interupt | **White** = Encoder Digital | **Green** = Analog | **Purple** = H-bridge Switches | **Cyan** = Motor PWM | **Orange** = Digital.

3.3.2 Motors, Encoders and H-Bridge

The motors are connected to an Arduino via an H-Bridge. The H-bridge supplies 12 V from the battery to the motors. It uses MOSFET switches to change the direction that the current flows through the motors, which intern changes the direction that the motors spin. These switches are connected to logic gates which are then connected to the digital outputs of the Arduino. Three digital outputs control each motor. Two of the digital outputs, shown by the purple wires in Figure 4, are set to opposite high and low values to change the motor rotation direction. The third digital output, connected by the cyan wires in Figure 4, is a pulse width modulation (PWN) signal. The PWM signal controls the speed that the motors rotate. It does this by changing the duty cycle of a rectangular wave output, which changes the average power going to the motors and their speed. The motors and H-bridge were connected to the Arduino via following a guide by Bill Jamshedji [24].

The encoders are attached to the motor output shafts. These can be used to measure the velocity and direction that the motors rotate. The encoders use Hall effect sensors that measure the motor's changing magnetic field to determine how their position changes. The Hall effect sensors are powered with a 5 V source from the Arduino, shown by the blue wires in Figure 4. Each Hall effect sensor is connected to two digital inputs of the Arduino. One of the digital inputs, shown by the yellow wires, is connected to a pin which is attached to an interrupt. When this event interrupt is triggered, the Arduino reads the output from the Hall effect sensor attached to the yellow wire. Then it reads other Hall effect sensor attached to the white wire. The time difference between when interrupts are triggered can be used to calculate the velocity of the motor. Depending on the output of the Hall effect sensor attached to the white wire, the direction that the motor is rotating can be determined. The Hall effect encoders were connected to the Arduino by following the DFRobot wiki for the purchased motors.

3.3.3 IMU Connection

The IMU is connected to the Raspberry PI's GPIO ports and uses the I^2C communication protocol. It would have been easier to connect the IMU to the Arduino as there is an MPU9250 library with excellent documentation. However, it was later found that the IMU does not have enough RAM to publish the ROS IMU sensor message, so it was moved to the PI. The Arduino library was helpful to calibrate the IMU as it automatically gave the magnetometer bias values.

To make the IMU work on PI, an MPU9250 ROS driver created by rishabhdevyadav [25] was used. This driver handles the I^2C communication between the PI and IMU. It also automatically calibrates the accelerometer and gyroscope. Then it publishes the IMU data as a data_raw topic which is subscribed to by the complementary filter node. The Pi has a higher IMU data rate than the IMU; therefore, a more accurate location estimate can be calculated.

3.3.4 Connection Testing

Initially, the hardware shown in the schematic was set up on a breadboard to allow the connections to be easily changed. Once the whole system had been tested, everything was rewired onto a Veroboard and wire connections were soldered together. This required lots of planning as there were many wires to organise within the robot's chassis. The Veroboard was then tested and debugged for correct functionality. The Arduino case lid was used to trap the wire pins to stop them from coming loose. A quick-disconnect pin header was created on a piece Veroboard so the layers could be separated and work on easily.

The connection between the Arduino and H-Bridge was tested using the Arduino example code supplied by [24]. Then the different combinations of digital output values which control H-bridge switches were linked to the actual direction that they cause the motor to rotate. The Arduino example code from the DFRobot wiki was then used to test the encoders work. The IMU and LCD screen were tested using example code from their Arduino Libraries.

The LIDAR was first tested by using the manufactures scanning software on a desktop to check it worked. The LIDARs and Arduino's connection to the raspberry pi was then tested by following a guide by s.a.alghamdi94 [26]. This guide was followed to install Ubuntu Linux and ROS on the Pi. Then it explained how to visualise the LIDAR scanning data in Rviz using the RPLIDAR ROS package. Once a LIDAR scan had to be obtained, the ROS serial Arduino package was used to test the connection between the Pi and Arduino. The package came with a ROS Arduino example code used to show the connection was set up correctly.

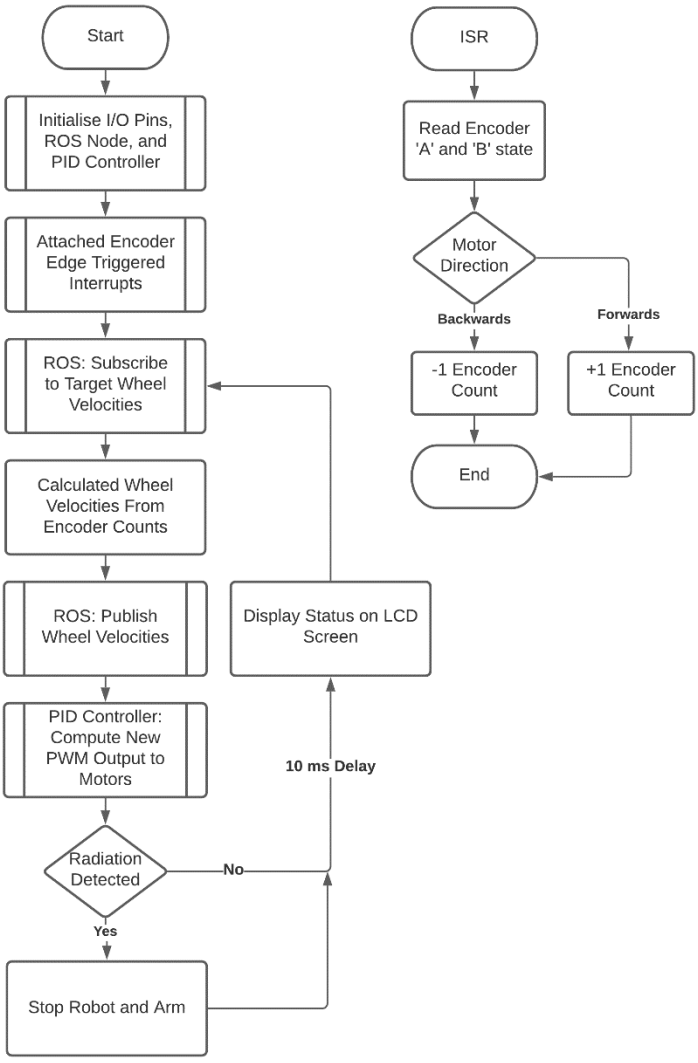
3.4 Software setup

The main advantage of building a ROS robot is that many helpful code packages can control the robot. This means that only a small amount of high-level programming is required to make the robot perform complex tasks like SLAM. The primary programming involved in this project was Arduino code, which is for low-level control of the robot. The Arduino was programmed first to check that the hardware was correctly connected. Then efforts were focused on installing Linux and all the required ROS packages on the PI.

3.4.1 Arduino

The Arduino code was created by following the same example code used to test the motors and encoders. The robot uses a PID control system library to set the velocity that motors rotate by reading the dater from the encoders. To ROS serial Arduino package is then used to transmit the motor velocities to the Pi. Similar ROS robot projects by Bruton [27] and Baron[5] were used as guidance to program the PID control system, ROS communication, and enable differential drive. Figure 5 shows how the Arduino is programmed to use ROS and the PID controller.

Figure 5: Arduino Code Flow Chart



The encoder data that is input into the Arduino is attached to an edge-triggered interrupt. This ensures all the encoder data is captured even when the Arduino is performing other calculations. Whenever there is a changing edge in the encoder data, an interrupt service routine (ISR) is triggered. The ISR shown in Figure 5 works out the direction that the motor is turning. Then it either adds or subtracts to a total encoder count, which can be used to determine the motor speed.

A ROS node is initialled on the Arduino using the ROS serial Arduino package. This node is an executable program that can communicate to other nodes on a network using ROS topics [1]. Another ROS node is set up on the Pi, enabling the Arduino and Pi to communicate over the USB connection. The Arduino node subscribes to a cmd_vel topic, a Twist message [28] type published from the move_base node running on the Pi. This topic includes the angular and linear velocities that the robot needs to drive. These velocities are calculated by the move_base node running on the Pi. Differential drive works by using these two velocities to calculate the individual left and right wheel velocities, as shown in equations 1.0 and 1.1 from the Nox_robot project [5]. The angular velocity is multiplied by half of the wheelbase, which causes the robot to rotate at one radian per second if the angular velocity is one. The Arduino then publishes the wheel velocities calculated from the encoder data in a vector topic. This topic is then subscribed to by the node running on the Pi, which uses the velocities to calculate the odometry data.

$$Left\ Wheel\ Velocity = \ Linear\ Velocity - (Angular\ Velocity \times \frac{Wheelbase}{2}) \qquad (1.0)$$

$$Right\ Wheel\ Velocity = \ Linear\ Velocity + (Angular\ Velocity \times \frac{Wheelbase}{2}) \qquad (1.1)$$

The PID controller used is an Arduino library created by Brett Beauregard [29]. This controller computes the error between the measured encoder counts (Input) and the target encoder counts (Setpoint), calculated using the velocities published from the move_base node. It then adjusts the PWM signal (Output) sent to the H-bridge to change the motor velocity. The goal of the PID controller is to make the Input equal to the Setpoint by adjusting the Output. The PID controller adjusts the output using three tuning parameters, Kp (proportional gain), Ki (integral gain) and Kd (derivative gain). These values are chosen to make the controller respond in a desired behaviour. The PID controller can behave in several ways, but generally, the tuning parameters are chosen to either give a quick response or an accurate response.

The Output of the PID controller can be adjusted from -255 to 255. Positive and negative PWM values are used to make the motor to rotate in different directions by changing the switches on H-bridge. Then absolute value of the PWM is calculate and input into the “analogwrite” function, which produces a rectangular wave of a set duty cycle. A higher absolute PWM value corresponds to a greater duty cycle and therefore motor speed. The PWM signal frequency was increased from the default 490.20 Hz to 31372.55 Hz to stop an audible noise coming from the motors.

The radiation is detected by the Arduino using the method described on page 12.If radiation is detected, the Setpoint of the PID controller is set to zero to stop the robot from moving. Because if the robot is allowed to carry on driving, it may contaminate itself and spread radiation around the lab. It is possible to get the SLAM program to avoid the detected radiation by making the radiation area an object. The ROS obstacle avoidance program would then try to plan a path around this object to avoid the radiation. The CARMA platform team at the University of Manchester proposed this method [2]. However, this is not required in this project as it is infrequent for radiation to found on the lab floor. If radiation were to be found, the robot would inform the user. The area could then be cleaned, and the robot would carry on sweeping the lab.

The Nokia 5110 LCD screen quickly integrated into the robot using its Arduino Library. It was programmed to display the charging status, detection time left and whether radiation is found. The charging status is calculated using the method detailed on page 31. The detection time remaining uses the onboard clock of the Arduino to work out how long is left of known scan time.

The main programming loop of the Arduino has a delay time of 10 ms. This was found from online recommendations and testing to be the smallest sample period that the PID control loop can run on an Arduino. When the loop time was made smaller, it reduced the encoder data noise. However, the system's input would never reach the target setpoint even with high gain values. The main loop uses the "millis" functions to produce a delay rather than "delay" function. The "millis" function is essentially used as a time-triggered interrupt. Therefore, other code like the encoder ISR and ROS node executables can still run between the 10 ms loops. This would not be possible if a delay function were used.

3.4.2 Linux

The guide by s.a.alghamdi94 [26] was followed to install Ubuntu Linux 18.04 on the Pi. Ubuntu Linux 18.04 was selected for the reasons described in section 3.1.1. Etcher to was used to flash a desktop Linux image onto a SD card to make it easier to set up the ROS code and test the hardware. Once the robot is ready with all the ROS code the server image of Linux will be installed so operating system uses less processor power and RAM is used.

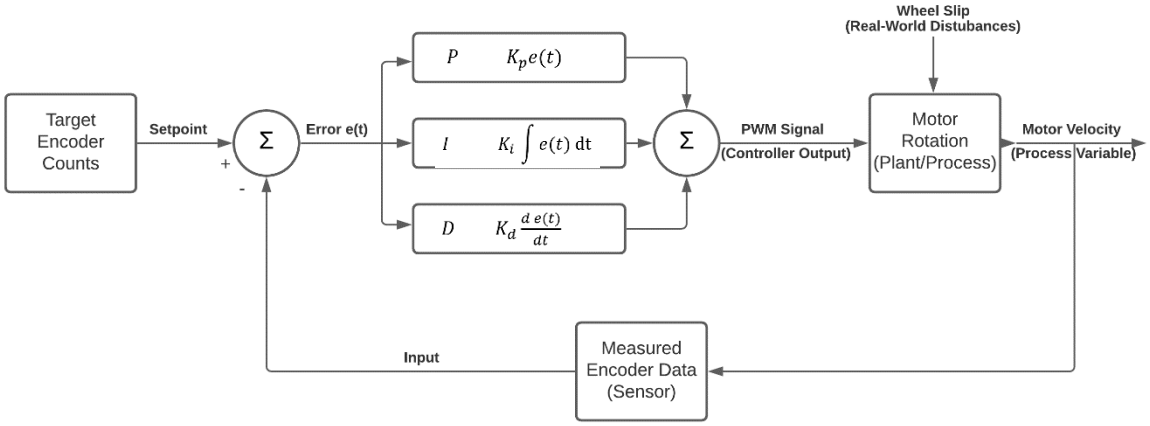
Ubuntu Linux 18.04 was then installed on a laptop using a virtual machine. This virtual machine was mainly used to SSH into the Pi. This was to run ROS launch files on the Pi when it was not plugged into a monitor and keyboard. IP addresses for the PI and virtual machine MAC addresses were then reserved in the router settings to stop the IP's from changing upon restarts.

The virtual machine could also subscribe to ROS topics published from the Pi. The data from these topics could be then visualised in Rviz so the map that the robot produces and path it plans can be seen whilst it is running. Rviz cannot be ran over SSH that is why the laptop has to subscribe to the ROS topics. For the ROS nodes to communicate over the network a Master needs to set up on one of the machines. This Master is the server for node to node communication over the network [1]. For this project the Master is set up on the robot by running the "Roscore" command. This is so a laptop is not required by the user for the robot to run. However, when a separate computer is required to visualise the robot's data, the IP of the Master machine has to be entered on the Pi and virtual machine. The .bashrc files on both machines was changed to include the master IP information so it did not have be manually entered each time.

3.5 PID Control System Tuning

The robot's encoders have to measure the motors velocities accurately to produce usable wheel odometry data. A PID control system was used as it is the most common control system type in other ROS robot projects, and it has an updated Arduino library with excellent documentation. There is also a great understanding of PID control system tuning methods because it is taught in the ELEC2540 control system module. This robot uses a closed-loop PID control system which is visualised in Figure 6. Section 3.4.1 describes how the PID control system was implemented into the Arduino code, this section details how it was tuned.

Figure 6: Closed-Loop PID Control System Flow Diagram



A PID control system uses three tuning parameters Kp, Ki and Kd, to adjust the Output. These values have been tuned to produce the fast system response, shown in Figure 13. Having a quick system response means the wheels accelerate more quickly to a set velocity. During testing of the Nox robot code, more accurate wheel odometry data was produced using a faster system response. This is because the ROS navigation package controls the robot's acceleration. Therefore, if the system had a slow response, shown in Figure 12, the actual acceleration of the robot would be different to was set in navigation settings, which decrease the accuracy of the odometry data.

The main characteristics that are looked at when tuning a closed-loop control system are the Rise time, Overshoot, Settling time and Steady-state error. Generally, Kp is used to decrease the rise time, Ki is used to decrease the steady-state error, and Kd is used to decrease the overshoot and settling time [30]. To achieve a quick system response, the PID controller needs a short rise time, a slight overshoot, a quick settling time and a small steady error. This response was achieved by the following methods outlined below.

3.5.1 Encoder data

Before any tuning could be done, the motor speeds had to be calculated from encoder data. This involved checking that the encoders were accurate to the manufacturer's specifications. Both the motors were rotated ten times, and the total number of encoder counts was recorded. The average number of encoder counter per revolution was calculated for each wheel. The measured encoder counts values were the same as the manufacturer's specifications of 2096 counts per revolution.

As the encoders were considered accurate, the number of encoders counts per one meter could be calculated using the wheel diameters. This calculation is outlined in equations 2.0 and 2.1.

$$Wheel\ Circumference = \pi \times Wheel\ Diameter = \pi \times 0.08 = 0.251\ m \quad (2.0)$$

$$Counts\ per\ Meter = \frac{Counter\ per\ Revolution}{Wheel\ Diameter} = \frac{2096}{0.251} = 8351 \quad (2.1)$$

The robot was then programmed to drive forward till 8351 encoder counts were reached. However, it was found that the robot only drove 0.48 m. This was due to disturbances in the plant, for example, the wheels slipping on the floor. Therefore, the number of encoder counts in one meter had be to measured rather than calculated. This experiment involved pushing the robot over a meter length ten times and then recording the average number of encoder counts for each wheel.

Table 1: Encoder Counts In 1 m For Each Wheel

Test Number	Left Wheel	Right Wheel
1	15975	16341
2	15831	16301
3	15906	16267
4	15856	16228
5	15720	16300
6	15952	16335
7	15846	16335
8	15783	16272
9	15810	16233
10	15988	16242
Average	**15866.7**	**16285.4**

The results in Table 1 show how inconsistent the measured number of encoder counts per one meter was for each wheel. There were very different values between each test and the two motors. As the average measured values were almost double the calculated, it is clear that the wheels were slipping on the floor. The original design of the 3D printed wheel couplers, detailed on page 25. was attached directly to the motor output shaft. Through inspection, it was found that the couplers were slipping on the motor shaft. This caused the inconstancy in the results and why the wheels had different average values. The wheel couplers were redesigned to attach to a larger metal coupler using a locking nut. The larger metal coupler then attached to the motor output shaft. The experiment above was repeated with the new coupler design.

Table 2: Encoder Counts In 1m For Each Wheel with New Couplers

Test Number	Left Wheel	Right Wheel
1	15340	15182
2	15362	15167
3	15323	15178
4	15276	15143
5	15397	15208
6	15412	15210
7	15392	15216
8	15291	15228
9	15360	15261
10	15350	15272
Average	**15350.3**	**15206.5**

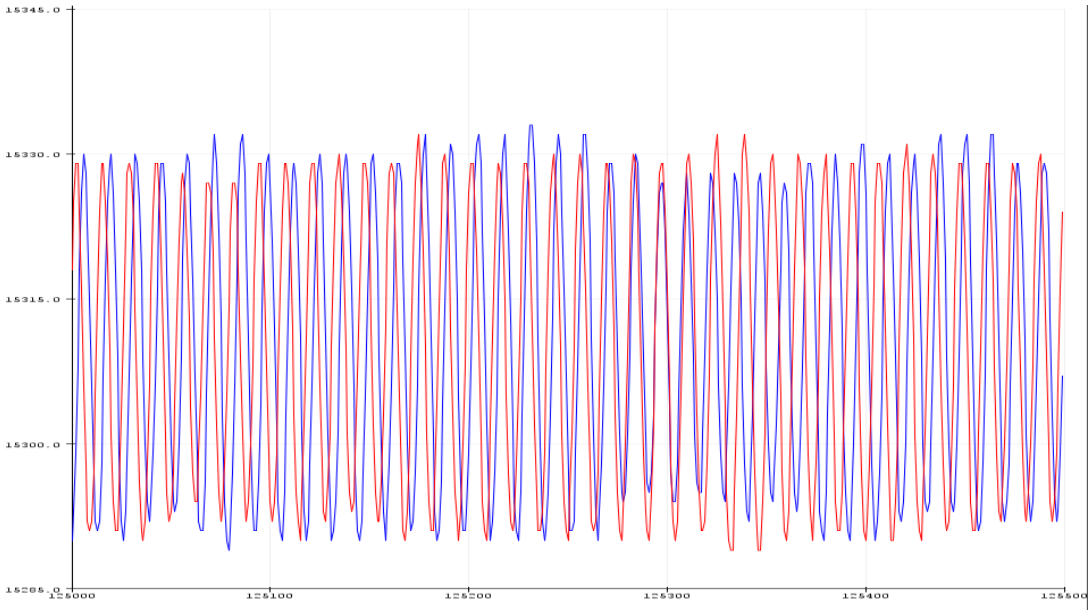
The results in Table 2 are very consistent when compared to the results in Table 1. The average encoder counts for each wheel are now very similar. This proves that changing the coupler design stopped it from slipping on the motor output shaft. The slight difference in the left and right wheel encoder counts is most likely due to the motors not being mounted exactly square to the chassis. The new encoder count averages can now be used to tune the PID controller. The motors need to spin at 15350 and 15207 encoder counts per second for the robot to drive at 1 ms^{-1}. When the max PWM signal of 255 was sent to the motors, the max speed they can rotate is 52 encoder counts per 10 ms loop. Therefore, the max forward speed of the robot is 0.34 ms^{-1}.

3.5.2 Position tuning

Originally it was thought that the PID controller needed to be tuned to make the robot drive to an exact position. This was because the robot was going to be given coordinates. However, it was later found that differential drive uses wheel velocities and not position. Therefore, tuning the PID controller for velocity was attempted, but as described in section 3.5.3 this was found to be very difficult. A workaround to the velocity tuning was to tune for position, and then if this position was constantly moved, it was hypothesised that it would cause the robot to drive at a constant velocity. It was also known that position tuning would be required to control the position of the Geiger counter arm. Therefore, tuning for position would not have been a complete waste of time if the tuning for velocity was eventually figured out.

When tuning for position, the Ziegler Nichols method gave a good starting point for the parameter values. The Ziegler Nichols method is the most popular tuning method. As it is taught in the ELEC2540 control system module, it is also very familiar. A guide by Giorgos Lazaridis [31] was followed during the tuning process. Even though the encoder counts in one meter varied slightly between the two wheels, using different setpoint values and tuning each PID controller separately would take too long. Therefore, the setpoint of the PID controllers used during tuning was the average of the two average wheels encoder values from Table 2. The robot was roughly driven forward one meter each time by having the setpoint equal the encoder values. The robot has to go from stopped then to its maximum output and then back to stopped so the Ziegler Nichols can be method is used.

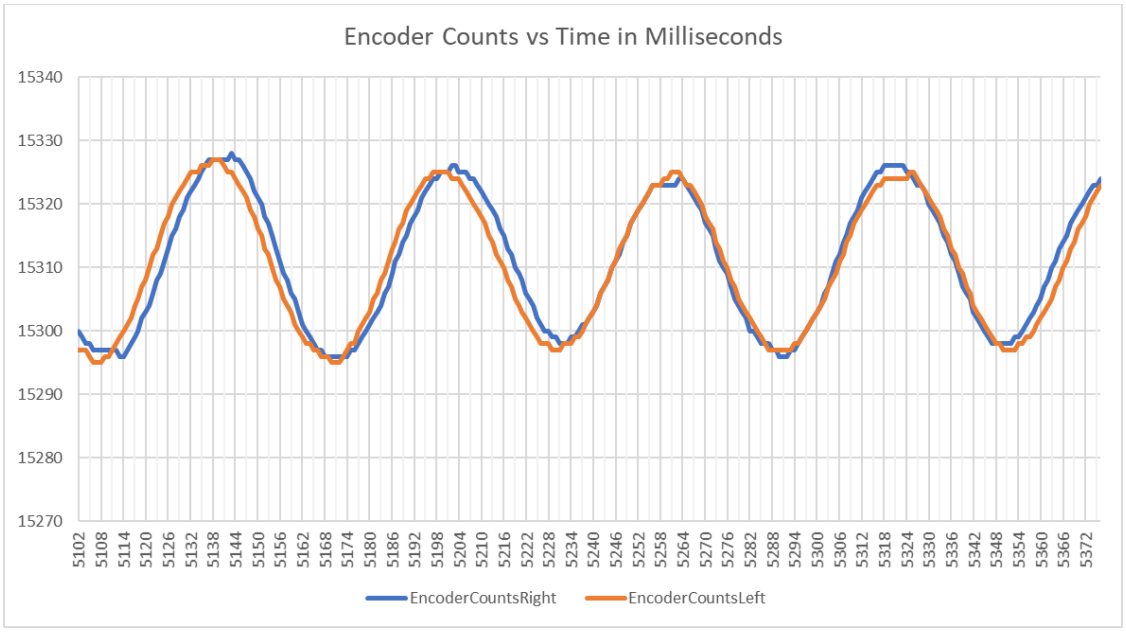
Figure 7: Arduino Serial Plotter output, Encoder Counts vs Sample Number



To start with, the Ki and Kd parameters are set to zero. Then the Kp value was slowly increased until the output oscillates around the setpoint at a stable rate. The serial plotter of the Arduino IDE was used to monitor when the output started oscillating, shown in Figure 7. The value of kp was nine when these oscillations occurred. This value is known as kc (critical p gain). The period of the oscillations was then measured. This value is Pc. However, it was challenging to read the oscillation period of the signal using the Arduino serial plotter. This is because the axes are not adjustable, and the x-axis is the number of samples and not the time.

To measure the oscillation accurately, the Arduino data had to be exported to Excel. This was done by following a guide from CrtSuznik [32]. The PLX-DAQ plug-in for Excel was used to export the output data, and the actual time is from the Arduino's inbuilt clock. Then a graph of the encoder counts vs time was plotted, shown in Figure 8. The oscillation period Pc was measured to be 0.0598 seconds.

Figure 8: Left and Right Wheel Encoder Counts vs Time



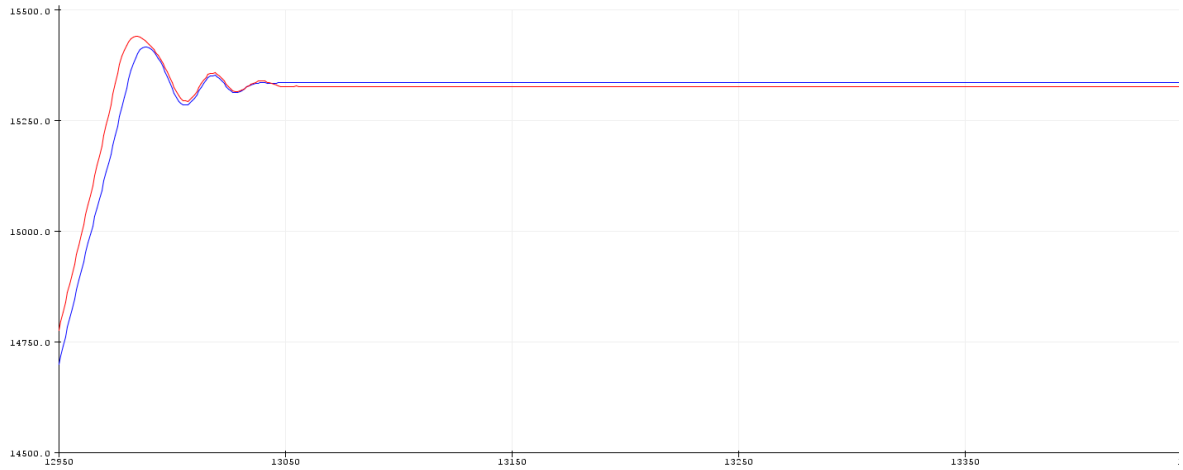
The starting values of the parameters for the PID controller are calculated using Table 3. These were Kp = 5.4, Ki = 0.0299 and Kd = 0.007475.

Table 3: Ziegler Nichols Method Parameter Starting Values Calculations

Type of Controller	Kp	Ki	Kd
P	0.5Kc	Inf	0
PI	0.45Kc	Pc/1.2	0
PID	0.6Kc	0.5Pc	0.125Pc

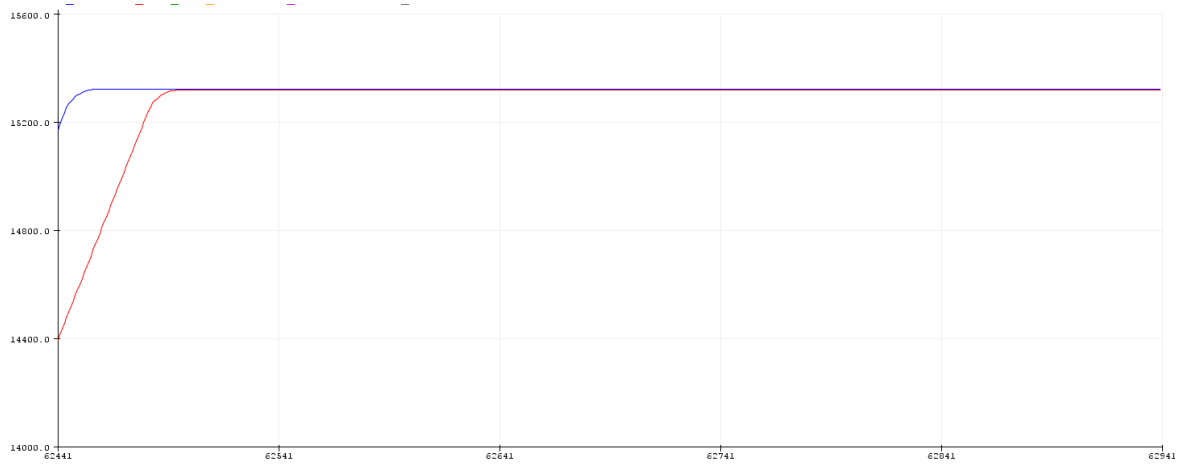
The Ziegler Nichols values were input into the PID control system, and the robot was driven to the setpoint. The output results with the Ziegler Nichols values are shown in Figure 9. The output has a significant overshoot and long settling time shown by the oscillations. A small, steady state error was shown by the difference in the settled values, which were then compared to the setpoint. The output for the left and right wheels are not the same as both the wheels use the same setpoints. In practice, the setpoints used should be the different values from Table 2.

Figure 9: Arduino Serial Plotter output, Encoder Counts vs Sample Number, Ziegler Nichols Values



The Ziegler Nichols method values are just meant to give a starting point, so the system response is not perfect when using those values. Therefore, the PID parameters were changed using the general behaviours described at the bringing of section 3.5**.** There was a small steady-state error, so Ki was not changed that much. The rise time was very fast, so Kp did need to be changed that much either. There was a large overshoot and long settling time, so Kd was increased to 0.3 to reduce this. These changes produced a much better system response with almost no overshoot and a short settling time, showing in Figure 10.

Figure 10: Arduino Serial Plotter output, Encoder Counts vs Sample Number, Further Tuned Values



The final parameters from tuning the PID control system for position where: Kp = 5.3, Ki = 0.03 and Kd = 0.3. The robot was then programmed to drive to the average encoder count from Table 2, 15311 counts. The robot travelled distance was measured to be 0.92 m. Therefore, a new encoder count setpoint was calculated by 15311/0.92 = 16642 counts. The robot now drove exactly one meter forward with the new setpoint but veered slightly to the left. This is due to the average setpoint value being used when the actual encoder counts per meter are different for each wheel. The left wheel setpoint was increased, and the right wheel setpoint was decreased until the robot drove one meter in a straight line. As the control system's loop time is 10 ms, the number of encoder counts required per loop to get the robot drive one ms^{-1} was calculated. This was 166 counts for the left wheel and 165 for the right wheel. Therefore, the measured number of encoder counts per loop can be used to calculate each wheel's velocity. As shown in Figure 5, these velocities are then published via a vector topic to the Pi, which uses them to calculate the wheel odometry.

3.5.3 Velocity tuning

As described in part 3.6, the Arduino subscribes to the cmd_vel topic, which includes angular and linear velocities. Differential drive works by using these two velocities to calculate the individual left and right wheel velocities. Therefore, the PID controller must be tuned to rotate the wheels at a setpoint velocity and not to a setpoint position. However, tuning the PID controller parameters for velocity was found to be very difficult. As the Ziegler Nichols method used when tuning for position did not work when tuning for velocity.

The Ziegler Nichols method usually works well for systems with a long time constant compared to their dead time [33]. This is why it worked for position tuning, as the time for the system output to reach 63% of the setpoint (time constant) was much longer than the time for the wheels to initially respond to change in the setpoint (deadtime). There was always going to be a dead time of a few milliseconds due to the 10 ms delay time in the main loop. When tuning for velocity, the setpoint is much smaller, around 16 counts per 10ms loop to drive at 0.1 ms^{1}. Therefore, the time constant is not much larger than the deadtime.

When the Ziegler Nichols method was tried, the system would not reach constant oscillation until really high pk values. Then when it did reach oscillation, the output for the low setpoint value was very noisy, meaning the oscillation period Pc could not be measured.

The noisy signal is due to the limited speed that the Arduino can read the encoder data because of the max sample time of the PID library. Also, the hall effect encoders used are not as precise and accurate as optical encoders [34]. Even if optical encoders were bought, the Arduino could only read a limited number of encoder counts per second. Therefore, at the low speeds that the robot is moving, the setpoint is always very small, making it challenging to measure oscillations.

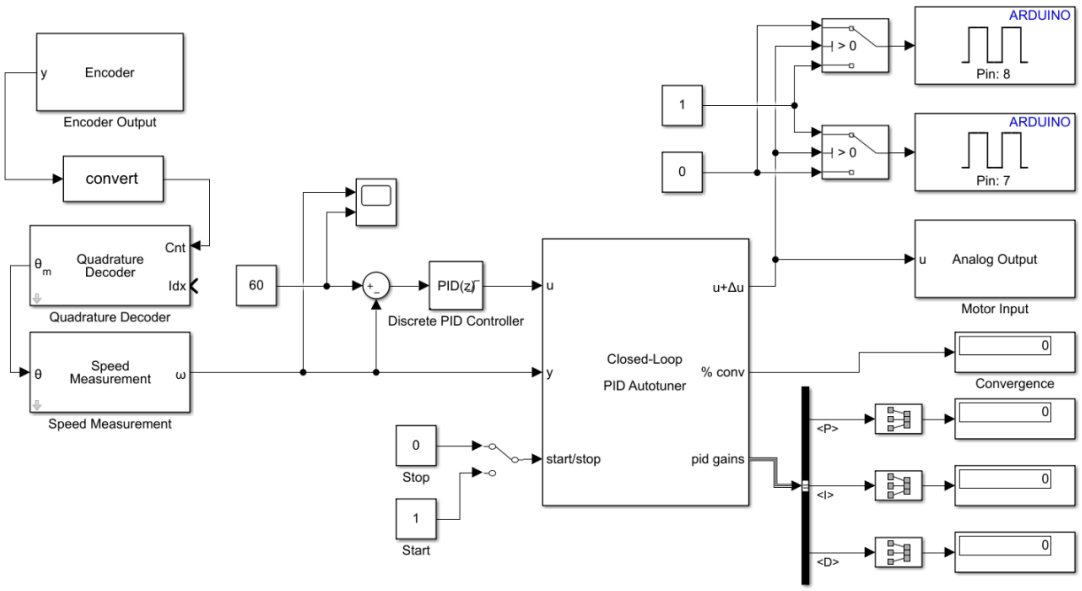
The other Ziegler Nichols method of tuning, which uses the stepped response of the plant, was also attempted. This method calculates the PID control system parameters using the deadtime and the delay time of the stepped response. However, as the encoder data is not precise or accurate, the exact deadtime could not be measured. Therefore, the starting point values obtained from the deadtime were not helpful. This also meant that other methods like Cohen-Coon tuning were not successful when they were tried. Manual PID tuning was attempted, but very high gains were required to get a decent response, so the motor's output was very choppy.

As the Ziegler Nichols method did not work and trying other different tuning methods became very time-consuming, efforts were focused on model-based PID auto-tuning [35]. An Arduino PID Autotune library was created by Brett Beauregard, who also made the used PID controller library [36]. However, this library was in early release and had really poor documentation, so it did not work when implemented into the code.

It was then found that model-based PID auto-tuning could be done using Simulink [37]. This method uses MATLABs control system toolbox and Simulink's control design add-on. MathWorks also has many helpful guides for PID control and excellent documentation. First, the system's plant was attempted to be simulated in Simulink. Then using the graphical interface, it was going to be tuned with response sliders [38]. However, as the plant has a complex behavior, it would have been challenging to simulate accurately as it could be linearised.

The Arduino hardware support package was then found, which meant a Simulink block diagram could be compiled and ran on the Arduino. Output data from the Arduino could be monitored live, and the PID parameters could be tuned in real-time. A Simulink block diagram shown in Figure 11 was built to run on the Arduino and connected motor hardware. It was then used to monitor and tune the PID controller parameters.

Figure 11: Simulink Block Diagram

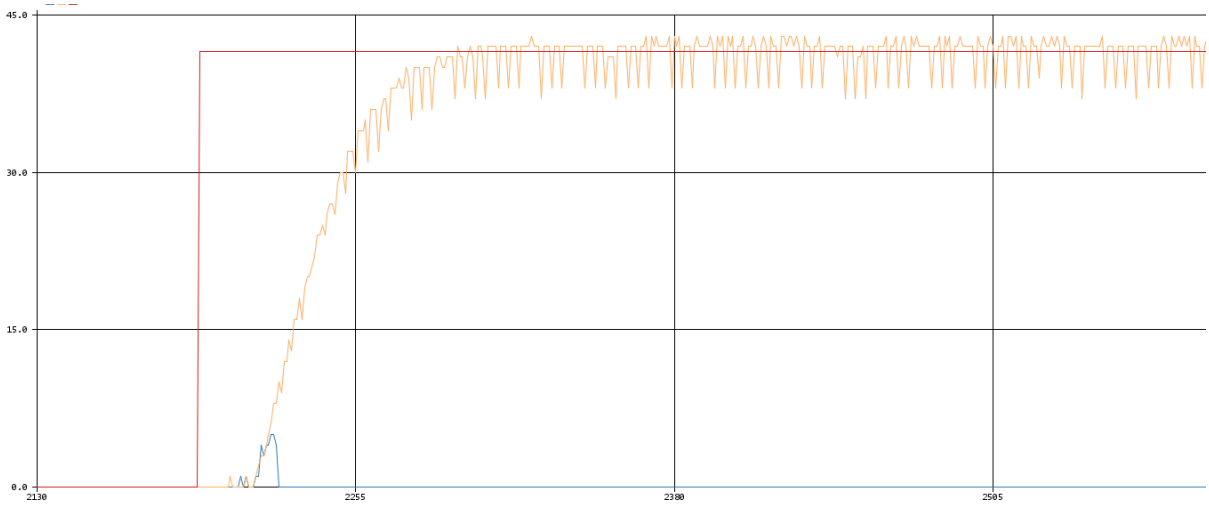


The Simulink block diagram was built around the Closed-loop PID Autotune block. This enables the real-time PID tuning of a physical plant rather than using a simulated plant. Which produces more accurate PID parameter values as it accounts for real-world factors like wheel slip that affect the system.

Setting up the closed-loop PID Autotuner block was straightforward due to the valuable guides on MathWorks. However, reading the encoder data from the Arduino and sending the PWM signal to the motors was much more difficult. There were no Simulink examples on how to this with the specific type of encoders and H-bridge used on the robot. It was possible to read the read the encoder data and the control the motors in MATLAB. Therefore, it was thought that custom Simulink blocks would have to programmed using MATLABs Function blocks. Fortunately, Giampiero Campa [39] had created a device driver add-on for the Arduino, which included Simulink blocks for reading the encoders and sending the analogue PWM output to the H-bridge. The encoder data had to go through the Converter block in Figure 11. This changed the variable type of the data so that the Quadrature Decoder block could be used. The Quadrature decoder block converts the encoder counts to the angular position of the motor. Then the Speed Measurement block was to work out the angular speed of the wheels.

Before tuning the PID values, the Closed-loop PID Autotune block parameters were set up. They were changed to aim for a fast response time. The memory and task overrun of the block had to be reduced. Otherwise, it would not compile onto the Arduino. Then the auto tuner was run until the convergence value equalled approximately 100%. The auto-tuned values obtained were Kp = 1.122, Ki = 20.03 and Kd = 0. This model uses discrete-time. This achieved much better results as the main loop in the Arduino is not continuous. The Simulink tuned PID values were then updated onto the PID controller in the Arduino code, which produced the output in Figure 12 when the robot was asked at drive 0.25 ms^{-1}.

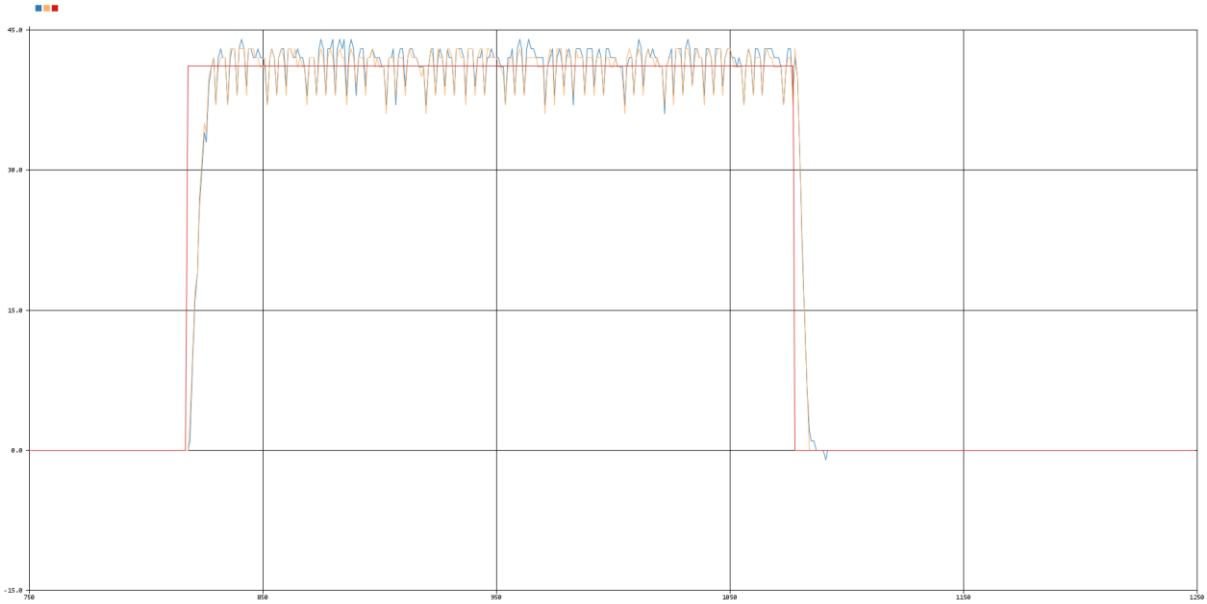
Figure 12: Arduino Serial Plotter output, Encoder Counts vs Sample Number, Simulink Tuned Values



The output shown in Figure 12 has a fast-settling time, low steady-state error, and no overshoot. However, the rise time and deadtime of output are very long, which would not lead to a fast system response. This was most likely because there was a limit on how quickly the response time could be set in Closed-loop PID Autotune block parameters.

The Simulink auto-tuned values set Kd to zero. It was interesting to see that a better system response could be achieved using a PI controller instead of a PID controller. This was crucial information to learn as any further manual tuning should not involve adjusting the Kd value to achieve a better system response. Also, through manual tuning methods it would have never been guessed that the Ki value should be much larger than the Kp value. Usually, Kp is much larger than Ki as having a larger Ki increases the settling time but this shows that every plant is different and manual tuning methods do not always work. PI controllers, which have Kd set to zero, are much better than PID controllers at avoiding noise in a system [40]. This system does have a noisy output, which explains why Kd is set to zero in the parameter values found from Simulink auto-tuning. It is known that when Kp is increased, the rise time decrease. Therefore, this value was then adjusted until the faster response in Figure 13 was obtained.

Figure 13: Arduino Serial Plotter output, Encoder Counts vs Sample Number, Adjusted Simulink Tuned Values



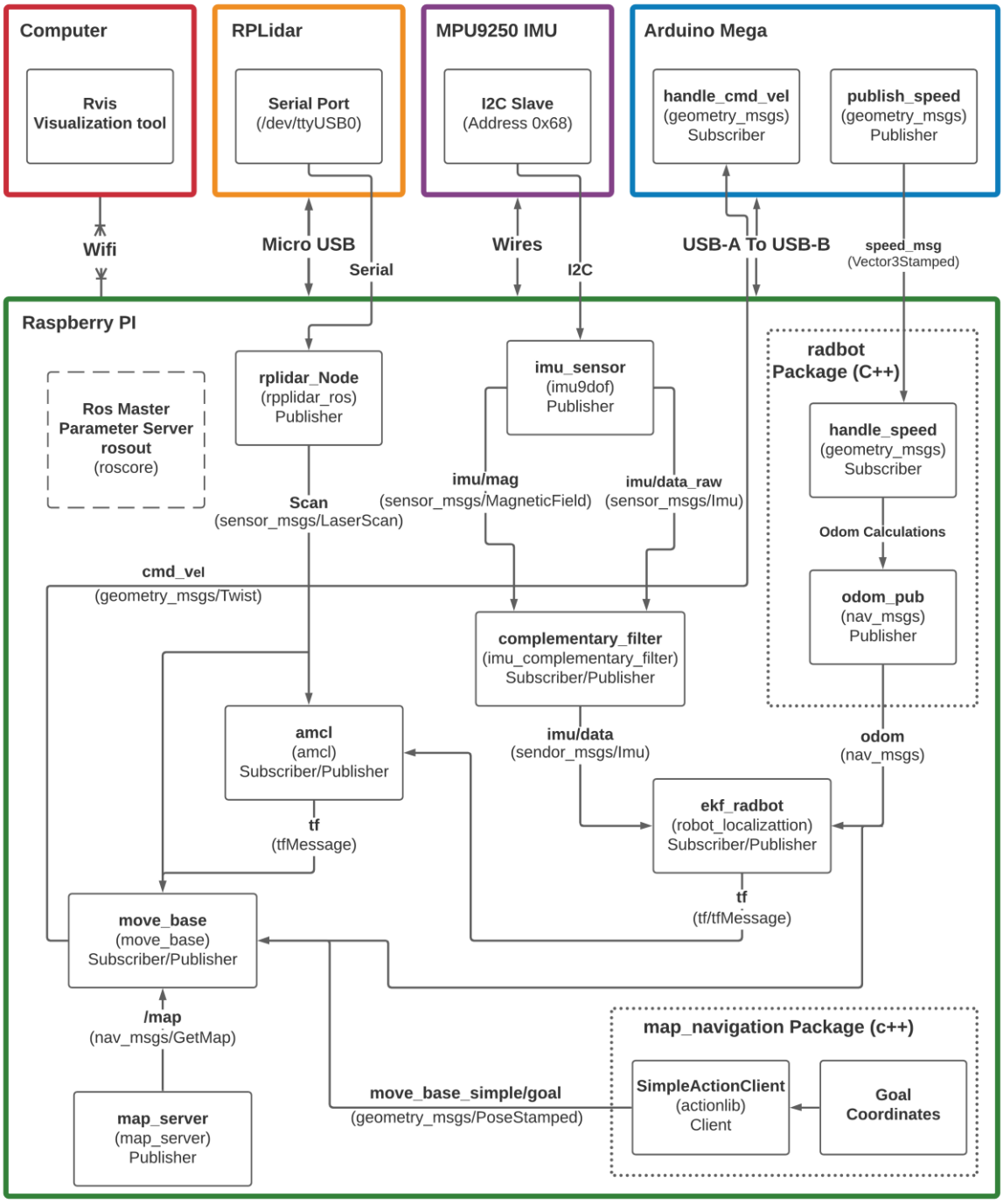
The final parameters from further tuning the PID control system for velocity where: Kp = 5.1, Ki = 20.03 and Kd = 0. The system now has a fast response, no overshoot, and small steady-state error caused by the encoder data noise. To test if the robot now had well-tuned encoders, it was driven at 0.25 ms^{-1} for four seconds. It dove approximately 1m and did not drift to left or right.

It did take a long time to tune the PID controller so the system would have a good response. To save time, it is possible to purchase motor controller boards like an Odrive. This would mean tuning is not be required, and higher resolution optical encoders could be used. However, motor controllers are very expensive, and IMU data can be used if the encoder data is not accurate enough. A lot was learned from the tuning process, so a greater understanding of the robot's control system works was developed.

3.6 ROS

As detailed on page 41, the hardware runs ROS packages to enable high-level control of the robot. Most of the work involves installing the packages and dependencies to use ROS, then programming launch files to start nodes within those packages. Make files have to be programmed to compile the ROS packages and any C++ code. The "Roslaunch" command can then be called via SSH to run the launch files. Figure 14 details how the different ROS nodes were set up to communicate with each other using topics and enable the SLAM of the robot. It also shows which nodes are included in each package and which piece of hardware it is running on.

Figure 14: ROS Node and Packages Connection Flow Diagram



The arrows in Figure 14 represent physical and ROS communication buses. The physical buses detail how the hardware is connected and the communication protocols used. The ROS buses are called topics. The topics are named in bold, and the type of message that the topic is in brackets. The topic is used to communicate between ROS nodes. The ROS nodes are the rectangles, their names are in bold, and the packages that the nodes are launched from is in brackets. Each node has details of whether it subscribes to topics, publishes topics or both. Each launched node has different parameters that are set in the launch files. Most of the nodes include parameters to change the type of topics that are published and the physical parameters of the robot.

The dotted rectangles detail custom c++ packages that have been programmed using c++. One of these packages is the radbot package. This is a package created by Baran [5]. It is responsible for calculating the wheel odometry from the speed_msg topic published from the Arduino. Usually, the recommended ROS method uses the microcontroller to conduct this calculation and have it published the wheel odom topic. This is because all the Pi's processor resources should be reserved for SLAM calculations. However, the Arduino does not have enough RAM to publish the Odom topic, so nodes are required on the Pi to do this instead. It can be argued that a Teensy microcontroller would have been a better choice for this project as it has enough RAM to publish the odometry topic. As the Arduino was already available for this project and with the solution from Baran [5], this justified not buying a Teensy. The other custom package publishes the move_base_simple/goal topic, and this is details on page 51.

As detailed in section 3.1.5, the IMU uses a ROS driver, which runs a node that publishes the imu/data_raw and imu/mag topics. These topics are combined together using the complementary filter [25], which then publishes the imu/data topic, including Z-axis orientation. Orientation can be calculated manually, but using this filter helps to stop z-axis drift, and it reduces inaccuracies in the magnetometer data caused by the motor magnetic fields.

Page 43 explains why IMU data is fused with the odometry data using the robot localization package [14]. The odometry and IMU inputs have to be appropriately configured to work the localization package. This is done by adjusting the node parameters to specify which inputs supply what type of data. Sensor fusion was implemented using a covariance matrix to configure the Kalman filter. It was found that the default values for the matrix produced accurate sensor fusion data for carpet flooring. Therefore these values were not adjusted that much.

The move_base node, detailed on page 51, control the movement of the robot. This node is responsible for publishing the linear and angular velocities which the Arduino subscribes. Section 3.4.1 details how the Arduino calculates the wheel velocities from the published cmd_vel topic. A URDF file used by the move_base node has been programmed. This describes the move_base node of the robot physical parameters. The physical parameters included are where LIDAR and wheel are attached to the chassis regarding the robot's centre point. Move base uses the sensor data from the LIDAR, which is configured in the AMCL node using the LIDARs specifications. Finally, cost map parameter files have been configured for the move base node. These include the robots footprint and collision avoidance parameters which have been tuned to make sure the robot does not collide with obstacles.

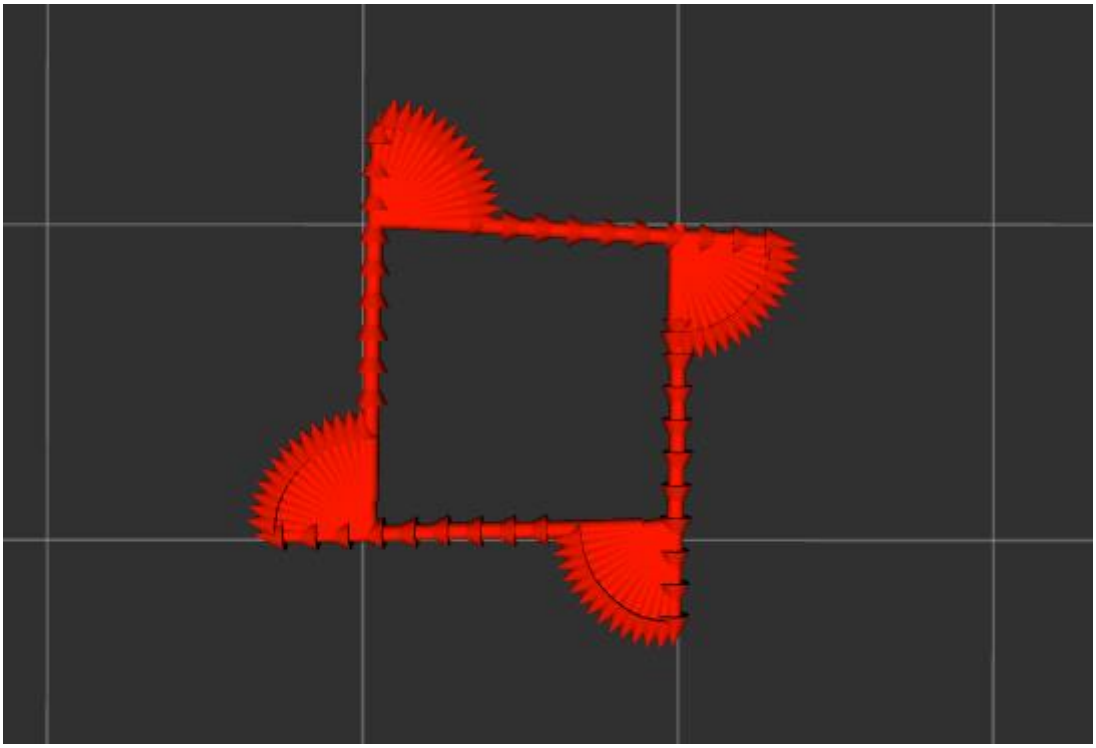
During testing and to create maps, Rviz is used to visualise any published topics on the ROS network. Rviz displays all the sensor data, so it was possible to confirm what information the robot is seeing and to check its accuracy.

3.6.1 ROS Scaler Tuning

Even with well-tuned encoders, the robot may not always move at the exact set speed as there are many factors like floor type which affect the wheel speed. When changing floor types, the number of encoder counts per 10ms loop would have to be changed so accurate velocity data could be published from the Arduino node. Doing this within the Arduino code is very time-consuming as the experiment in section 3.5.1 would have to be repeated. Therefore, the CPP code on the Pi that calculates the wheel odometry has linear and angular velocity scaler parameters that can be changed in the launch file. This means the velocity values can be tuned more quickly over SSH, and the Arduino does not need to be plugged into a computer each time.

To tune the velocities scaler values for different floor types, the robot localizes itself in Rviz. Then robot was driven using a computer keyboard at the speed that the radiation scan is conducted. It was driven until it reached one meter in Rviz. The Rviz distance is how far the robot thinks it has travelled. Then how far the robot physically moved was measured. The linear velocity scaler value was tuned until the robot would physically drive one meter. This process was repeated for the angular velocity scaler value, except that the robot was rotated 360 degrees instead of driving one meter forward. Then a box test was conducted to verify the wheel odometry was accurate enough. This evolved, driving the robot in square shape and checking it ended up back at the same point. However, on carpet, the robot was not very accurate in the box test when using the wheel odometry data. This is most likely because the casters unpredictably catch the carpet. When adding the IMU data, the robot could pass the box test on carpet, as shown in Figure 15. This highlights the importance of having an IMU on the robot.

Figure 15: Odometry Data from Box Test Visualised in Rviz

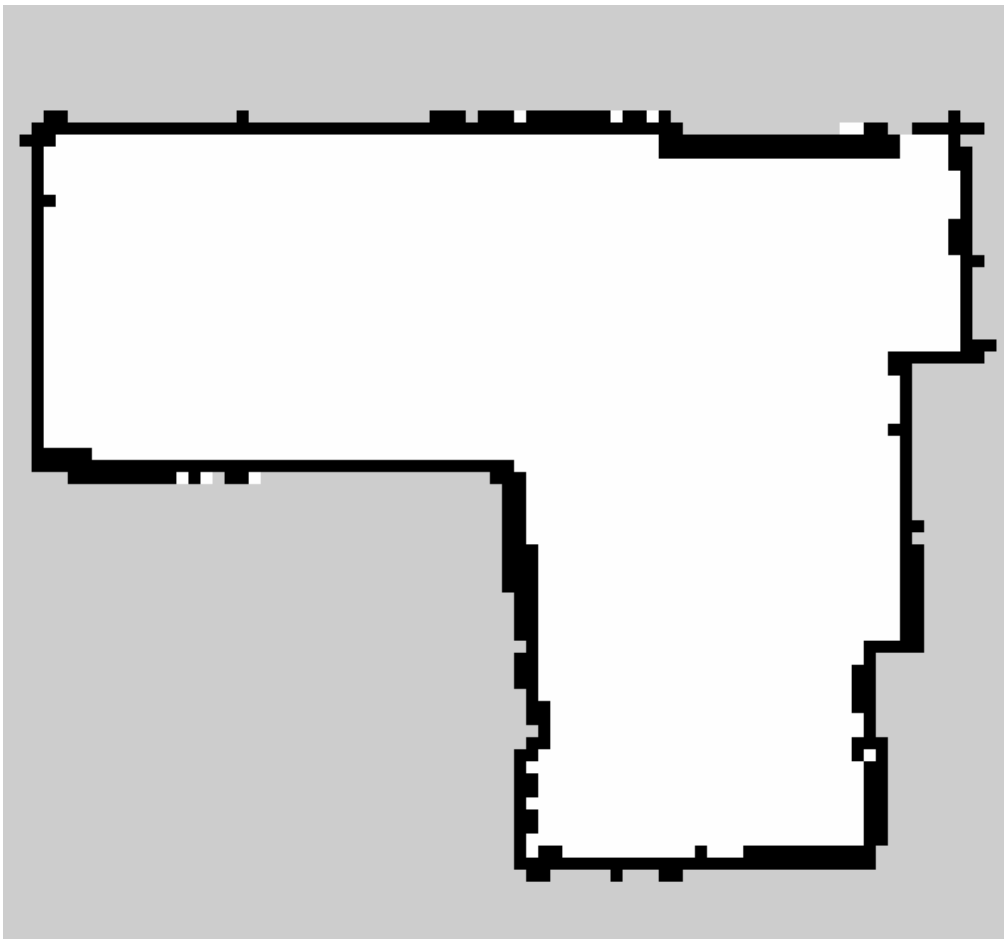


3.6.2 Mapping and Navigation

Due to limited access to the university radiation lab, a bedroom has been used to test the localisation and navigation of the robot. Before a path around the room can be planned, a room map must first be created. To create the map, the robot uses the ROS mapping node detailed on page 43.

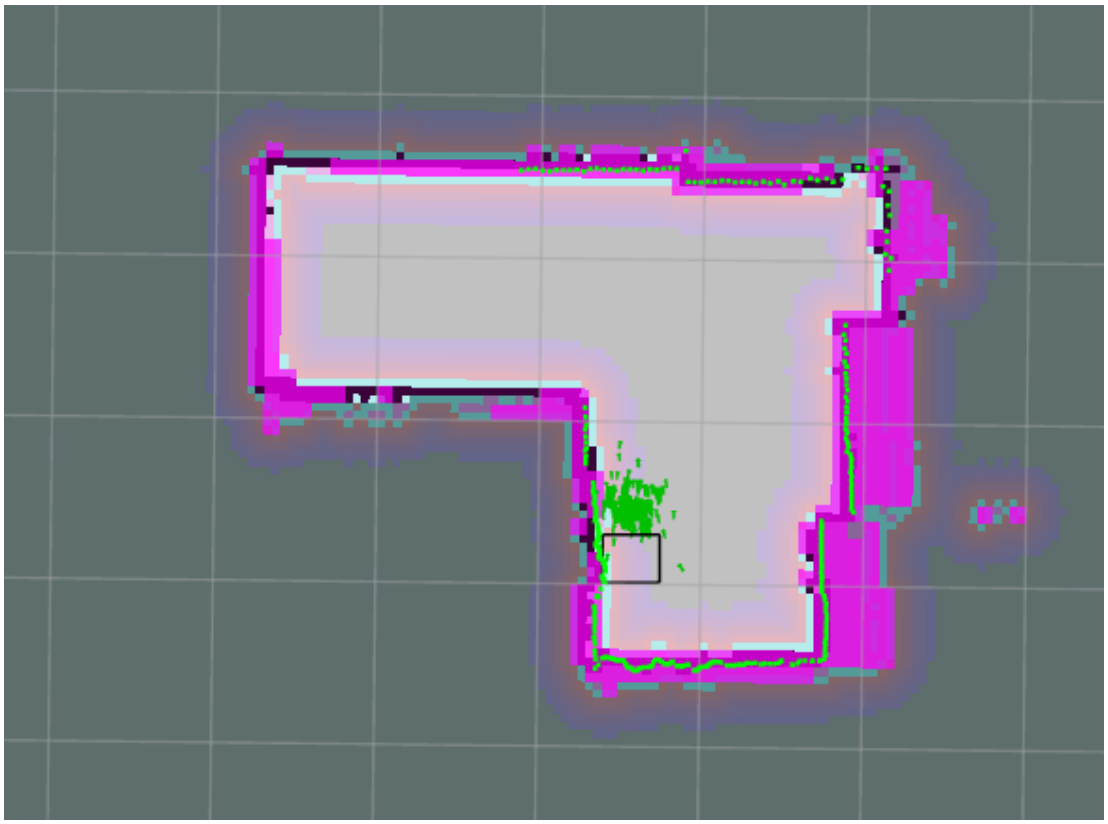
The robot has been slowly driven around the room for mapping to create the map. This is done by running the radbot c++ package and the teleop_twitst_keyboard package, which enable control of the robot via a computer keyboard. Rviz is used to visuals the map as it is created and to confirm it is correct. The map is then saved as a pgm and YAML file using the map server package. The PGM file is a visual representation of the map, and the YAML file contains the starting origin of the robot in the room. The visual representation of the map created is shown in Figure 16. A rosbag needs to be recorded. This was simultaneously the map recording, and the LIDAR and Odom data were included.

Figure 16: PGM of Map Created in Bedroom using gmapping



The map_navigation c++ package was then launched, which loads the saved map using the map_server node: Amcl and move_base nodes also launched, which localised the robot in the room after driving it around. Rviz was used to check if the robot had correctly localised. This is shown in Figure 17. The green dots are amcl particles that converge into the robot footprint as it moved around and localises itself more accurately. The blue highlighted areas are the inflation radius, which is the distance that the robot's footprint will not go into. This was tuned in the costmap setting and allows some clearance so the robot does not collide with the walls. The pink highlighted area is the marking and clearing area, which is the space that the robot has scanned again for other obstacles which it would try to avoid.

Figure 17: Robot amcl Localisation in Rviz



The robot was then given a goal to travel to. The action lab client is used to give goal coordinates to the move base node—the goal coordinates were set to drive the robot one meter forward. The move_base node then uses the ROS TrajectoryPlanner to plan a path to the goal coordinates, shown in Figure 18. The TrajectoryPlanner file can be configured to control how the robot plans routes. It has been set up to ensure the robot only drives forwards to avoid contamination as the Geiger counter is constantly checking for radiation in that movement direction.

Figure 18: Move_base Planed Path in Rviz

3.6.3 Final Testing

To begin with, the robot was driven via keyboard to check if it stopped when radiation was detected. This verified that the Arduino Geiger counter code works alongside the motor control system whilst the robot is moving. The robot was then given a path around the room to sweep, and the radioactive source was placed in a random position. For this room, the robot successfully made it around the whole room and stopped when it scanned the radiative source. The robot's position was tracked in Rvis throughout the scan.

Final testing still needs to be conducted in the university lab. This will require some retuning of the ros scaler values as the liberty floor is a different material to the floor in the previous test. The same tuning procedure conducted in 3.6.1 will be repeated to do this. From analysing the previous testing, the outcome from a laboratory test is very promising.

3.7 Future Developments

The future development for this project mainly involves producing a user-friendly robot. This would require more testing of the robot in the lab environment to make sure it works for multiple edge cases. The virtual user interface needs to be installed and tested on the Pi. Finally, the server version of Ubuntu 18.04 needs to be installed on the Pi and the launch files have to be configured to run on start-up.

The robot currently uses hall effect sensors which are relative quadrature encoders. Currently, the arm position cannot be determined as there the motor used does not have an attached encoder. To determine the arm position, an absolute quadrature encoder could be used. This type of encoder knows the position of the arm on start-up without needing a zeroing switch. Knowing the arm position enables it to add to the URDF file of the robot, so the object avoidance system knows where the arm is. Attempts could also be made to try and get the robot the carry on the lab scan after finding radiation by using the method proposed by the CARMA platform team at the University of Manchester. This would be made possible by knowing the position as more complex turning manoeuvres could be programmed onto the robot.

4 Conclusion

A custom ROS enabled robot platform to which the Geiger counter attaches has been successfully built. The Geiger counter has been set up and tested to work within the ROS system. All the selected ROS specific hardware has allowed for full implementation of SLAM. A PID controller has to be tuned to increase the movement accuracy of the robot. Sensor fusion of the wheel encoders of an IMU has allowed the robot to operate on multiple floor surfaces. The robot can be autonomously controlled by a path planning algorithm, enabling complete coverage of a room floor. The robot meets the overall specification of sweeping a room for radiation and stopping if radiation is found. Additive manufacturing is solely responsible for the robot being assembled in time and enabling fast prototyping of design ideas.

An excellent understanding of how to use the ROS API has been gained. This has built a great foundation of knowledge to use on future robotics projects. Many methods of tuning PID control systems have been explored. Even though model-based auto-tuning using Simulink successfully produced a fast system response, it was found that using a motor controller would have been a more time effective approach. More testing is required in a lab environment to confirm the robot's functionality over a longer scanning time, which has a more significant chance of sensor dating becoming inaccurate.

The hardware selected for the robot was cost-effective, with the total coming to the third price of the prebuilt ROS robot. Low-cost hardware does come with excellent documentation and online for implantation with ROS, enabling quick implementation time. However, further testing is beginning to expose the low accuracy of sensor data from inexpensive hardware. If the project were going to be developed any further for even higher-level control, much of the hardware would need to be upgraded.

Designing the robot platform from the ground up enabled a greater understanding of how the system works and allowed cleaner implementation of the Geiger counter arm. Although throughout the project, it was found that this caused many problems due to inaccuracy of mechanical designs affecting ROS system behaviour. This supports that buying a prebuilt platform designed explicitly for ROS would have meant further project progress could have been made. Less time would need to be spent on the mechanical design, and more time could be dedicated to ROS programming.

5 References

[1] ROS Robot Programming 1st ed, ROBOTIS Co.,Ltd., Seoul, South Korea 2017, pp. 1-487

[2] A. G. Benjamin Bird, Horatio Martin, Eduardo Codres,, A. S. Jennifer Jones, Barry Lennox, Simon Watson,, and a. X. Poteau. (2019) A Robot to Monitor Nuclear Facilities: Using Autonomous Radiation-Monitoring Assistance to Reduce Risk and Cost. *IEEE Robotics & Automation Magazine (Volume: 26, Issue: 1, March 2019)*. 35-43.

[3] matop_m. (2019, Nov 26). Build Your Own Turtlebot Robot! [Online]. Available: https://www.instructables.com

[4] Y. Shaoul. (2020, Jan 23). A Full Autonomous Stack, a Tutorial | ROS + Raspberry Pi + Arduino + SLAM. [Online]. Available: https://yoraish.com

[5] R. Baran. (2020, June 5). Nox_robot. [Online]. Available: https://github.com

[6] Raspberry Pi 4 Computer Model B,1st ed, Raspberry Pi Trading Ltd, Cambridge, UK, 2020, pp. 3

[7] C. Pietschmann. (2019, Jun 24). Raspberry Pi 4 vs NVIDIA Jetson Nano Developer Kit. [Online]. Available: https://build5nines.com

[8] R. Backend. Using ROS on Raspberry Pi: Best Practices. [Online]. Available: https://roboticsbackend.com

[9] Arduino. ARDUINO MEGA 2560 REV3. [Online]. Available: https://www.arduino.cc

[10] L. Shanghai Slamtec.Co., "RPLIDAR A1," A1M8_v2.2 datasheet, Mar. 2013 [Revised Feb. 2019].

[11] J. Palacin, D. Martinez, E. Rubies, and E. Clotet, "Mobile Robot Self-Localization with 2D Push-Broom LIDAR in a 2D Map," *Sensors (Basel)*, vol. 20, no. 9, Apr 28 2020, doi: 10.3390/s20092500.

[12] Y. Chen *et al.*, "The Accuracy Comparison of Three Simultaneous Localization and Mapping (SLAM)-Based Indoor Mapping Technologies," *Sensors (Basel)*, vol. 18, no. 10, Sep 25 2018, doi: 10.3390/s18103228.

[13] R. Üveges, F. Ďurovský, and V. Fedák, "Skid Steering of Robotic Vehicle for Autonomous Applications," in *2019 International Conference on Electrical Drives & Power Electronics (EDPE)*, 24-26 Sept. 2019 2019, pp. 335-341, doi: 10.1109/EDPE.2019.8883907.

[14] T. Moore. robot_localization [Online]. Available: http://docs.ros.org

[15] Z. Wang, Y. Chen, Y. Mei, K. Yang, and B. Cai, "IMU-Assisted 2D SLAM Method for Low-Texture and Dynamic Environments," *Applied Sciences*, vol. 8, no. 12, 2018, doi: 10.3390/app8122534.

[16] V. Mazzari. (2020, Feb 10). IMU and robotics: All you need to know. [Online]. Available: https://blog.generationrobots.com

[17] T. B. Schön, J. D. Hol, and M. Kok, "Using Inertial Sensors for Position and Orientation Estimation," *Foundations and Trends® in Signal Processing*, vol. 11, no. 1-2, 2017, doi: 10.1561/2000000094.

[18] InvenSense, "MPU-9250 Product Specification Revision 1.1," PS-MPU-9250A-01 datasheet, Dec. 2013 [Revised Jun. 2016].

[19] D. Aoyagi and S. Choi, "Angular Position Estimation of an Inverted Pendulum Using Low-Cost IMUs," *American Journal of Sensor Technology*, vol. 5, no. 1-6, 2018, doi: 10.12691/ajst-5-1-1.

[20] Felfil. (2021, Jan 18). PETG FILAMENT FOR 3D PRINTING: LEARNING ABOUT PLASTIC MATERIALS. [Online]. Available: https://felfil.com

[21] S. Ba. (2016, Oct 08). Raspberry Pi 4, Pi 3, Pi 2 case with Vents, 40mm, or 30mm fan. [Online]. Available: https://www.thingiverse.com/thing:1815268

[22] 3DRay. (2016, Apr 01). Arduino Mega 2560 Case. [Online]. Available: https://www.youmagine.com

[23] HGROSS. (2019, Mar 02). RPLIDAR A1M8 CASE. [Online]. Available: https://cults3d.com

[24] B. Jamshedji. (2017, Mar 11). Controlling DC Motors with the L298N Dual H-Bridge and an Arduino. [Online]. Available: https://dronebotworkshop.com

[25] rishabhdevyadav. (2020, Mar 25). ROS_IMU_Filter. [Online]. Available: https://github.com
[26] s.a.alghamdi94. (2017, Dec 11). Using Raspberry Pi 4, With Ubuntu , Ros , Rplidar , Arduino. [Online]. Available: https://www.instructables.com
[27] J. Bruton. (2021, Mar 19). ReallyUsefulRobot. [Online]. Available: https://github.com
[28] T. Foote. Twist Message. [Online]. Available: http://docs.ros.org
[29] B. Beauregard. (2017, Jun 20). PIDLibrary. [Online]. Available: https://playground.arduino.cc
[30] H. Jenkins. TuningforPIDControllers. [Online]. Available: https://engineering.mercer.edu
[31] G. Lazaridis. (2011, Aug 13). PID Theory. [Online]. Available: http://www.pcbheaven.com
[32] CrtSuznik. (2015, Aug 20). Sending Data From Arduino to Excel (and Plotting It). [Online]. Available: https://www.instructables.com
[33] L. PAYNE. (2014, July 1). Tuning PID control loops for fast response. [Online]. Available: https://www.controleng.com
[34] enrikds. (2018, Mar 14). Magnetic Encoders VS Optical Encoders. [Online]. Available: https://enriquedelsol.com
[35] M. Huizer. (2020, July 14). When the Ziegler-Nichols PID tuning method doesn't do the trick. [Online]. Available: https://blog.incatools.com
[36] B. Beauregard. (2012, Oct 5). PIDAutotuneLibrary. [Online]. Available: https://playground.arduino.cc
[37] J. Avendano. How to Automatically Tune PID Controllers. [Online]. Available: https://www.mathworks.com
[38] A. Turevskiy. PID Controller Design in Simulink. [Online]. Available: https://www.mathworks.com
[39] G. Campa. (2016, Sep 01). Device Drivers. [Online]. Available: https://www.mathworks.com
[40] Smriti Rao and R. Mishra, "Comparative Study of P, PI and PID Controller for Speed Control of VSI-fed Induction Motor," *International Journal of Engineering Development and Research (IJEDR)*, vol. 2, no. 2, pp. 2740-2744, June 2014.