Development of a Ground Robot for Indoor SLAM Using Low-Cost LiDAR and Remote LabVIEW HMI

By

Jack Schubert

Under the supervision of

Dr David Parlevliet
PhD, BSc (Hons)

Submitted to the Murdoch University School of Engineering and Information Technology in partial fulfilment of the requirements for the degree of

Bachelor of Engineering (Hons)

in the discipline of

Electrical Power Engineering & Industrial Computer Systems Engineering

Murdoch University, Perth, Western Australia

Author's Declaration

I declare that this paper is my own work, has not been copied from another person's work and has not been previously submitted to Murdoch University or any other academic institution.

_____

Jack Schubert

_____

Word Count (body text)

Abstract

The simultaneous localization and mapping problem (SLAM) is crucial to autonomous navigation and robot mapping. The main purpose of this thesis is to develop a ground robot that implements SLAM to test the performance of the low-cost RPLiDAR A1M8 by DFRobot. The HectorSLAM package, available in ROS was used with a Raspberry Pi to implement SLAM and build maps. These maps are sent to a remote desktop via TCP/IP communication to be displayed on a LabVIEW HMI where the user can also control robot. The LabVIEW HMI and the project in its entirety is intended to be as easy to use as possible to the layman, with many processes being automated to make this possible.

The quality of the maps created by HectorSLAM and the RPLiDAR were evaluated both qualitatively and quanitatively to determine how useful the low-cost LiDAR can be for this application. It is hoped that the apparatus developed in this project will be used with drones in the future for 3D mapping.

Acknowledgments

First and foremost, to my supervisor, David Parlevliet. Your dedication to your work, your knowledge and your willingness to make time when you have none made it possible for me to achieve and learn what I did throughout this semester. Your support, curricular or otherwise has been greatly appreciated.

I would also like to thank all the staff in the Murdoch University School of Engineering and Information Technology, particularly Graeme Cole, Gareth Lee, Linh Vu and many others. I have learned things that I never imagined I would learn throughout this course and your dedication to your students and to improving the course curriculum and university facilities made my time at Murdoch University an enjoyable and exciting journey. You all helped me to find my passion and for this I will be forever grateful.

To my friends and family, and once again the staff at Murdoch University I would like to thank you all for your support and understanding through my ill health. Without your encouragement and assistance through this time I would not have been able to continue with my studies and finish my degree by the time I did.

Finally, I would like to thank Iafeta “Jeff” Laava. Whose fun, selfless nature and desire to help everyone around him made everyone’s time at Murdoch University a much smoother and more enjoyable ride. From fetching and fixing equipment for our projects, often in his own time, to making his expertise available whenever they were needed, to being a friend to all, no task was too difficult. May he rest in peace.

Table of Contents

Table of Figures

Abbreviations

HMI – Human machine interface

ROS – Robot operating system

USAR – Urban search and rescue

UGV – Unmanned ground vehicle

RGB-D - Red, green, blue & depth

TCP – Transmission control protocol

IP – Internet protocol

GPIO – General purpose input/output

Glossary

Bash script – A modern form of shell scripting that can be used to execute a sequence of commands upon its own execution.

BCM GPIO – Referencing Raspberry Pi GPIO pins using the Broadcom SOC channel option.

Board GPIO – Referencing Raspberry Pi GPIO pins using the board option.

Catkin workspace – A folder in which ROS packages are “built” and modified.

1 Introduction

It has long been hoped that robots would be the answer to many of the problems we, as humans face today. For many years robots have been used to carry out repetitive, mundane and laborious tasks because we don't want to, or of course because people don't want to pay us to. With robots originally being created for these kinds of tasks it would have been unfathomable to imagine what they would become for those around to witness their inception. Not only can robots carry out simple, repetitive tasks with speed and precision, today's robots can execute much more complex ones with such sophistication that the way they act and "think", to some, is comparable to that of a human.

Today the autonomous car industry is booming with experts betting on the idea that having robots making decisions for us on the road will make daily driving a safer and more efficient experience [1]. For a car to be able to navigate its environment effectively it must be able to interpret its surroundings. This usually involves building a map. It must also be able to determine its location in this environment. This is known as simultaneous localization and mapping or SLAM and is one of the most fundamental problems that must be solved in the realm of autonomous navigation [2]. Since the SLAM problem was solved (to a reasonable extent) decades ago its applications in other fields has also been realized.

Light detection and ranging (LiDAR) sensors are devices that emit pulses of laser light to measure distance. Data collected from these sensors can be used to build incredibly detailed 2D and 3D maps of the world around us. Maps generated with LiDAR have taken over from more traditional methods of mapping for use in surveying, agriculture, mining and almost any other application requiring detailed maps. With the help of complex algorithms, LiDAR sensors can be used to solve the SLAM problem for use in autonomous navigation in military applications and urban search and rescue missions.

Perhaps the most critical drawback when it comes to using LiDAR sensors for SLAM is their cost. This project will focus on utilizing a low-cost LiDAR and other low-cost components to develop a

teleoperated robot that implements SLAM. The aim is to use the Robot Operating System (ROS) on a Raspberry Pi to drive the robot and to integrate this platform with a LabVIEW human machine interface (HMI) on a remote desktop. The LabVIEW HMI and the project in its entirety is intended to be simple to operate for the layman and considerable efforts were made to automate any start-up and execution processes on the robot and on the remote PC.

This work is somewhat intended to lay the foundations for future projects aimed at developing autonomous aerial or ground vehicles for a variety of applications. There will be consideration taken to make the methods and outcomes of this work as transferrable to aerial vehicles as possible.

2 Background

2.1 The SLAM Problem

The SLAM problem describes a robot's ability to build a map of its environment and localize itself in space at the same time. It is not possible for a robot to do one of these without the other. Initially it was believed to be an unsolvable problem, but with significant research over last few decades, mostly due to its necessity in the autonomous car industry, there are now many algorithms that make SLAM possible.

Occupancy grid mapping is a mapping technique that represents the environment as a 2D array by occupying cells based on distance measurements from range finding devices [3]. Figure 1 is a low-resolution example of a 2D occupancy grid map. Higher resolution maps with many more, smaller cells can display extremely precise and detailed maps. This technique allows a robot to identify features in the environment that can then be used to determine its location by a process known as feature-based scan matching. Scan matching considers real-time laser scan data, transforms this data into a map and compares features on this map with features on the map learned so far to determine its current position relative to these features [4].

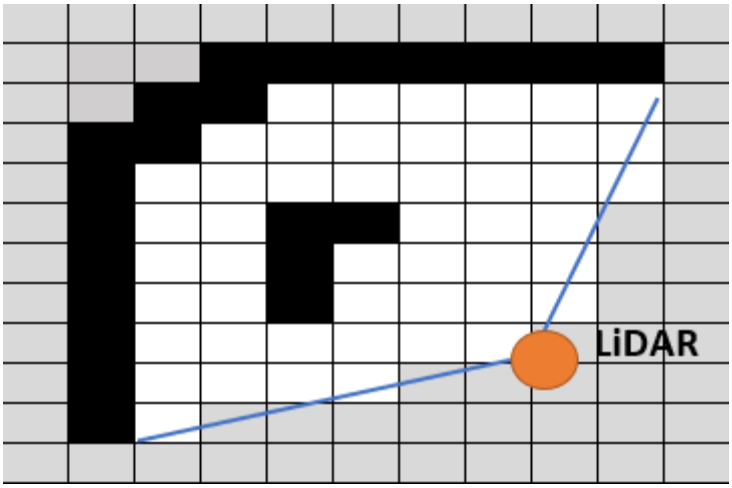


Figure 1: Example of a low-resolution occupancy grid map

Most SLAM methods determine a robot's position on a map by relying either solely on odometry data to measure how far and in what direction a robot has deviated from its initial position. Many algorithms also use a combination of odometry and scan matching to estimate the robots orientation (pose). The nature of physical measurement devices such as wheel encoders and laser scanners are such that they introduce uncertainty into the problem due to measurement errors. The uncertainty related to map accuracy increases exponentially over time due to these errors being present with each scan. Of all the working SLAM algorithms today, most rely heavily on probabilities to determine what is likely to be the most accurate map. Different algorithms use different methods of evaluating uncertainty and making adjustments to the maps accordingly, known as filtering [5].

Today, there are many existing algorithms that can achieve SLAM with the most widely adopted being the Extended Kalman Filter and fastSLAM [6].

2.2 Applications of LiDAR SLAM

2.2.1 Autonomous Navigation

Arguably one of the most exciting applications for LiDAR mapping is autonomous navigation [7]. As previously discussed, solving the SLAM problem is crucial for a car to be able to navigate its environment without human interaction. With majority of the autonomous car industry betting

primarily on this technology for mapping, LiDAR technology is forecast to become a US $5billion market in the next 5 years [8].

2.2.2 Urban Search and Rescue

The use of mobile robots with LiDAR to solve the SLAM problem is also of interest in the urban search and rescue (USAR) field [9]. In natural disasters or terrorist attacks people often become trapped and/or injured in collapsed buildings and other damaged structures. Often these areas are dangerous or physically impossible for search and rescue teams to enter and offer support. Robots were first used for this purpose after the World Trade Centre attacks [10].

Deploying autonomous robots in USAR scenarios can serve a variety of purposes. They can locate and offer support to trapped or injured people in the affected area. They allow rescue teams to allocate resources to efforts other than controlling a teleoperated vehicle, like those used after the September 11 attacks. Furthermore, creating maps of the area can give insight as to how the environment has changed due to the event and make rescue teams aware of potential hazards.

2.2.3 Surveying

LiDAR mapping also has a place in many different areas of surveying. Unmanned ground vehicles (UGVs) with an onboard LiDAR are now being used to create 3D maps of forests [11]. These maps can be used to determine the number and size of trees in a forest to gain insight into forest exploitation and the overall state of these forests. UGVs can also be used to map tunnels and sewer systems to check for damage or to determine how far they span underground to assist with future development and improve safety.

Abandoned mines that haven't been appropriately documented or mapped pose a serious threat to society. Some of the hazards associated with abandoned mines include a risk of unwitting passers by falling into open mine shafts, toxic or flammable gases, cave-ins, unstable structures and flooding [12]. In 2002 nine coal miners became trapped in the Quecreek Mine in Pennsylvania after breaching a wall between them and an old, flooded abandoned mine shaft. A special hearing held in October of 2002

found that the direct cause of the disaster was a lack of complete and detailed maps of the abandoned shaft [13]. There are an estimated 60,000 abandoned mines in Australia today, many with little or no data with regard to their state or layout beneath the Earth's surface [14]. Autonomous mapping robots could propose a solution to this problem by creating detailed maps of mine shafts and uploading them to a cloud database for the public and mining companies to access and view.

2.3 Rangefinders

Rangefinders are devices that measure the distance between itself and a target object remotely. Historically they have mostly been used for depth measurements in maritime applications and surveying. Today they are crucial for obstacle avoidance and SLAM in the autonomous car industry. Most distance sensors such as ultrasonic and LiDAR rely on time-of-flight measurements to determine distance. In this application the time-of-flight is the time taken for an emitted pulse of sound or light to reach the target and return to the sensor. This measurement, paired with the speed of sound or light in air can be used to determine the distance between the sensor and the target object.

2.3.1 Ultrasonic

Ultrasonic distance sensors are a high frequency form of sonar that rely on time-of-flight data sound waves to measure distance. Ultrasonic sensors are generally low-cost and are often used as a low-cost solution for obstacle detection, collision avoidance and less often mapping in autonomous navigation, particularly by hobbyists [15].

While sonar has been heavily researched and tested as a 2D map building solution there are some characteristics of sound waves make them less useful for map building. The speed of sound in air is relatively slow compared to light making for a lower sample rates of around 10-20Hz and consequently fewer measurements per second [16]. Another drawback of ultrasonic rangefinders is their poor range due to acoustic attenuation. Many low-cost ultrasonic sensors only measure up to 2 meters reliably. Larger distances introduce significant errors rendering the data useless for map building [17].

2.3.2 LiDAR

Light Detection and Ranging (LiDAR) works in the same way as ultrasonic rangefinders with laser light being used instead of sound waves. There are many applications for LiDAR rangefinders from making simple linear distance measurements in surveying to creating incredibly detailed maps of the sea floor in bathymetry [18]. BMW [19], Uber and Waymo [20] are all investing heavily in LiDAR technology for their autonomous car programs.

High sample rates and accurate readings are key to what makes LiDAR sensors so attractive for map building. Sensors that can make more measurements per 360^{o} rotation, naturally, produce higher resolution maps and detect more features in the surrounding environment. As a consequence of this they also require more computing power to process the data.

LiDAR sensors – particularly those designed for mapping in driverless cars, are notoriously expensive with a cost of around $75,000 USD per car [21]. There are, however, an increasing number of low-cost options that are either in development or already on the market. An example of such is the RPLiDAR range by DFRobot with their entry level 360^{o} LiDAR scanner, the A1M8, coming in at just $99 USD. The A1M8 boasts a 12-meter range and a sample rate of 8000 samples per second [22].

Most of the 360^{o} LiDAR sensors used today use a single sensor that rotates 360^{o} while taking readings. While effective, there is a growing concern about the lifespan of these devices due to moving parts. There is currently a focus on developing technology that uses a solid-state design to omit this problem [8]. There are also research efforts towards converting existing 2D LiDARs into 3D LiDARs in an effort to reduce the cost and increase accessibility to 3D mapping [23].

2.3.3 Camera & Microsoft Kinect

Certain cameras can also be used for distance measurement for 2D mapping and localization. The main benefit of using cameras for this purpose in place of LiDAR is their availability and low cost. Elon Musk of Tesla Inc. is adamant that LiDAR technology is not crucial to autonomy and insists that Tesla cars will be capable of full autonomy with only cameras, radar and ultrasonic sensors [24]. Another

benefit of using cameras for SLAM is their ability to identify features by colour. LiDAR sensors can only identify features based on their dimensions. The main drawback of using cameras for SLAM is their dependence on ambient light to function. In scenarios with a high variance in illumination such as going through tunnels or frequent weather changes cameras tend to make inaccurate readings [25].

Microsoft's Kinect is a colour and depth (RGB-D) camera developed for use with the Xbox 360 and Xbox One video game consoles. Interestingly, Microsoft released a software development kit to encourage enthusiasts and researchers to explore the technology. It can now be used for 3D mapping, localization and navigation in robotics [26]. The Kinect differs from other cameras in that it can measure depth in the images it captures. The depth component of the camera relies on a form of LiDAR known as flash LiDAR. In contrast to scanning LiDARs, these can measure the depth of an entire scene with a single light or laser pulse rather than a series of scans [27]. While Microsoft has ceased production of Kinect cameras they have a continued partnership with Intel to further develop their Realsense depth cameras and encourage development of the technology [28, 29].

2.3.4 Kinect and LiDAR integration (Sensor Fusion)

With different sensors having different advantages and disadvantages sensor fusion serves to integrate sensors in a way that exploits the positive characteristics of each sensor. One example of sensor fusion is using LiDAR with cameras to improve mapping and obstacle detection [30]. In general, LiDAR is much more effective than cameras for the localization aspect of SLAM and cameras are often more useful for obstacle detection and feature recognition.

Work presented at the 2017 Workshop on Positioning, Navigation and Communications conference (WPNC) in Germany found that using a 2D LiDAR for mapping was more effective than using a Kinect. The average deviation of the measured trajectory from the real trajectory when using a Microsoft Kinect was found to be 0.42m and only 0.11m when using a 2D LiDAR [31]. The Kinect's limited field of view was determined to be the main factor responsible for its poor performance for localization [32]. Maps based on LiDAR scans alone were found to be much more accurate than those based on

Kinect scans. However, when merging the LiDAR and Kinect maps it was found that the merged maps showed more features and obstacles and also reduced the extent of the skew evident in LiDAR-only maps [33]. These findings and many others support the notion that sensor fusion may be a more appropriate solution to the SLAM problem than single-sensor solutions. Figure 2 compares the performance of a LiDAR and the Microsoft Kinect in terms of the average deviation between the real trajectory and the measured trajectory.

SLAM method	Sensors	Average deviation, m	Maximal deviation, m
Hector SLAM	2D LiDAR	0.11	0.18
RTAB-Map	Kinect 2.0 depth sensor	0.42	0.67

Figure 2: Comparison of trajectory deviations between LiDAR-based SLAM and camera-based SLAM (data from[31])

2.3.5 GPS

One method of localization when solving the SLAM problem is using GPS. GPS is most useful in featureless environments where scan matching algorithms are less effective such as long roads or deserts. However, it does have its drawbacks. GPS technology, in general, is not accurate enough for autonomous navigation and sophisticated error evaluation techniques must be used to counteract these errors [34]. Another drawback of GPS is its availability, GPS-denied areas such as tunnels and areas with overhanging trees or canopies can also present situations where localization is not possible with GPS. While GPS does have significant drawbacks its fusion with other methods of localization such as inertial measurement units (IMUs) is of interest for larger urban areas [35, 36].

2.4 Robot Operating System (ROS)

Supported primarily on Ubuntu Linux, ROS is a middleware that provides a plethora of libraries, drivers and tools for robot development. ROS was developed with the intention of encouraging collaboration between developers and hobbyists to advance the field of robotics. Many of the packages available are open source allowing anyone to use or modify code to suit their needs. Software in robotics can be incredibly complex and time consuming to create. ROS makes it possible for people to build on

other people's work rather than starting from scratch with projects, allowing time and resources to be focused on original and useful work.

A particularly attractive feature of ROS is its modular nature that allows for simple integration between packages and across machines. ROS packages contain executable files known as *nodes* that carry out certain tasks such as collecting and interpreting LiDAR data [37]. *Topics* are a medium for nodes to send or receive messages to or from other nodes [38]. When a node sends data to another node it must *publish* to a topic, when they receive data it is *subscribing* to a topic. This form of architecture allows commands and messages to be sent from multiple machines and from different platforms such as LabVIEW.

The image below is a ROS rqt_graph graph. It shows what topics and nodes are currently running and how they interact. This example is a graph of the turtle simulator package which allows the user to use the arrow keys on their keyboard to move a simulated turtle on the screen. The teleop_turtle node in blue takes input from the keyboard and publishes this data to the /turtle1/command_velocity topic. The /turtlesim node displays the turtle on screen and receives the input data by subscribing to the /turtle1/command_velocity topic and moves the turtle accordingly.

Figure 3: rqt graph in ROS showing running topics and nodes and how they interact [39]

The node and topic structure makes for simple integration between packages. An example of this is how maps built by the Hector SLAM package can be used by the Hector navigation stack package for autonomous navigation and exploration [40].

2.5 SLAM Packages in ROS

2.5.1 Hector SLAM

Hector SLAM is an open-source LiDAR-based package that can build 2D and 3D maps of an unknown environment while simultaneously determining the pose of the robot [41]. One of the most impressive features of Hector mapping is that, unlike other SLAM algorithms such as Gmapping, it does not require wheel odometry data to solve the SLAM problem [42].

To eliminate the need for odometry Hector SLAM relies on a feature-based scan matching process to build its maps. A Gaussian-Newton approach is used to eliminate laser scan measurement errors and evaluate the best-fit transformation from scan data to map data [43]. Having a robust scan matching process means that less particles (laser scan end-points) are necessary for reliable pose estimation. This is an important feature as it may potentially make up for the relatively low scan rate of lower end LiDARs.

Hector SLAM is also effective for use with platforms that exhibit roll or pitch motion such as drones or in environments with ground that is not flat. This 3D motion estimation is made possible by fusing laser scan measurements with an inertial measurement unit (IMU). The hector_pose_estimation node in the Hector SLAM package uses an Extended Kalman Filter approach to estimate the pose of the robot [44].

2.5.2 Gmapping

Gmapping is another widely used LiDAR-based package for implementing SLAM. Gmapping differs from Hector SLAM in that it requires wheel odometry data for localization, making it more appropriate for ground vehicles rather than aerial vehicles. Gmapping, like Hector SLAM uses a scan matching process for more reliable pose estimation.

2.5.3 Comparison of Hector Mapping and Gmapping

2013 research presented at the IEEE International Symposium on Safety, Security, and Rescue Robotics [45] evaluates the performance of a range of SLAM packages available with ROS. The error

between the final map and the ground truth is the key performance metric in this research and was estimated both qualitatively and quantitatively. The final map generated by HectorSLAM was found to be closer to the true ground than that produced by Gmapping. Maps produced by Hector SLAM exhibited thicker lines than that of Gmapping, which, in certain applications such as autonomous navigation/obstacle avoidance, could be detrimental to its performance. The KartoSLAM map exhibits skewed walls and there is a lack of documentation for its use compared to Gmapping and HectorSLAM. The other two methods explored in the research, KartoSLAM and GraphSLAM give maps that are severely skewed and inaccurate.

2.6 Controllers and Single Board Computers

The following section gives a comparison between single board computers and microcontrollers that could be used as the main controllers and processors for the robot.

2.6.1 Raspberry Pi

The Raspberry Pi is a widely used and low-cost single board computer that is capable of almost anything a PC is capable of. Coming stock with a 1.4GHz 64-bit quad-core processor and 1GB of RAM it also has the power to run ROS when running a Linux operating system. Many developers choose this platform as the main processing unit for their robotics projects with significant published research supporting its high performance for this purpose [46].

2.6.2 Beaglebone Black

The Beaglebone Black is a low-cost, single-board computer developed by Texas Instruments that is comparable in many ways to the Raspberry Pi. Like the Raspberry Pi it can run a variety of Linux-based operating systems as well as Android and Windows CE and is used by developers and hobbyists for robotics and electronics projects. The main advantage of the Beaglebone Black over the Raspberry Pi is that is has 92 GPIO pins, over double that of the Raspberry Pi. However, with only half the RAM and a slower processor the Beaglebone is not as powerful as the Pi, nor does it have as much community support as the Pi.

2.6.3 Arduino

Arduino boards, at first glance, may seem like single-board computers such as the Raspberry Pi or the Beaglebone, however it differs in many ways. Arduinos are microcontrollers that are programmed in C and are ideal for carrying out dedicated tasks such as pulse width modulation control of motors or for use in small electronics projects such as basic home security systems. Arduinos have a small number of both digital and analog input and output pins. Unlike the Pi and Beaglebone, they don't run an operating system and therefore cannot run multiple. If pulse width modulation (PWM) speed control of the robot's motors is necessary for this project an Arduino Nano would be ideal for this purpose as the Raspberry Pi is not capable of consistent PWM output due to the fact it runs many other processes at the same time.

2.7 TCP/IP Communication

Transmission Control Protocol (TCP) and Internet Protocol (IP) are the most extensively used protocols for communication between hosts over the local networks or over the internet. It offers a simple solution for direct, reliable and error-checked transmission of data packets of varying sizes [47]. TCP/IP communication is widely supported across a variety of platforms and makes communication between devices of different natures easy.

LabVIEW comes built-in with tools that allow the user to create their own TCP/IP clients and servers and build their own data packet structures [48]. LabVIEW is not supported on the Raspberry Pi and as such will only be used on the client side on the remote desktop for this purpose. The *socket* module available with Python will be used to create the server side on the Raspberry Pi.

3 Design

The following section details why certain design decisions were made over the course of the project. It also gives insight into the overall system architecture between the robot and the remote PC.

3.1 Design decisions

3.1.1 SLAM algorithm

While Gmapping was determined to be the superior method of SLAM in the comparison presented in section 2.5.3, the need for odometry with Gmapping makes HectorSLAM a more attractive solution for this application. This is not only because low-cost encoders may potentially reduce the performance of Gmapping due to measurement errors but also because wheel encoders are not an option with aerial vehicles.

3.1.2 Robot chassis

A differential drive robot [49] was chosen for this project as these types work best with HectorSLAM due to the fact they turn on the spot [43]. They are also easy to manoeuvre using simple commands as opposed to traditional steering systems seen on regular cars which would require the use of servo motors and forward motion to steer the vehicle. A four-wheeled robot was chosen to ensure the robot could drive straight with ease.

3.1.3 Controllers

The Raspberry Pi was chosen over the beagle bone as the main controller of the robot for a multitude of reasons. Firstly, the Raspberry Pi has significantly more processing power and RAM than the Beaglebone. Because this project required multiple high intensity programs to be running simultaneously the Raspberry Pi was the obvious choice. The Raspberry Pi also has significantly more of a support community to rely on for any problems that may have occurred.

When it was realized that PWM speed control was needed, the Arduino Nano microcontroller was chosen for this purpose. The Arduino Nano was chosen as it was physically the smallest of the Arduino controllers. The smaller number of I/O channels compared to larger Arduino controllers was not an issue for this purpose.

3.1.4 LiDAR

When deciding on which LiDAR to use, two options were considered, mounting a simple laser distance sensor such as the LiDAR-Lite v3 on a servo motor to scan 360^{o} or buying a LiDAR that does this straight out of the box. The A1M8 360^{o} LiDAR scanner by RPLiDAR was chosen as it was the easier of the two options and by far the cheapest 360^{o} LiDAR on the market. Not only are they cheap they are also well built and of high quality.

3.1.5 Motor driver

A Duinotech dual motor driver was used to control the four onboard 3-12V DC motors. This motor driver contains two L298 driver chips and allows PWM speed control and direction control of four motors individually. However, as the motors on each side work in pairs only two of the four outputs were used. Hence only a single L298 chip is shown in the wiring schematic Appendix A.

3.1.6 Power supply

The Raspberry Pi requires a supply voltage of 5V and an average supply current of 1A. It can also supply up to 1A to external devices through its USB ports [50]. The RPLiDAR, which is powered through a USB port on the Raspberry Pi has a start current of 500mA. The LiDAR scanning system has a steady-state working current of 300mA in addition to the 100mA current of the motor [51]. Between the Raspberry Pi and the RPLiDAR a power supply of 5V at 1.4A was required. A 3000mAh 5V 1.5A battery bank was chosen for this purpose not only for its low cost but also for its light-weight and small dimensions.

3.2 System architecture

Figure 4 shows a high-level representation of the overall system architecture. A wiring diagram showing how each of the hardware components interact can be found in Appendix A.

The Raspberry Pi acts as the central controller in the project and runs three main processes. The HectorSLAM package on ROS collects LiDAR data via USB to build maps using SLAM. These maps are saved periodically and opened and modified by a program called *MapSend.py* to be sent via TCP/IP communication to the remote PC and viewed on the HMI. The *CarControl.py* Python program controls

the robot by changing the state of the onboard DC motors with the motor driver in response to commands sent by the remote PC. The onboard Arduino Nano is responsible for controlling the speed of the robot via PWM and acts in accordance with commands sent by the Carcontrol.py program.

The remote PC runs two processes that are both executed when running the LabVIEW HMI. The LabVIEW HMI displays images that are received and saved by the *MapReceive.py* program. The HMI also allows the user to send control commands to the robot and view the current state of the robot, whether it be stationary or moving.

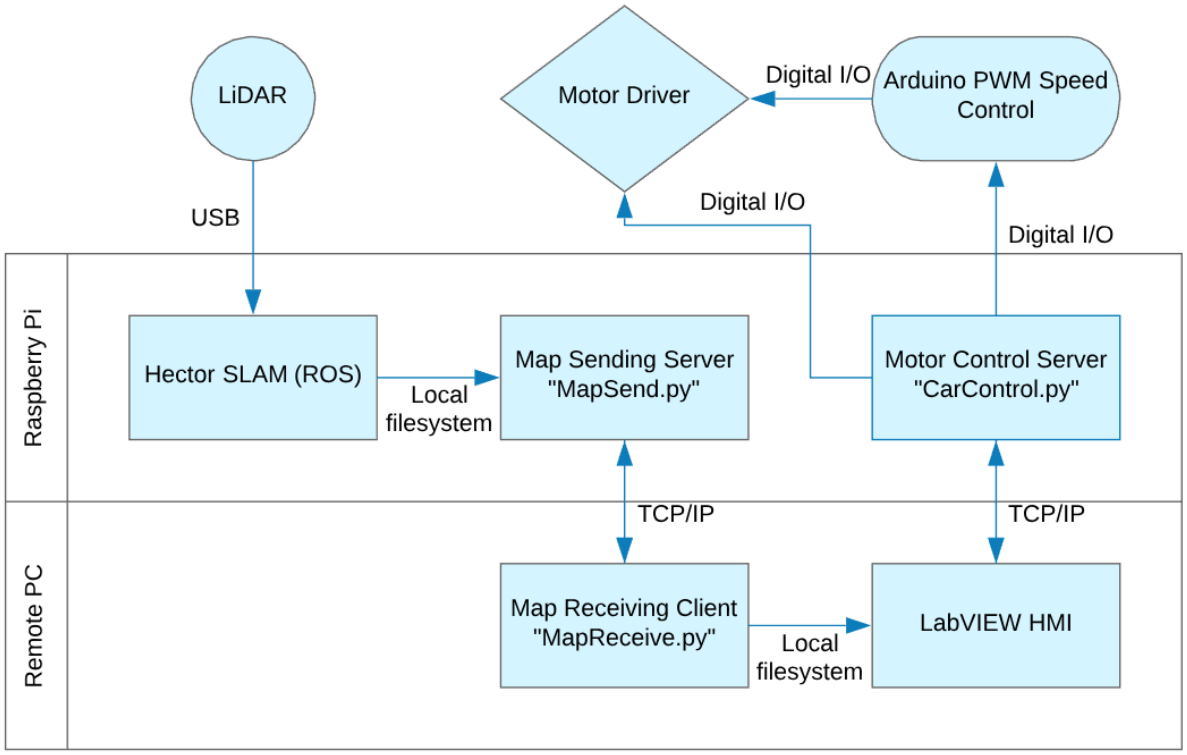


Figure 4: High-level system architecture

4 Implementation

The implementation phase of the project was carried out in a logical order that ensured each stage had the necessary prerequisites filled to make the work possible. The Raspberry Pi was set up with ROS and HectorSLAM in a way that it could be tested to work as a standalone device before developing the remote HMI.

4.1 Initial setup

To set up the robot ready for the project Ubuntu Mate version 18.10 was installed on the removable SD card on the Raspberry Pi. Ubuntu mate was chosen as it comes pre-installed with all the dependencies necessary for ROS. This streamlines the installation process and results in less issues when using ROS after installation. ROS Kinetic was then installed as per the instructions on the ROS website [52], the desktop option was chosen so basic packages were pre-installed while saving storage space.

A basic test to check that ROS was working was carried out by running the Turtlesim package. A tutorial for this was found at [37].

4.2 SLAM

4.2.1 Installing packages

After ROS was installed the robot was ready to have the SLAM packages installed. ROS packages are installed by *building* them in what is known as a *catkin workspace* [53]. To create a catkin workspace and install the desired packages one first creates an empty directory for the workspace and an empty source directory named "src". The desired package(s) can then be cloned or downloaded into this new src directory. Following this, in the catkin directory, the workspace is built with the catkin_make command.

The following commands entered to the console show the process of making the hector_cat workspace that was used in the project. It contains the *rplidar_ros* package which retrieves the LiDAR data and the *hector_slam* package which interprets this data to build maps. Both packages were sourced from Github.

```
$ mkdir -p ~/hector_cat/src
$ cd hector_cat/src
$ git clone https://github.com/robopeak/rplidar_ros.git
```

```
$ git clone https://github.com/tu-darmstadt-ros-pkg/hector_slam.git
$ cd ..
$ catkin_make -j1
```

Building workspaces can be a laborious task for the Raspberry Pi. The -j1 option next to the catkin_make command ensures that only a single processor core is used for the task which prevents crashing.

After being built the catkin workspace directory will contain the *build* and *devel* directories along with the original src directory. The devel directory contains bash files that, when sourced (executed), overlay the workspace over the ROS environment to resolve any dependencies and run scripts required for the packages to work.

Before HectorSLAM can work without odometry data the launch files contained in the catkin workspace directory must be modified. In ROS launch files contain a set of parameters relating to the way nodes operate in ROS. Launch files for nodes can be found in the respective node's directory within the src directory of a catkin workspace.

The *mapping_default.launch* file can be found in the ~hector_cat/src/RPLidar_Hector_SLAM/hector_slam/hector_mapping/launch directory. Figure 5 gives the parameters used to eliminate the need for odometry with HectorSLAM. The map resolution was decreased by increasing the cell length from the default setting of 0.025 meters to 0.07 meters. Doing this reduces the likelihood of map breaking.

Parameter	New parameter setting	Description
base_frame	base_link	Frame used for localization and transformation of laser data to map data.
odom_frame	base_link	Tells hector to use the change in the robot's location within the map as odometry data.

map_resolution	0.07	The resolution in meters is the length of the cell edges in the occupancy grid map. Increasing this decreases the resolution.

Figure 5:Parameter settings in mapping_default.launch for HectorSLAM without odometry

One final parameter that must be changed is the "use_sim_time" parameter in the *tutorial.launch* file found in /hector_slam/hector_slam_launch/launch. This must be changed from true to false as this project is a real experiment and not a ROS simulation.

4.2.2 Executing HectorSLAM

To execute the hector_slam package the rplidar_ros package must first be executed as the LiDAR data collected by rplidar_ros is used by hector_slam to build the maps. The rplidar_ros package is executed with the following commands while in the hector_cat directory.

```
$ cd hector_cat
$ source devel/setup.bash
$ sudo chmod 666 /dev/ttyUSB0
$ roslaunch rplidar_ros rplidar.launch
```

The chmod 666 command gives read/write permission for all users to access the USB ports so that the package can collect LiDAR data. The chmod command only temporarily changes access permissions. To avoid executing this command after each reboot the a udev rule was created to permanently grant read/write permissions to all users.

In a new terminal window, with the rplidar_ros process still running the following commands are executed while in the hector_cat directory to run the hector_slam package.

```
$ cd hector_cat
$ source devel/setup.bash
$ roslaunch hector_slam_launch tutorial.launch
```

An instance of Rviz will then open with an incomplete map of the robot's surroundings. The map will update each time the robot moves the distance specified by the *map_update_distance_thresh* parameter or whenever the robot rotates by an angle specified by the map_update_angle_thresh parameter. The distance and angle parameters are in meters and radians respectively and can be found and modified in the mapping_default.launch file in the ~hector_cat/src/RPLidar_Hector_SLAM/hector_slam/hector_mapping/launch directory.

For the robot to operate as a standalone machine bash scripts named "RplidarSetup.sh" and "HectorSetup.sh" were created to automate the execution process described above.

4.2.3 Geotiff Maps

The HectorSLAM package comes with an ability to save maps and trajectory data in real time with the hector_geotiff node. The parameters in Figure 6 can be modified in the geotiff_mapper.launch file, the settings used in this project are also shown in the table.

Parameter	New parameter Setting	Description
map_file_path	"(find hector_geotiff)/maps"	Sets the file path for maps to be saved to. The directory must already exist for the hector_geotiff node to save files.
map_file_base_name	"hector_slam_map"	Sets the name of the saved files. The current time is appended to the end of the file name.
geotiff_save_period	1	Sets the rate at which the maps are saved in seconds.

Figure 6: geotiff_mapper.launch relevant parameters

There is an issue with the saved Geotiff files in that the current trajectory is not saved. This means that the yellow arrow denoting the robot's current trajectory doesn't move from the initial position. The robot path is drawn as the robot moves however. The fact that the initial trajectory is correct suggests that the trajectory service is working correctly.

4.3 Motor control

4.3.1 Hardware

Initially the motor driver was powered by the same 5V power supply as the Raspberry Pi. Upon testing it was noticed that the motors were moving far too slowly, and the robot was unable to turn. This was due to there being a minimum voltage drop of 1.8V across the H-bridge (see Appendix B). To amend this issue a 9V battery pack consisting of six AA batteries was used and the robot was able to move at a reasonable speed.

After constructing the robot for the first time it was soon realized that it was not able to turn. This was because the robot has four individually driven wheels powered by low-cost DC motors. The motors could not provide the torque necessary to turn as the wheels on the turning side had to make the wheels on the non-turning side slide for the robot to pivot. To alleviate this, the wheels were mounted closer together to create a smaller area for the robot to pivot on, resulting in less torque on the motors.

4.3.2 Direction control

Direction control of the motors is achieved with a Python code named “CarControl.py” (See Appendix C) running on the Raspberry Pi that responds to commands sent from the remote machine via TCP/IP communication. After establishing a connection with the remote LabVIEW HMI the program waits for a command generated by keystrokes while on the LabVIEW front panel. The RPi.GPIO module is used to toggle the digital outputs on the Raspberry Pi and change the motor driver state accordingly. When turning, a low signal is sent via GPIO pin 20 to the Arduino to reduce the motor speed, when moving straight this pin is set to high to move the robot at full speed. Figure 7 details the Raspberry Pi GPIO connections and the purpose of such connections.

Raspberry Pi GPIO pin (board)	Raspberry Pi GPIO pin (BCM)	Connection	Purpose
29	5	In2 of motor driver	Right-hand motors forwards
35	19	In1 of motor driver	Right-hand motors backwards

33	13	In3 of motor driver	Left-hand motors forwards
31	6	In4 of motor driver	Left-hand motors backwards
36	16	LED on breadboard	Indicates robot is listening for connections
38	20	Digital pin 5 of Arduino	Tells Arduino that robot is turning to reduce speed
9	GND	Motor driver ground	Common ground
39	GND	Arduino ground	Common ground

Figure 7: Raspberry Pi GPIO connections in board and BCM notation

Figure 8 shows how the state of the Raspberry Pi outputs corresponds to motor direction, these pins are set in response to keystroke commands sent by the LabVIEW HMI.

Direction	Keystroke Command	GPIO 5	GPIO 19	GPIO 13	GPIO 6	GPIO 20
Forward	**W**	HIGH	LOW	HIGH	LOW	HIGH
Backward	**S**	LOW	HIGH	LOW	HIGH	HIGH
Left	**A**	HIGH	LOW	LOW	LOW	LOW
Right	**D**	LOW	LOW	HIGH	LOW	LOW
Stop	**Q**	LOW	LOW	LOW	LOW	HIGH

Figure 8: Rasperry Pi GPIO output states for motor direction

4.3.3 Speed control

One of the issues associated with HectorSLAM is the tendency for maps to *break* or *drift* under certain circumstances, particularly when the robot rotates too fast. Map drifting was not an issue when the robot was moving in a straight line at full speed, so it was only the turning speed that needed to be addressed to alleviate map drifting. An Arduino Nano was used to reduce the motor speed via PWM. While the Raspberry Pi does have PWM pins it was necessary to use a controller dedicated entirely to this purpose to ensure consistent PWM output.

A *turning speed* and a *straight speed* is output by the Arduino and is set depending on the state of a digital signal sent by the Raspberry Pi to digital input 5 of the Arduino. When the signal is high the robot travels in a straight line and the Arduino sends a maximum PWM signal of 255 to the motor driver. When the signal is low the robot turns, and the Arduino sends one of four PWM options to the driver. These options are necessary to ensure the motors are getting enough power to turn the robot as the battery discharges. At full charge the robot can turn with a PWM signal of 100, as it discharges this value must increase to achieve the same turning speed. The speed can be increased by selecting

the appropriate option using a jumper cable connected to the 5V supply (as seen in Appendix A) to toggle digital inputs 2, 3 or 4. The connections for the Arduino can be found in Figure 9 and the Arduino C code for this functionality can be found in Appendix D.

Pin	Mode	Connection	Purpose
D10	Output	En1 of motor driver	PWM speed control of right-hand motors
D9	Output	En2 of motor driver	PWM speed control of left-hand motors
D2	Input	5V or none	PWM option 1: Sets PWM to 100 when HIGH
D3	Input	5V or none	PWM option 2: Sets PWM to 140 when HIGH
D4	Input	5V or none	PWM option 3: Sets PWM to 180 when HIGH
-	-	-	PWM option 4: Sets PWM to 220 when D2, D3 & D4 are LOW
D5	Input	GPIO 20 of Raspberry Pi	Speed switch for turning

Figure 9: Arduino connections and PWM options

For digital signalling between devices to work the Arduino, Raspberry Pi and motor driver must all share a common ground, especially when using floating power sources such as batteries. Without a common ground a low output from the Arduino to the motor driver may still be above the threshold for the driver to switch off resulting in a 100% duty cycle and no speed control.

4.4 Viewing maps on remote machine

One of the main objectives of the project was to be able to visualize the maps from the remote machine as they are being built while the robot moves around. There were two methods explored to achieve this.

4.4.1 Running ROS across multiple machines

The first method that was tested to retrieve the maps remotely was to take advantage of ROS' ability to be run across multiple machines. The nature of ROS makes it possible to start, stop, monitor and control hundreds of ROS nodes from any machine on a distributed network with a single master. With the Raspberry Pi as the master, bidirectional connectivity was achieved between the robot and the remote PC making it possible to access the Rviz node and view maps from the remote PC.

It was noted very quickly that there was a delay of around 2 seconds between the map updating on the master and the map updating on the remote PC. At the time this seemed very significant and while

this could have potentially been fixed there were other drawbacks that made other options much more attractive. The most crucial problem with this method is that it was only possible on a Linux machine, as ROS is only compatible in its full capacity with Linux. Conversely, LabVIEW is currently not available in its full capacity for use with any distributions compatible with ROS. With one of the main objectives of the project being to create a LabVIEW HMI for ease of use, this method of viewing maps from the remote PC became much less attractive.

It may have been possible to run ROS in a virtual machine on a windows host and pass map files through to the host for LabVIEW to access. This option seemed like it may slow down the map update rate even more if it was even possible, so another method was explored first.

4.4.2 Map sending via TCP

The second and most effective method explored involved sending maps saved by the hector_geotiff node to the remote machine via TCP. A Python server named "MapSend.py" (See Appendix E) is executed on start-up of the Raspberry Pi and establishes a connection with a Python client named "MapReceive.py" (See Appendix F) on the remote machine and sends map images synchronously as the maps are built and saved by HectorSLAM.

The server first waits for the client to send the string "GO". Following this the glob module is used to scan the directory specified in the Hector_geotiff launch file for any GeoTIFF (.tif) files within, this ignores files of the other format also saved by the hector_geotiff node. The latest-saved map is then resized to fit the LabVIEW front panel and saved as a .PNG file to be read by LabVIEW after being sent to the remote client (see Figure 10).

```
21          if "GO" in str(data):              # GO is sent by client when connection is made
22              name = glob.glob('/home/jack/hector_cat/src/RPLidar_Hector_SLAM'
23                               '/hector_slam/hector_geotiff/maps/*.tif')       # Scan directory for Geotiff files
24              latest_file = max(name, key=os.path.getctime)                    # Scan for latest saved file
25              im = Image.open(latest_file)
26              im = im.resize((720, 585))       # Resize image to fit LabVIEW front panel
27              im.save('Map.png')              # Format file as .PNG to be readable by LabVIEW
```

Figure 10: Image formatting in Mapsend.py

Figure 11 shows how the file size in bytes is firstly sent to the client to be compared to the number of bytes received throughout the data transfer process. This ensures that the entire file is transferred, and no data is lost. The map image is then opened and the first 1024 bytes are read and sent to the client as hexadecimal values. The server continues to send data in 1024-byte blocks with a 250ms wait time in between until there is nothing more to be read from the file. The 250ms wait time ensures that the client has enough time to fully receive all the bytes before sending more.

```
size = os.path.getsize('/home/jack/Map.png')   # Determine size of image to be sent
size = str(size)                                # Make size variable a string
conn.sendall(size.encode('utf-8'))              # Send file size to remote PC
f = open('Map.png', 'rb')
bytestosend = f.read(1024)          # Read first 1024 bytes to be sent to remote PC
while bytestosend:                  # While there are still bytes to be read from file send next 1024
    conn.sendall(bytestosend)
    print('Sent ', len(bytestosend))
    bytestosend = f.read(1024)      # Read next 1024 bytes
    time.sleep(0.25)                # Allow 0.25 seconds for all bytes to be sent and read by client
f.close()
```

Figure 11: Sending map data to the remote client (MapSend.py)

On the client side, after sending “GO” to the server to initiate data transfer a file named “Map.png” is opened to be written to. The file size of the incoming map image is then received from the server and the total number of received bytes is reset to 0.

As seen in Figure 12 the client then proceeds to receive image data from the server in 1024-byte blocks. Each time the loop executes the total number of received bytes is recalculated and printed to the console for the user to see how much of the file has been received. The final chunk of the PNG hex dump is the IEND chunk and marks the end of the file. If this is contained in the received data, the loop breaks and the final bytes (less than 1024) are written to the file. It is then closed and duplicated for LabVIEW to access while the program repeats this process with the next updated map.

```
while totalbytes < filesize:                    # While all bytes havent been received receive next 1024 bytes
    print('receiving data...' 'bytes received=',
          totalbytes, 'file size =', filesize)
    data = s.recv(1024)
    print(len(data))
    totalbytes += len(data)         # Recalculate total bytes received
    if "END" in str(data):          # "END" is appended to the end of the last message in PNG file
        f.write(data)               # Write last <1024 bytes to file and break loop
        print('received whole file')
        break
    f.write(data)                   # Write 1024 bytes received in while loop
```

Figure 12: Receiving and writing the image data on the client side (MapReceive.py)

Initially there was an issue with the image files corrupting. This was evident only because LabVIEW could not open them but could open other .PNG files. To see where the data was being lost and/or corrupted the received data was printed to the console on the client side and the sent data was printed to the console on the server side. By comparing the two it became apparent that the client was receiving and writing the sent data to the file in smaller chunks rather than the desired 1024-byte chunks. To alleviate this a wait time was implemented between each data transmission on the server side to allow the client to receive all the data. Through trial and error, it was determined that 250ms was the shortest wait time possible to prevent the file corrupting.

4.5 LabVIEW HMI

The LabVIEW HMI is a relatively simple program that serves to bring all components of the project together into a simple, user-friendly interface. It carries out three main tasks – sending commands to and receiving data from the robot, executing the Python server for receiving the map images and displaying map images.

The commands are executed by pressing keys on the keyboard while on the LabVIEW front panel (See Figure 8 for a list of commands). When key strokes are recorded the corresponding ASCII character is sent to the robot via TCP/IP to change the motor state. The program then waits for a response from the server to confirm that the command has been executed.

The MapReceive.py program is automatically executed when the LabVIEW program is run. This is achieved by using the *System Exec* VI to run the following command in the windows terminal from the appropriate working directory.

```
python.exe MapReceive.py
```

Finally, the LabVIEW program opens the map images saved by the MapReceive.py program with the *Read PNG File* VI. It then displays the maps using the *Draw Flattened Pixmap* VI.

4.6 Robot Start-up sequence

To make the apparatus as user-friendly as possible a bash script was made to execute all the required programs on start-up. The script first executes the RplidarSetup.sh shell script, waits 20 seconds for the lidar to initialize and then runs the HectorSetup.sh shell script (See Executing HectorSLAM). After another 20 seconds the TCPCarControl.py and Mapsend.py programs are executed and the red LED on the front of the robot is toggled indicating the end of the start-up sequence. Each of the four processes are executed in separate terminals using the gnome-terminal command (See Appendix G). Gnome-terminal is used because the processes do not end when they are executed making it impossible for all of them to be executed from the same terminal. One must ensure that the Raspberry Pi is set to automatically connect to the desired Wi-Fi network to operate.

5 Analysis

The following section is an analysis of the robot's performance both quantitatively and qualitatively. The key analysis metrics include map quality, the physical performance of the robot and the communication performance between the robot and the remote PC.

5.1 Map Quality

Figure 13 shows a broken map of a rectangular-shaped room which occurred as a result of the robot turning too aggressively. By reducing the turning speed of the robot map drifting was avoidable in most cases. However, it can still be an issue particularly in environments that lack features such as long hallways. While desks and chairs may seem like very distinct features in a room, it is important to note that the LiDAR scans at a height of only 150mm above the ground. The LiDAR scan height means that in a room like that shown in Figure 13, the robot mainly has chair and table legs as reference points for the scan matching process which is most likely another cause of map drifting in this scenario.

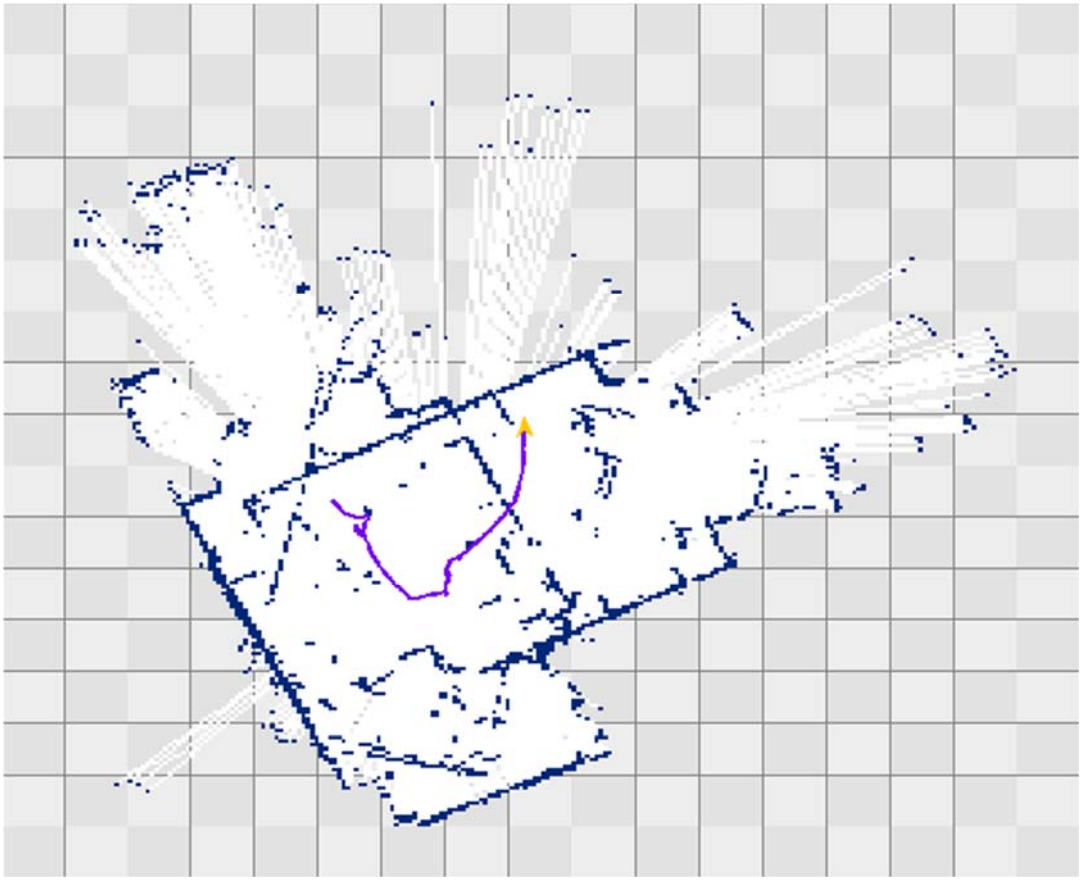


Figure 13: A broken map of a rectangular room due to aggressive turning

Glass doors and windows encountered during mapping resulted in significant complications. Laser light emitted by the LiDAR simply penetrated glass objects instead of being reflected to the LiDAR. These scenarios often resulted in HectorSLAM interpreting these scans as very far objects or not recording the scans on the map at all, resulting in a fuzzy or broken-looking walls, potentially breaking the map. This can be seen in Figure 14 circled in red. Not only do glass windows and doors result in bad readings, they can also cause HectorSLAM to crash due to lack of data coming from the LiDAR. HectorSLAM cannot simply ignore this lack of data as it is needed at all times to localize the robot, which in turn is needed to build the maps.

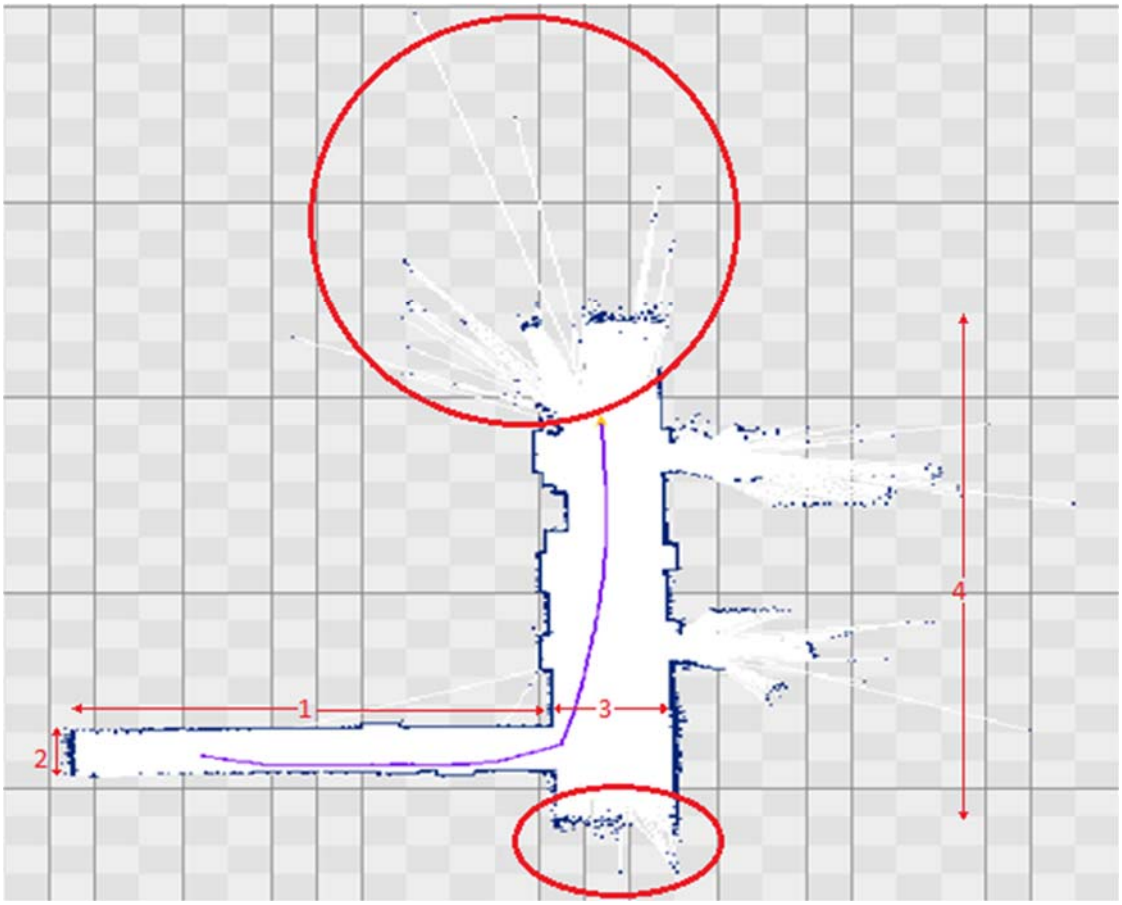


Figure 14: Map of a building with glass doors and windows

Aside from the occasional broken map and issues with glass doors and windows, the robot, when operating smoothly, was able to build accurate and detailed maps of its surroundings. Figure 16 shows a map of an entire house with no map drifting. Map detail could have been improved by driving the robot further into rooms, however, the main layout of the house is complete and accurate. The wall circled in red is a shared wall between adjacent rooms and is perfectly positioned on the map. This indicates that the robot had accurately localized itself throughout the entire mapping process.

To determine how accurate the scale of the maps produced by HectorSLAM are, the four lengths depicted in red in Figure 14 were compared to real-life measurements. Each square on the map represents $1m^2$. Figure 15 compares map measurements to real measurements. The only discrepancy between distances is on the fourth length. This discrepancy is most likely due to there being glass doors at each end of the corridor.

Length	Real measurement (m)	Approximate hector map measurement (m)
1	10.8	10.8
2	1.5	1.5
3	2.7	2.7
4	17.5	18.0

Figure 15: Comparison between map and real-life measurements.

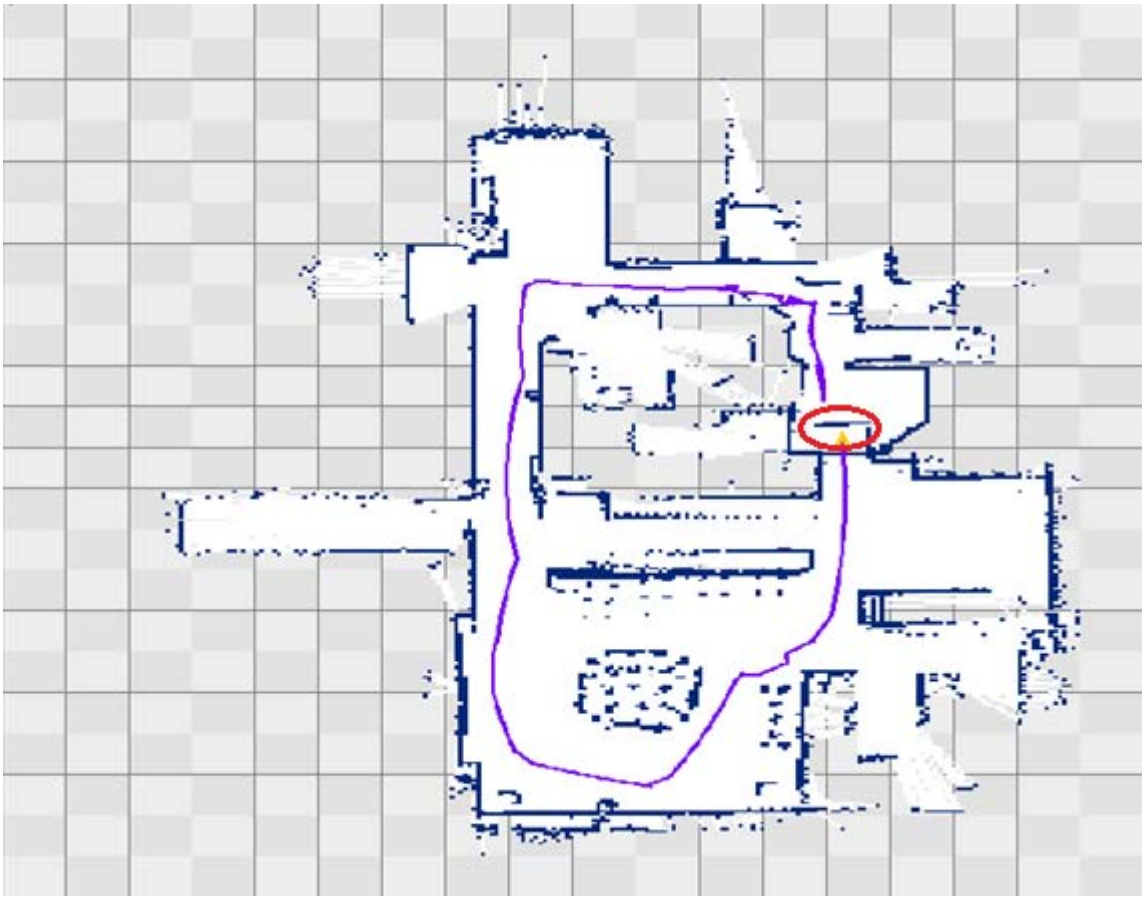


Figure 16: 2D map of a house built with HectorSLAM

In each of the maps shown, the yellow arrow (denoting the robots position) stayed at the robot's initial position. As mentioned in section 4.2.3, this is most likely a result of a recurring error while compiling the HectorSLAM catkin workspace. The hector_slam_trajectory package was installed individually in an effort to amend this issue to no avail. The developer was contacted and believed there was an error when compiling the workspace but it was unclear as to why this was happening. This does not affect the SLAM but can be confusing for the user and could make it difficult to determine the robot's location on the map when not in direct line of sight.

5.2 Physical performance

The robot's driving ability was, at times, sub optimal which often resulted in poor mapping performance. Most notably, the robot often struggled to turn on smooth surfaces such as wooden and concrete floors. In an attempt to alleviate this issue, rubber bands were wrapped around the wheels to increase traction. This was counterproductive as the motors did not provide the torque needed to turn the vehicle with all the wheels gripping well on the floor.

In situations where the wheels weren't gaining traction the robot was driven forward slightly and another attempt to turn was made. While it usually only took one or two attempts to successfully turn, this is not an appropriate solution as it is often necessary to turn on the spot due to space limitations. Sporadically turning and moving around in a small space also resulted in map breaking at times. Using a 2-wheeled robot would alleviate this issue by allowing it to pivot on its own axis. This was tested but prevented the robot driving straight due to the low quality build of the chassis.

The robot often struggled to stay on a straight path which can be seen in both Figure 14 and Figure 16. This is due to the poor build quality of the robot chassis and motors being mounted slightly off centre. This did not seem to have any effect on mapping performance.

5.3 Communications

Communications between all platforms and programs functioned as desired with exceptional reliability. By constantly comparing the number of bytes received with the total file size the Python client on the remote PC was able to consistently receive uncorrupted map images from the server on the robot.

While file transfer between the client and server was effective and reliable the transfer rate was considerably slower than desired. On average, images took between three and four seconds to transfer meaning the maps on the LabVIEW HMI took just as long to update. This is because of the 250ms wait time between data transmissions on the server side that was introduced to avoid files being corrupted. Originally it was hoped that maps would be sent fast enough such that one could

determine the position of the robot in real-time by just looking at the maps on the LabVIEW HMI. This way the operator could guide the robot even when out of direct sight or in another room.

Occasionally while in operation there is a delay between keystrokes and LabVIEW reading said keystrokes. This results in commands being sent to the robot one or two seconds after intended which often results in collisions. These collisions, when significant enough, often cause the map to break due to the sudden, violent movement. Collisions have also caused the HectorSLAM process to crash. This is most likely due to the LiDAR temporarily disconnecting on impact.

6 Future Works/Improvements

6.1 Using ROS for LabVIEW

One of the biggest and most frustrating issues with the final apparatus is the need to reboot the Raspberry Pi to restart the mapping process after the map breaks or when HectorSLAM crashes. *ROS for LabVIEW* is a set of VIs that allows the user to connect to a ROS network and initialized nodes and publish messages to topics. It may be possible to use these VIs to start and monitor the HectorSLAM process. Communicating directly between LabVIEW and ROS will also make it possible to do away with some if not all of the Python programs that were used to interface between the robot and the remote PC.

6.2 Using C++ for file transfer

The nature of Python being an interpreted language means that it is relatively slow compared to compiled languages such as C++. Rewriting the file transfer client and server in C++ may make it possible to reduce or eliminate the 250ms wait time between data transmissions on the server resulting in faster file transfer and faster map update rate on the HMI.

6.3 Drone implementation

One of the initial aims of this project was to explore SLAM methods that don't require odometry data, so the findings would be transferrable to drones. By adding an IMU and a height sensor it is possible to use HectorSLAM and the RPLiDAR with a drone for 3D mapping and SLAM.

6.4 Motor driver replacement

The motor driver chosen was sub-optimal for a few reasons. Most notably was the significant voltage drop across the H-bridge, which resulted in the need for a separate, higher voltage power supply. If another driver could be found with minimal voltage drop, the power supply for the Raspberry Pi could also be used for the motor driver. If another driver were to be used it would only be required that it supply two motors as the two motors on each side work in pairs and can be connected to the same outputs.

6.5 Robot chassis replacement

A more robust chassis that could reliably drive in a straight line would improve useability and performance of the robot. A robot tank chassis with tracks instead of wheels would be able to tackle small bumps easier and hence prevent map breaking while negotiating such obstacles.

7 Conclusion

Some aspects of robotics and computing can be incredibly complex with many layers hidden beneath the surface of the simple interfaces we use to interact with computers and machines. The Robot Operating System gives amateur and professional developers alike the power to create complex robots by providing open source tools and libraries made by other developers. It allows one to "stand on the shoulders of giants" rather than writing every code yourself, leaving more time to focus on the main task at hand. While ROS allows developers and hobbyists to better understand and work with robots, LabVIEW gives developers the tools to enable the layman to interact with their work. LabVIEW makes it simple to not only develop functional programs but to create human-machine interfaces to

allow anyone who knows how to use a computer the ability to control and monitor systems and processes or in the case of this project, robots.

The combination of the LiDAR used in this project and HectorSLAM made it possible to create impressively accurate and detailed maps. While the outcomes of this project were realized in terms of creating an easy-to-use indoor mapping robot, it did have its limitations. The robot had to be operating smoothly at all times with no sudden movements due to collisions or bumps and no aggressive turns. With the low build quality of the robot chassis itself, these types of movements were often hard to avoid. Perhaps the addition of an IMU or simply using a higher quality chassis would alleviate such an issue. One other limitation was that glass windows, doors and walls resulted in fuzzy readings or no readings at all, resulting in HectorSLAM crashing and the robot having to be rebooted.

The GeoTIFF saving functionality of HectorSLAM made it possible for these maps to be sent to a remote desktop via TCP/IP communication with a Python client and server. After being converted to the .PNG file type, these maps were able to be viewed on the LabVIEW HMI. While it was originally hoped to view the maps as they were updated in real-time (or close to), the algorithm developed in this project limited the rate at which the map images could be transferred to around 3 seconds per update. With the use of standard protocols, a faster update rate could be achieved, making it possible for the robot to be controlled confidently without being in one's direct line of sight as the user would know where the robot is by simply looking at the map. Amending the issue of the robots pose not being saved to the GeoTIFF files would also make this easier.

By using a combination of Python and LabVIEW it was possible to manoeuvre the robot remotely over a TCP/IP connection. After the robot boots, the user must simply start the LabVIEW program, which in turn starts the Python server, and with the use of the keyboard the robot can be controlled. Without LabVIEW it would most likely be difficult for the layman user to interact with this project as the Python client and commands would have to be executed via the command line or similar.

This work served to prove that the low-cost RPLiDAR is a suitable solution for implementing SLAM. This is significant because up until recently LiDAR was not an economical solution for many developers or hobbyists. With this technology becoming more and more available, progress and innovations can be made in the fields of autonomous navigation, USAR and surveying. If this work were to be transferred and adapted to drones 3D maps of abandoned mine shafts could be made safely and with ease to reduce risk to the public. These drones or even all terrain ground vehicles with LiDAR could also be deployed in disaster areas such as earthquakes to assist and reduce risk for rescue teams by giving insight into how the area has been affected.

With so many people willing to openly share work and information, to collaborate and create it is an exciting time for those in the technology field – and for the rest of the world that benefits from technology. Being open-source, the Robot Operating System and Unix operating systems are a testament to this fact and it has been an interesting and exciting journey learning about this technology with the assistance of the online community.

8 References

[1] S. Pettigrew. (2018, 16/01/2019). *Driverless cars really do have health and safety benefits, if only people knew*. Available: http://theconversation.com/driverless-cars-really-do-have-health-and-safety-benefits-if-only-people-knew-99370

[2] S. Thrun, "Simultaneous localization and mapping," in *Robotics and cognitive approaches to spatial mapping*: Springer, 2007, pp. 13-41.

[3] T. Colleens, J. J. Colleens, and D. Ryan, "Occupancy grid mapping: An empirical evaluation," in *2007 Mediterranean Conference on Control & Automation*, 2007, pp. 1-6.

[4] A. Milstein, *Occupancy Grid Maps for Localization and Mapping*. IntechOpen, 2008.

[5] Q. Lang *et al.*, "An Evaluation of 2D SLAM Techniques Based on Kinect and Laser Scanner," in *Cognitive Systems and Signal Processing*, Singapore, 2017, pp. 276-289: Springer Singapore.

[6] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem," 2002.

[7] B. Schwarz, "Mapping the world in 3D," *Nature Photonics,* vol. 4, p. 429, 07/01/online 2010.

[8] Nature, "LiDAR drives forwards," *Nature Photonics,* vol. 12, no. 8, pp. 441-441, 2018.

[9] Y. Liu and G. Nejat, "Robotic Urban Search and Rescue: A Survey from the Control Perspective," *Journal of Intelligent & Robotic Systems,* vol. 72, no. 2, pp. 147-165, 2013/11/01 2013.

[10] A. Ramirez-Serrano. (2018, 1/10/2018). *Robots to the rescue: Saving lives with unmanned vehicles*. Available: https://theconversation.com/robots-to-the-rescue-saving-lives-with-unmanned-vehicles-90516

[11] M. Pierzchała, R. Astrup, and P. Giguere, *Mapping forests using an unmanned ground vehicle with 3D LiDAR and graph-SLAM*. 2018.

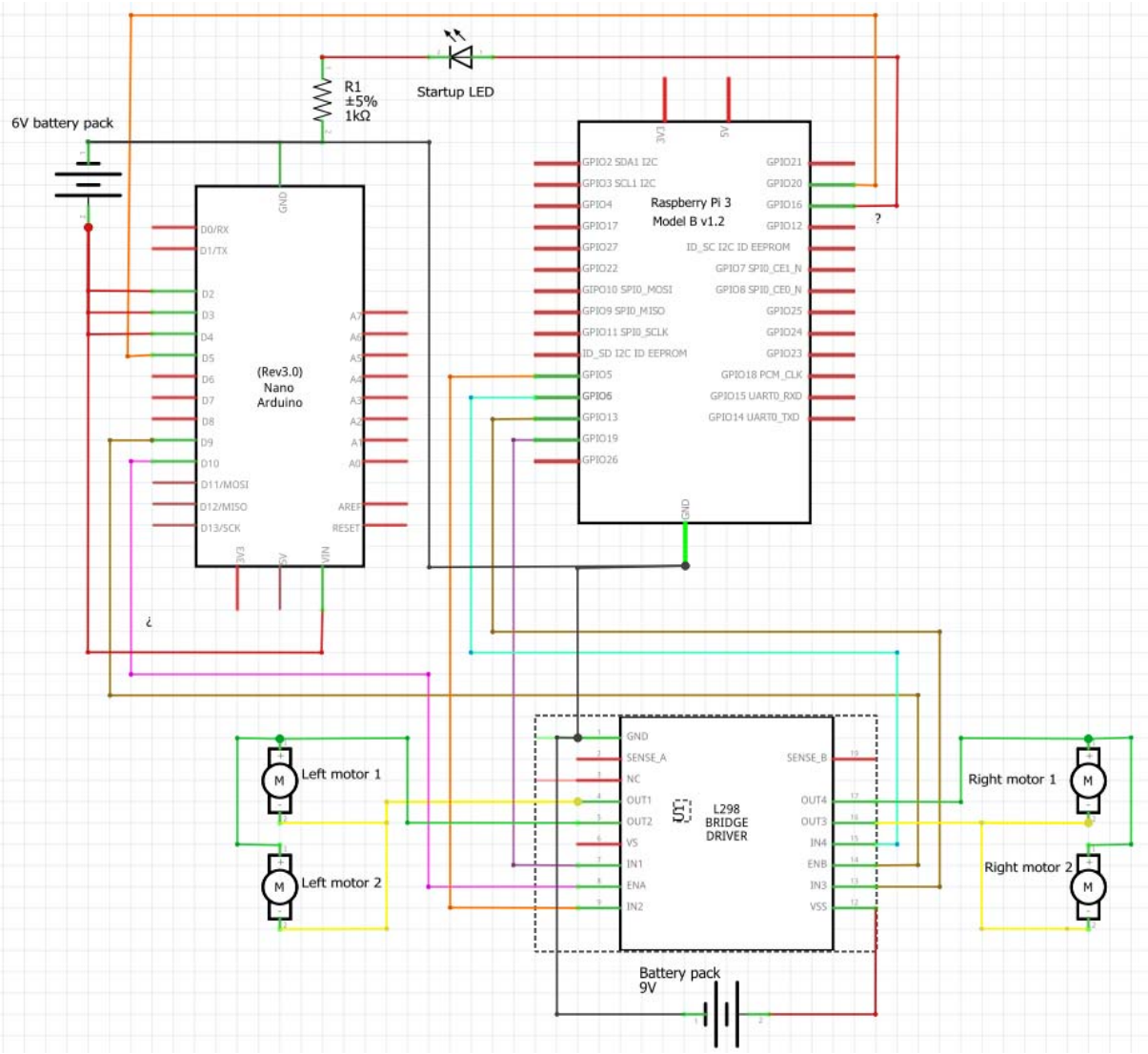
[12] (2018). *Danger of Old Mine Workings*. Available: http://www.dmp.wa.gov.au/What-makes-old-mine-workings-3210.aspx
[13] U. S. Government, "Mine Disaster at Quecreek," in *Commitee on Appropriations United States Senate*, 2 ed. Washington: U.S. Government Printing Office, 2002, pp. 107-876.
[14] ABC. (2017, 2/10/2018). *Mining report finds 60,000 abandoned sites, lack of rehabilitation and unreliable data*. Available: http://www.abc.net.au/news/2017-02-15/australia-institute-report-raises-concerns-on-mine-rehab/8270558
[15] X. Wang, "2D Mapping Solutions for Low Cost Mobile Robot," Master of Science Masters, Computer Science and Communication, KTH Royal Institute of Technology, 2013.
[16] D. Pagac, E. M. Nebot, and H. Durrant-Whyte, "An evidential approach to probabilistic map-building," in *Reasoning with Uncertainty in Robotics*, Berlin, Heidelberg, 1996, pp. 164-170: Springer Berlin Heidelberg.
[17] N. Gageik, T. Müller, and S. Montenegro, "Obstacle Detection and Collision Avoidance Using Ultrasonic Distance Sensors for an Autonomous Quadrocopter," Aerospace Information Technology, University of Würzburg, 2012.
[18] (2017). *Green, waveform lidar in topo-bathy mapping – Principles and Applications*. Available: https://www.ngs.noaa.gov/corbin/class_description/Nayegandhi_green_lidar.pdf
[19] BMW. (ND). *Autonomous driving: Digital measuring of the world*. Available: https://www.bmw.com/en/innovation/mapping.html
[20] A. Charlton. (2018). *Lidar, radar and cameras: This is the tech used by autonomous Ubers to see the world around them*. Available: https://www.gearbrain.com/autonomous-uber-car-technology-2551793303.html
[21] R. Mitchell, "LiDAR costs $75,000 per car. If the price doesn't drop to a few hundred bucks, driverless cars won't go mass market.," in *Los Angeles Times*, ed. San Fransisco, 2017.
[22] DFRobot. (2018). *RPLIDAR A1M8 - 360 Degree Laser Scanner Development Kit*. Available: https://www.dfrobot.com/product-1125.html
[23] M. Cooper, J. Raquet, and R. Patton, "Algorithm on Converting a 2D Scanning LiDAR to 3D for use in Autonomous Indoor Navigation," *Adc Robot Autom,* vol. 7, p. 184, 2018.
[24] A. Hawkins;, "Elon Musk still doesn't think LIDAR is necessary for fully driverless cars," in *The Verge*, ed, 2018.
[25] Y. Wu, F. Tang, and H. Li, "Image-based camera localization: an overview," *Visual Computing for Industry, Biomedicine, and Art,* journal article vol. 1, no. 1, p. 8, September 05 2018.
[26] Y. Cui, S. Schuon, D. Chan, S. Thrun, and C. Theobalt, "3D shape scanning with a time-of-flight camera," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 1173-1180.
[27] G. J. Iddan and G. Yahav, "Three-dimensional imaging in the studio and elsewhere," in *Photonics West 2001 - Electronic Imaging*, 2001, vol. 4298, p. 8: SPIE.
[28] Microsoft. (ND, 12/09/2018). *Kinect for Windows*. Available: https://developer.microsoft.com/en-us/windows/kinect
[29] Intel. (ND, 12/09/2018). *Intel Realsense Technology*. Available: https://software.intel.com/realsense
[30] P. Moghadam, W. S. Wijesoma, and F. Dong Jun, "Improving path planning and mapping based on stereo vision and lidar," in *2008 10th International Conference on Control, Automation, Robotics and Vision*, 2008, pp. 384-389.
[31] I. Z. Ibragimov and I. M. Afanasyev, "Comparison of ROS-based visual SLAM methods in homogeneous indoor environment," in *2017 14th Workshop on Positioning, Navigation and Communications (WPNC)*, Bremen, Germany, 2017, pp. 1-6.
[32] K. Kamarudin, S. M. Mamduh, A. Y. Shakaff, and A. Zakaria, "Performance analysis of the Microsoft Kinect sensor for 2D Simultaneous Localization and Mapping (SLAM) techniques," (in eng), *Sensors (Basel),* vol. 14, no. 12, pp. 23365-87, Dec 5 2014.

[33] K. Kamarudin *et al.*, "Improving performance of 2D SLAM methods by complementing Kinect with laser scanner," in *2015 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*, 2015, pp. 278-283.
[34] Z. Tao, P. Bonnifait, V. Frémont, and J. Ibañez-Guzman, "Mapping and localization using GPS, lane markings and proprioceptive sensors," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 406-412.
[35] J. Carlson, "Mapping large, urban environments with gps-aided slam," Robotics Doctor of Philosophy, The Robotics Institute Carnegie Mellon University, Pittsburg, Pennsylvania, 2010.
[36] W. Rahiman and Z. Zainal, *An overview of development GPS navigation for autonomous car*. 2013, pp. 1112-1118.
[37] ROS. (2018, 14/09/2018). *Understanding ROS Nodes*. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
[38] ROS. (2014, 14/09/2018). *Topics*. Available: http://wiki.ros.org/Topics
[39] ROS. (2018). *Understanding Topics*. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics
[40] ROS. (2013, 14/09/2018). *Hector_navigation*. Available: http://wiki.ros.org/hector_navigation
[41] S. Kohlbrecher. (2012, 7/09/2018). *hector_mapping*. Available: http://wiki.ros.org/hector_mapping
[42] ROS. (2018, 09). *gmapping*. Available: http://wiki.ros.org/gmapping
[43] S. Kohlbrecher, J. Meyer, T. Graber, K. Petersen, U. Klingauf, and O. Von Stryk, *Hector Open Source Modules for Autonomous Mapping and Navigation with Rescue Robots*. Springer, 2014, pp. 624-631.
[44] S. Kohlbrecher, O. v. Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable SLAM system with full 3D motion estimation," in *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2011, pp. 155-160.
[45] J. M. Santos, D. Portugal, and R. P. Rocha, "An evaluation of 2D SLAM techniques available in Robot Operating System," in *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2013, pp. 1-6.
[46] M. Al-Nesf, H. Aagela, and V. Holmes, "Using the Raspberry Pi with ROS as The Main Compute Unit for Turtlebot Robot," 2017.
[47] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications,* vol. 22, no. 5, pp. 637-648, 1974.
[48] N. Instruments. (2015, 4/10/2018). *Basic TCP/IP Communication in LabVIEW*. Available: http://www.ni.com/white-paper/2710/en/
[49] S. M. LaValle, "Differntial Models," in *Planning Algorithms*Cambridge: Cambridge University Press, 2006, pp. 726-728.
[50] R. Pi. (ND). *Power Supply*. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md
[51] SLAMTECH. (2016). *RPLiDAR A1 Introduction and Datasheet*. Available: https://www.robotshop.com/media/files/pdf/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf
[52] ROS. (2017, 12/11/2018). *Ubuntu install of ROS Kinetic*. Available: http://wiki.ros.org/kinetic/Installation/Ubuntu
[53] ROS. (2017, 12/11/2018). *Create a workspace for catkin*. Available: http://wiki.ros.org/catkin/Tutorials/create_a_workspace

9 Appendix

9.1 Appendix A

Wiring diagram between components on the robot.


6V battery pack
R1
±5%
1kΩ
Startup LED
(Rev3.0)
Nano
Arduino
Raspberry Pi 3
Model B v1.2
L298
BRIDGE
DRIVER
Left motor 1
Left motor 2
Right motor 1
Right motor 2
Battery pack
9V

9.2 Appendix B

Dual full-bridge driver electrical characteristics state that there is a minimum voltage total drop, V_{CEsat} of 1.8V.

ELECTRICAL CHARACTERISTICS (V_S = 42V; V_{SS} = 5V, T_j = 25°C; unless otherwise specified)

Symbol	Parameter	Test Conditions	Min.	Typ.	Max.	Unit
V_S	Supply Voltage (pin 4)	Operative Condition	V_{IH} +2.5		46	V
V_{SS}	Logic Supply Voltage (pin 9)		4.5	5	7	V
I_S	Quiescent Supply Current (pin 4)	V_{en} = H; I_L = 0 V_i = L V_i = H		13 50	22 70	mA mA
		V_{en} = L V_i = X			4	mA
I_{SS}	Quiescent Current from V_{SS} (pin 9)	V_{en} = H; I_L = 0 V_i = L V_i = H		24 7	36 12	mA mA
		V_{en} = L V_i = X			6	mA
V_{iL}	Input Low Voltage (pins 5, 7, 10, 12)		–0.3		1.5	V
V_{iH}	Input High Voltage (pins 5, 7, 10, 12)		2.3		VSS	V
I_{iL}	Low Voltage Input Current (pins 5, 7, 10, 12)	V_i = L			–10	µA
I_{iH}	High Voltage Input Current (pins 5, 7, 10, 12)	Vi = H ≤ V_{SS} –0.6V		30	100	µA
V_{en} = L	Enable Low Voltage (pins 6, 11)		–0.3		1.5	V
V_{en} = H	Enable High Voltage (pins 6, 11)		2.3		V_{SS}	V
I_{en} = L	Low Voltage Enable Current (pins 6, 11)	V_{en} = L			–10	µA
I_{en} = H	High Voltage Enable Current (pins 6, 11)	V_{en} = H ≤ V_{SS} –0.6V		30	100	µA
$V_{CEsat\ (H)}$	Source Saturation Voltage	I_L = 1A I_L = 2A	0.95	1.35 2	1.7 2.7	V V
$V_{CEsat\ (L)}$	Sink Saturation Voltage	I_L = 1A (5) I_L = 2A (5)	0.85	1.2 1.7	1.6 2.3	V V
V_{CEsat}	Total Drop	I_L = 1A (5) I_L = 2A (5)	1.80		3.2 4.9	V V
V_{sens}	Sensing Voltage (pins 1, 15)		–1 (1)		2	V

9.3 Appendix C

Python server for motor direction control via TCP/IP. Repetitive blocks of code are collapsed for simplicity – CarControl.py

```
#!/usr/bin/env python3

import RPi.GPIO as GPIO
import socket

#  Setup GPIO Pins
GPIO.setwarnings(False)   # Don't allow warnings to stop program
GPIO.setmode(GPIO.BCM)    # Reference pins by broadcom channel number
GPIO.setup(5, GPIO.OUT)   # Wired to in4 of driver (Right forward)
GPIO.setup(19, GPIO.OUT)  # Wired to in3 of driver (Right backward)
GPIO.setup(6, GPIO.OUT)   # Wired to in5 of driver (Left backward)
GPIO.setup(13, GPIO.OUT)  # Wired to in6 of driver (Left forward)
GPIO.setup(16, GPIO.OUT)  # LED to indicate startup is complete
GPIO.setup(20,GPIO.OUT)   # Turning indicator - HIGH is straight, LOW is turning

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # AF_INET = ipv4, SOCK_STREAM = TCP
ip = '127.0.0.1'
port = 8089
address = (ip, port)
s.bind(address)
s.listen(1)
print('Started listening on addr:', ip, 'and port:', port)
client, addr = s.accept()       # Accept connection with client
print('Got connection from addr', addr[0], 'on port:', addr[1])
client.sendall(b'Connection established')
while True:
    client, addr = s.accept()      # Accept new connection each loop as LabVIEW closes port
    data = client.recv(1024)
    if data == "disconnect":
        client.sendall(b'disconnect')
        client.close()
        break
# FORWARD --- 5 and 13 HIGH , 6 and 19 LOW
    elif "w" in str(data):
        client.sendall(b'forward')
        GPIO.output(5, GPIO.HIGH)
        GPIO.output(13, GPIO.HIGH)
        GPIO.output(6, GPIO.LOW)
        GPIO.output(19, GPIO.LOW)
# BACKWARD --- 6 and 19 HIGH , 5 and 13 LOW
    elif "s" in str(data):...
# STOP--- 6 , 19 , 5 and 13 LOW
    elif "q" in str(data):...
# RIGHT--- 6 and 13 HIGH, 5 and 19 LOW
    elif "d" in str(data):...
# LEFT--- 5 and 19 HIGH, 6 and 13 LOW
    elif "a" in str(data):...

    else:
        client.sendall(b'Key command must be w, s, d, a or q.')

```

9.4 Appendix D

Arduino code for motor speed control. Filename: “speed control”

```
int enA = 10; // Left motors PWM
int enB = 9;  // Rigth motors PWM
int in1 = 5;  // Speed switch, Turning speed = LOW, Straight speed = HIGH
// 3 PWM options determined by jumper position
int S1 = 2;   // Turning speed 1
int S2 = 3;   // Turning speed 2
int S3 = 4;   // Turning speed 3

void setup()
{
  pinMode(enA, OUTPUT);
  pinMode(enB, OUTPUT);
  pinMode(in1, INPUT);
  pinMode(S1, INPUT);
  pinMode(S2, INPUT);
  pinMode(S3, INPUT);

}

void loop()
{
 if(digitalRead(in1) == LOW)      // If robot is turning output one of 4 PWM options to driver
   {
      if(digitalRead(S1) == HIGH)
        {
          analogWrite(enA, 100);
          analogWrite(enB, 100);
        }
      else if(digitalRead(S2) == HIGH)
        {
          analogWrite(enA, 140);
          analogWrite(enB, 140);
        }
      else if(digitalRead(S3) == HIGH)
        {
          analogWrite(enA, 180);
          analogWrite(enB, 180);
        }
      else
        {
          analogWrite(enA, 220);
          analogWrite(enB, 220);
        }
   }
  else    // If robot is going straight output max PWM
    {
    analogWrite(enA, 255);
    analogWrite(enB, 255);
    }
}
```

9.5 Appendix E

Python server for sending map images to remote client - “Mapsend.py”.

```
#!/usr/bin/env python3

import glob
from PIL import Image
import os
import socket
import time

port = 50000
s = socket.socket()
host = "127.0.0.1"
s.bind((host, port))
s.listen(5)

print('Server listening....')
conn, addr = s.accept()
data = conn.recv(1024)
print('Server received', repr(data))
while True:
    try:
        if "GO" in str(data):              # GO is sent by client when connection is made
            name = glob.glob('/home/jack/hector_cat/src/RPLidar_Hector_SLAM'
                             '/hector_slam/hector_geotiff/maps/*.tif')       # Scan directory for Geotiff files
            latest_file = max(name, key=os.path.getctime)                    # Scan for latest saved file
            im = Image.open(latest_file)
            im = im.resize((720,585))       # Resize image to fit LabVIEW front panel
            im.save('Map.png')             # Format file as .PNG to be readable by LabVIEW
            size = os.path.getsize('/home/jack/Map.png')   # Determine size of image to be sent
            size = str(size)                                # Make size variable a string
            conn.sendall(size.encode('utf-8'))              # Send file size to remote PC
            f = open('Map.png', 'rb')
            bytestosend = f.read(1024)         # Read first 1024 bytes to be sent to remote PC
            while bytestosend:                 # While there are still bytes to be read from file send next 1024
                conn.sendall(bytestosend)
                print('Sent ', len(bytestosend))
                bytestosend = f.read(1024)     # Read next 1024 bytes
                time.sleep(0.25)               # Allow 0.25 seconds for all bytes to be sent and read by client
            f.close()
            print('Done sending')
    except OSError:                            # If error reading .tif file ignore error to not crash
        print("OSError")
```

9.6 Appendix F

Python client for receiving map images from server on the Raspberry Pi – “MapReceive.py”

```python
#!/usr/bin/env python3

import socket
from shutil import copyfile

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = input()  # Asks the user for the host IP address (from LabVIEW)
port = 50000

s.connect((host, port))
s.sendall(b"GO")        # Tells server to begin sending files
while True:
    with open('Map.png', 'wb') as f:                  # Open file to be written to
        print('file opened')
        filesize = s.recv(1024).decode('utf-8')         # Get image filesize from server
        filesize = int(float(filesize))                 # Set to integer
        print(filesize)
        totalbytes = 0                                  # Reset total received bytes to 0
        while totalbytes < filesize:                    # While all bytes havent been received receive next 1024 bytes
            print('receiving data...' 'bytes received=',
                  totalbytes, 'file size =', filesize)
            data = s.recv(1024)
            print(len(data))
            totalbytes += len(data)         # Recalculate total bytes received
            if "END" in str(data):          # "END" is appended to the end of the last message in PNG file
                f.write(data)               # Write last <1024 bytes to file and break loop
                print('received whole file')
                break
            f.write(data)                   # Write 1024 bytes received in while loop
        print('receiving data...' 'bytes received =', totalbytes, 'file size =', filesize)
        f.close()                           # Close file
        copyfile('Map.png', 'MapCopy.png')  # Duplicate file so only one is opened by Python and one by Labview
```

9.7 Appendix G

Bash script for automating start up process without user input. Gnome-terminal allows all processes to run individually as they cannot all be run from the same terminal.

```
#!/bin/bash
gnome-terminal -x ./RplidarSetup.sh
sleep 20
gnome-terminal -x ./HectorSetup.sh
sleep 20
gnome-terminal -x ./TCPCarControl.py
gnome-terminal -x ./Mapsend.py
```