Indoor Autonomous Navigation System for Low-cost Mobile Robots

University College Dublin

School of Electronic and Electrical Engineering

MEngSc Project EEEN40290

Student: Aidan O'Sullivan

Supervisor: Brian Mulkeen

September 2021

Abstract

This paper focuses on the development of a mobile robot capable of autonomously navigating in indoor settings. Using a digital map and sensor data, the project strives to develop a system that can trajectory plan in real-time in a dynamic environment. This technology has a vast array of potential uses and benefits in industry, but the current market price and immature development stages of these robots reduce its appeal at present. This system is implemented using low-cost hardware and avoids expensive sensors such as LiDAR, that negatively impact its affordability. The autonomous navigation technology could have a wide variety of applications if its accessibility was improved, and these are examined. The testing carried out provides an insight into its feasibility and reviews possible optimisations.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor Brian Mulkeen for the guidance, patience, and motivation that he provided me throughout the project. Luke Dalton was instrumental in providing support with many hardware problems encountered throughout

Table of Contents

1. Introduction

The last number of decades have seen a development drive in autonomous mobile robots (AMR). Spurred on by the potential to reduce human error and improve efficiency, the number of robots installed in industry has been rapidly on the rise. This growth is expected to continue, with a value of USD 1.9 billion in 2019 and is projected to increase at a compound annual growth rate (CAGR) of 19.6 percent from 2020 to 2027 [1]. AMRs can automate a broad range of labour-intensive activities in industrial environments and enhance productivity. At present, the current price of a single AMR is on average around $30,000, putting them beyond the financial means of many businesses to deploy in large scale. The aim of this project is to create a more feasible and low-cost autonomous machine that can navigate through irregular surroundings effectively and safely.

Navigation is a difficult undertaking that depends on creating an internal representation of space, based on familiar landmarks and powerful visual processing. It should support both continual self-localization and a representation of the destination. Indoor and outdoor navigation are two separate types of autonomous navigation for mobile robots. For localisation, outdoor navigation depends heavily on the Global Positioning System (GPS). For indoor environments, such as factory settings that the design of this project focused on, GPS cannot be used [2]. The required signals are obstructed by buildings, causing the connection to be unavailable or too unreliable. Simultaneous Localization and Mapping (SLAM) was explored as a reliable solution to overcome this issue in indoor environments. SLAM methods create a map of an unknown environment and locate the robot inside of it, with a heavy emphasis on real-time operation. Numerous SLAM algorithms have been developed over the years, that carry out the core tasks of tracking, mapping, localisation, and loop closing. Many of these were developed as computationally demanding algorithms, to be used with expensive sensors such as LIDAR. To increase the ubiquity and accessibility of autonomous robots to all businesses, low-cost options must be developed. The project's goal is to create a single robot

design with the lowest possible bill of materials and computing cost. An existing depth sensing RGBD camera and Roomba robot were repurposed to complete this task. A camera was chosen as it is much less expensive than LIDAR. This sensor also provides image recognition, allowing visual signage in the environment to be interpreted, assisting with localisation. A Raspberry Pi 4 Model B was acquired to provide a cheap but powerful processor capable of running the image processing tasks to achieve SLAM and navigation, without the need for external computation power. The Robotic Operating System (ROS) provided an environment where the SLAM and navigation tasks could be implemented on the Raspberry Pi simultaneously, while abstracting the robot's hardware.

1.1 Overview

This project explores an inexpensive implementation of an indoor AMR, carried out with a visual SLAM algorithm and a ROS based navigation system. Section 2 reviews the current state of the art autonomous indoor driving technology and common techniques used. Section 3 outlines the work carried out over the course of the project and is split into a hardware and software section. Section 4 presents the results of tests carried out on the system to examine its capabilities. Drawing from those results, section 5 details the limitations of the system, with the potential solutions described in section 6 if the project were to be developed further. Finally, section 7 concludes with an assessment on the achievements of the project.

2. Literature Review

This section synopsises the background research carried out prior to commencing the practical work encompassed by this project. A review into how mobile robots have begun to shape the industrial landscape is presented, with an emphasis on increasingly integrated autonomy. The techniques used for localisation and navigation in these systems are examined. In particular, the methods often utilized in indoor settings for mapping, localisation, and path planning.

2.1 Autonomous Mobile Robots

The concept of autonomous driving technologies has garnered considerable attention in both academia and industry over the past decade, due to its exciting potential to revolutionize industry and driving on our roads. This focus is primarily attributable to the number of crashes caused by driver inattention or negligence, and its promise to vastly improve efficiency. Considerable progress is being made but such technology is only operational and trusted in controlled and limited environments, suggesting that even fully autonomous vehicles won't be present on our roads for several years. To fulfil its safety and efficiency potential, widespread action is being taken to ensure that autonomous driving come to fruition, and it remains a very active research area that is heavily resourced [3].

In the industrial sector robotics has already greatly assisted humans and enabled ever-increasing large-scale manufacturing. At present new autonomous driving technology is beginning to be applied to vastly augment capacity. The traditional automated and guided robots have drastically increased efficiency and output. However, these are restricted to predetermined paths and limited by a lack of flexibility to vary from their rigid program. They are mostly unaware of their broader surroundings, with limited sensory input and predictable behaviour [4]. Expensive infrastructure

must be installed in the factory to guide the vehicle along its fixed path. With the advent of industry 4.0, AMRs are seen as an integral component to the smart factory [5]. This new age of manufacturing is concerned with the development of intelligent products and manufacturing processes. The current unfolding evolution is a response to the demands of accelerated product development, dynamic manufacturing, and an increasingly complex environment [6]. AMRs fulfil a broad range of functions, ranging from the most common purposes of transporting, loading, and unloading payloads to cleaning and surveillance assignments. Variabilities in the environment are not an issue due to its decentralized decision-making, facilitating the system to react dynamically to changes. Using a map and its ability to sense its surroundings, AMRs possess the ability to discern the most efficient route to reach its destination. They can safely manoeuvre around unfamiliar obstacles, utilizing the optimal alternative path within unobstructed floor space. As output increases there are more moving parts within the factory, with guided vehicles this can lead to constraints. Traffic jams may occur on their tracks, that will require manual intervention to recommence operations [7]. This issue highlights the overwhelming advantage of AMRs, as they can recalculate a route within all available floor space, using superior built in intelligence. These robots also have the advantage of not requiring wires, magnetic stripes and other expensive modifications to existing infrastructure, that were relied upon by previous generation technology to guide them collision free through the factory floor. They can execute a range of jobs at many locations, autonomously adjusting to changing conditions and production requirements

Figure 1: Omron's Autonomous Transport Robot (source: industrial.omron.eu)

Fully functional AMRs are becoming more widespread and accessible within industry, due to an increasing number of manufacturers. Omron has been amongst the key players in this market and has three robots designed for indoor transport. The smallest is designed to carry up to 90 kg and the largest can move payloads up to 1500 kg. The heavy load capability would be useful in aviation or automotive manufacturing, where sizeable parts need to be transported. It is equipped with 360° LiDAR scanners to provide a depth representation of all objects in its vicinity. These lasers are also used for flexible mapping, whereby the robot can build and constantly update a map as it moves around the environment. The user can also add in pickup and drop-off points, as well as restricted zones. Dynamic tracking is another intelligent feature that enables Omron robots to function well in an environment with many moving parts. This feature allows the AMR to track other vehicles or people and to project their motion. The algorithm will course correct to pass behind the path of the person or vehicle. It can determine whether a collision is imminent. If in danger, the robot either comes to a halt or takes evasive measures [8]. All these Omron robots have a market price of between $30,000-$50,000 each, presenting a considerable initial cost for a business, particularly if a full fleet is required [9]. Škoda Auto is an example of an established company with considerable means, who is making strides towards industry 4.0, by incorporating an Omron AMR at one of their plants. After a guided tour of their plant, it grasped the layout of the environment and route between its transport points, demonstrating its ability to learn and adapt quickly. The robot's utility to the company is evident, as it makes 120 trips each day and covers a total distance of 35 kilometres [10].

Figure 2: SMP Robotics' Surveillance Robot (source: SMPRobotics.com)

AMR applications are not limited exclusively to the transport of materials. SMP Robotics design them for the purpose of surveillance and inspection. They can patrol areas to provide professional security monitoring and also fulfil inspection roles in the oil, gas or energy industry, where they can detect leaks and carry out other checks on the facilities. This removes the need for humans to enter these inhospitable and dangerous environments. The robots can carry out all these tasks without operator supervision. SMP Robotics use a stereo vison module to generate a dense disparity map and filtering methods remove incorrect correspondences. A disparity map calculates depth information from two dimensional images. It does so by calculating the difference in location between an object in two corresponding images. These are obtained from its stereo vision, consisting of two cameras, which are displaced horizontally. This method is similar to how humans perceive depth using binocular vision [11]. Unlike Omron's AMRs that utilize LiDAR, SMP Robotics use a purely visual navigation system. This design decision was motivated by the cost of installing LIDAR in unmanned vehicles being too high. Additionally, multibeam laser scanners and the high-capacity computers that analyse data from them, demand a substantial amount of electricity. The vision system discerns its orientation and position by comparing current images with data obtained during the training run of the environment. The map is obtained using a fusion of sensor data for optimal accuracy, from a stereo camera, mechanical odometer and an Inertial Measurement Unit (IMU) but is not continuously updated during normal operation. In some SCP Robotics models designed for outdoor use, GPS is integrated into the system but not relied on exclusively, due to unreliable signals in the remote areas the robot might be deployed. The navigation module computes the position of the robot and correlates it with its digital map. Object recognition of the terrain ahead assists with calculating the best route. Obstacle avoidance is also provided by its robust vision system and it will calculate a bypass route [12]. The main disadvantage of a purely vision-based navigation and localisation system is that the robot will not perform well in a featureless environment, as it relies on distinctively shaped objects or colour differences to identify its position. For this reason, the robot should be kept in areas where distinguishing objects are continually present, to provide sufficient

reference or else markers can be placed along the route [13]. SMP robotics lease these AMRs for around USD $1,500 per month, making them one of the most affordable options on the market, particularly for short term use.

2.2 Sensors

Perception is a critical component in the development of any intelligent system, to gather information from its surroundings, sense objects in its vicinity, and determine its relative position. Sensors enable robot localisation to be performed autonomously. They can be split into two categories, proprioceptive and exteroceptive. Proprioceptive sensors record parameters internally in the system, such as wheel encoder, gyroscopes, battery level and motor speed. Exteroceptive sensors are used to acquire data from their surroundings and the world at large. These include technologies such as RADAR, LIDAR, GPS, and computer vision, that AMRs use to build awareness of their environment [14]. The perception module of the vehicle is made up of an array of these sensors that work together to monitor and record the environment. LiDAR is being used by many companies in the autonomous vehicle and AMR market as it provides a full 360◦ view of the vehicle, full of extremely dense information. It is also not as adversely affected by featureless surroundings or environmental and lighting conditions as camera vision. The main downside of this technology is the considerable expense of installing it. For this reason, some notable companies such as Tesla have opted for camera-vision based systems, which cost far less [15]. Furthermore, cameras hold many advantages, as they can visually recognize objects to react to in a human like manner. Most of the existing road and industrial system is already designed to be navigated with optical perception using traffic lights, signage and markings, implying that vision systems should be able to integrate easily with our existing infrastructure [16]. Sensor fusion is being utilized as a viable option to overcome the pitfalls and exploit the strengths of these perception technologies. The resulting information is more reliable than if the sources were used individually [17]. For this project, the AMR used an RGBD camera to perceive its surrounding. To sense its own motion, a fusion of odometry using an

Extended Kalman Filter was used from its wheel encoders and visual odometry obtained from the camera. The odometry data provides information on the robot's change of position over time.

2.3 Robot Operating System (ROS)

ROS is an open-source framework for developing robotics software. It performs the functions that you would expect from an operating system, such as hardware abstraction, low-level device control, process management and message-passing between processes and package management. It is the most used robot development platform in the world and works well on microcontroller devices such as Raspberry Pis, that are using a Linux based operating system compatible with ROS. For this reason, it is not a standard operating system and instead offers a communication layer above the host operating system for robotics software development [18].

2.4 Simultaneous Localization and Mapping (SLAM)

SLAM was chosen as the method to achieve autonomous positioning of the AMR. The system has the advantage of not requiring GPS, making it suitable for indoor settings. The localisation aspect informs the AMR on a combination of its position and orientation relative to some coordinate system (pose), and its location. The mapping stage builds a representation of the unknown region that it observes. [19] Due to the low cost, visual SLAM systems with a camera as the primary sensor are becoming increasingly popular. This algorithm will usually use a sensor feed from a stereo or RGBD camera. Place recognition is a critical element in a SLAM system for loop closing (detecting when the sensor returns to a previously visited location and correcting the error propagation) and relocating the camera after a tracking failure. Both of these can be caused by violent movement, temporary loss of vision or re-initialising the system [20]. RTABmap was the SLAM algorithm chosen as it is designed to take input from an RGBD camera and external odometry, which makes it compatible with the existing available low-cost hardware. The software was also well integrated with ROS. RTABmap is a graph-based SLAM, that builds the map using graphs, consisting of nodes

and links, and depth information from the camera. A node takes account of the robot's pose, the raw RGBD sensor data, visual words (identifiable part of an image, that includes information on the features e.g colour, shape or texture) and proximity information. A link is a fixed transformation between the two nodes. Links between nodes can be added under three different circumstances; Odometry transformations result in neighbour links, loop closure links are added when the current field of view matches with a previously visited location and the proximity link occurs when detected depths between objects match that of a previous node [21]. A dynamic loop closure detection system is used that intelligently selects only specific locations to detect loops, to limit the time necessary to search through previous locations. This efficient computing makes RTABmap a suitable SLAM program to run on the limited processing power of the microcontroller onboard the AMR and enables real-time mapping and localisation. The memory management assists in enabling this process, by storing the most recent and often seen places in working memory and moving the others to long-term memory. Long-term memory is not used in loop closing, to satisfy real-time constraints. When carrying out loop closures, only the limited number of likely locations are searched through. RTABmap can still gain access to locations from the entire map whenever necessary, but will only move what is stored in long term memory to the working memory when a time threshold for attempting to find matches in the working memory is exceeded [22]. As well as using wheel odometry, RTABmap generates visual odometry from the camera. The method of determining the camera's relative motion by examining a series of camera pictures is known as visual odometry. Similar to wheel odometry, but to a lesser degree, measurements produced using visual odometry are accompanied by inaccuracies that increase over time. One of the main advantages of this type of odometry is that it is not impacted by wheel slippages.

2.5 Autonomous Navigation

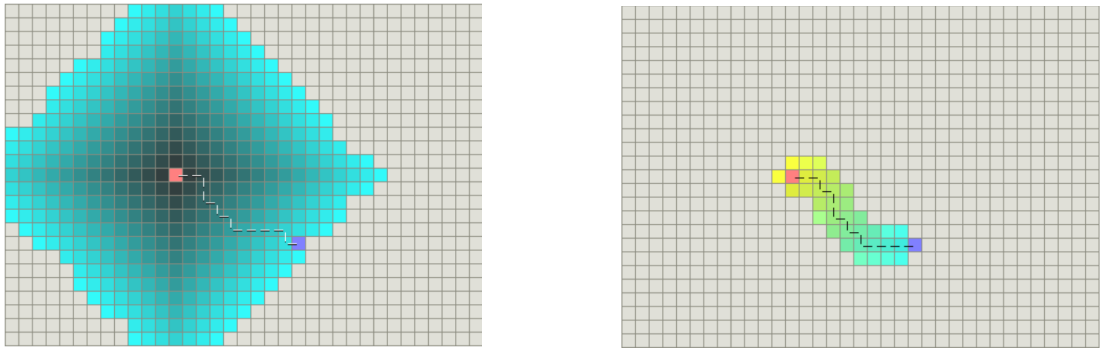
The main aim of autonomous navigation is to plan a collision free route through the map, to reach a certain desired location. This sequence of decisions carried out to reach the goal is based on a prior

knowledge of the environment acquired during the mapping process and sensory feedback of its immediate surroundings. There are primarily two classifications of navigation problems: local and global navigation. The Global planner provides a high-level low-resolution sequence of operations to fulfil the end goal, based on the static map. The local planner concentrates on a low-level high-resolution sequence of activities that are based on real-time sensor data and covers a subset of the global path [23]. The first step of path planning is to convert the continuous representation of space to a discrete map. Then a wide variety of path planning approaches can be used. There are three main methods used for discrete decomposition:

1. Road map: find a set of pathways in the free space
2. Cell decomposition: distinguish between free and occupied cells
3. Potential field: apply a mathematical function to the space

The road map method connects all free space using lines and curves. This free space connectivity will later be used by a shortest-path algorithm to determine the most efficient route. Cell decomposition in essence breaks down the geometric area into a grid. Each cell in the grid is assigned a discrete state of either free, occupied, or unknown. The free cells will be used by the path planning algorithms. For the Potential field method, every point on the map is assigned an artificial potential field by the algorithm. A gradient is created by setting the goal as the lowest gradient and the starting position has the maximum gradient, creating an attractive force within the system, similar to a ball rolling down a hill. The obstacles and boundaries in the map also have an artificial repulsive field that act to keep the robot away. This force is inversely proportional to the distance the robot is from the obstacle [24]. These methods create a practical set of potential routes for travelling through the region. Shortest route algorithms are used in conjunction with discrete decomposition methods to discern the most efficient path for the AMR to travel. A commonly used method of calculating the optimal route is Dijkstra's algorithm. It starts at the source node and analyses the graph to find the least cost (shortest distance or time) between it and all other nodes on the graph.

Expanding outwards, each node will be marked as visited and added to the path. After a node is visited, it checks all its unvisited neighbours and will only mark the adjacent node as visited that gives the lowest cost. At each step, their respective cost from the previous node is quantified and added to the total cost from the source. In this way the algorithm keeps track of the shortest distance between the current node and the source node [25]. When the destination is eventually reached the shortest path between it and the source will be known. While it is guaranteed to find the least cost path, this search is quite slow. Heuristic approaches such as the Greedy Best-First-Search algorithm can optimise the process, whereby the node closest to the destination is chosen instead of the closest node to the starting point. When obstacles are present in the map, this method can forgo a short route in the place of computational speed. The A* algorithm combines the advantages of both techniques in a heuristic function and its graphically explained in figure 3. When A* decides on a pathway to extend to the next node, it does so based on choosing the shortest path to the next node and also using an estimate of the cost of extending the path all the way to the target [26].



(A) Dijkstra's algorithm (B) A* algorithm

Figure 3: A comparison of Dijkstra's and A Algorithm [26]*

For robots interfacing with ROS a robust navigation stack can be used called move_base. As a result, this framework was chosen for the project. It links together a global and local planner to reach a navigation goal, taking inputs from the robots proprioceptive and exteroceptive sensors, a map and

a localisation algorithm. After internal processing velocity commands are outputted to move the AMR to its destination goal, the static map obtained from SLAM is converted from 2D occupancy grids into a global costmap, which represents the difficulty in crossing different areas of the environment. It provides information on where the AMR should and should not navigate, by labelling each cell in the grid as either free, occupied, or unknown. The input from the RGBD camera helps to update the local costmap on its immediate surroundings and newly observed objects or obstacles that are no longer present in the environment. The cost levels can vary between 255 values. A cell containing an object will have the maximum value. An inflation of each obstacle is carried out to propagate outwards from occupied cells, giving a varying cost depending on its distance from the obstacle and whether it is less than the robot's inscribed radius to the obstacle [27]. A route planner is used in conjunction with the costmaps to plan an intelligent and collision free path to the destination. A global planner will form a complete trajectory to the target using the static map, using the discussed methods such as roadmaps, cell decomposition and potential field methods. However, this is not sufficient to avoid obstacles. The local planner takes a nearby portion of global map and relies on information from the camera as well as the map, to avoid obstacles and evaluate the current highest scoring trajectory from what it observes in real-time. Velocity commands are computed by the planners that provide collision free and directional motion of the robot [28].

3. Project Implementation

Developing the AMR involved a challenge both from a hardware and software perspective. While the bulk the of the work was aimed at software development, hardware design and an integration of both components to make a functioning system accounted for a substantial proportion as well. Extensive knowledge was gained through tackling unfamiliar problems and using new technologies. The wide range of tasks involved enabled a varied skillset to be developed, that will be applicable and invaluable in a future engineering career.

3.1 Hardware Design

When deciding on suitable hardware components to design the AMR, size, space and cost constraints had to be considered. As the testing laboratory is limited in area the robot would need to be small and low profile. A flexible system could be developed with the potential of adapting it to larger or entirely differently shaped AMRs, that could fulfil a variety of purposes. For example, the size of the motors could be increased to enable it to transport heavy materials or new camera technology added to enable surveillance, while maintaining its core ability of autonomous navigation. As the aim of the project is to explore the possibility of creating a low-cost robot to support future ubiquity of such technology, inexpensive components were selected.

3.1.1 Robot

Figure 4: The IRobot Roomba 600 shown from overhead (Source: irobot.ie)

The robot chosen for the experiment was the IRobot Roomba 600, which used the IRobot Create 2 open interface. The software interface allows the Roomba's behaviour to be managed and its sensors to be analysed via a sequence of commands, including mode commands, actuator commands, and sensor commands. These can be transmitted and received through the Roomba's serial port, to and from a microcontroller [29]. Though it is a popular automated floor cleaning robot, it has frequently been used in various robotics experiments due to its robust functionality, open-source framework and array of inbuilt sensors [30]. The Roomba is a differential drive robot, consisting of two motor driven wheels. The speeds of both can be varied individually to enable turning. Due to its design for indoor household settings, it can rotate at sharp angles without requiring much space and even almost pivot on the spot. This made it extremely suitable for the relatively confined area that the AMR navigation experiments were ran in. The Roomba is equipped with six infrared sensors on its front panel for object detection, bump detectors to provide information on if the robot is obstructed by an obstacle, wheel encoders to discern the motion of both wheels and infrared cliff sensor to detect stairs and ledges to prevent the robot falling. It also

contains many less useful sensors for this project, pertaining to its intelligent cleaning function. With 0.17 m radius, the Roomba's flat top section can quite comfortably move between obstacles in its testing area and enable a camera, battery, interfacing circuit and microcontroller to be mounted on top. A seven-pin serial connector is also provided to enable the exchange of command and senor data with the microcontroller. The current model of the Roomba used in this project is currently retailed for $229 [31].

3.1.2 Microcontroller

Figure 5: The Raspberry Pi 4 Model B (source: raspberrypi.org)

The Raspberry Pi 4 Model B was selected as the microcontroller to run the AMR's mapping, localisation, and navigation processes. The 4 GB Random-Access Memory (RAM) option was chosen, as it optimises the performance to sufficiently cope with the demanding image processing tasks involved, while simultaneously capable of running and processing data from the AMR's other assignments. In conjunction with the Quad core ARM V8 processer running at 1.5 GHz, the Raspberry Pi provides fast and powerful processing to deal with the multi-faceted autonomous navigation task in real-time. There are four USB ports available for connection with the RGBD camera and forty GPIO pins that can be configured for serial communication. Connecting to the internet was important when downloading packages such as ROS, RTABmap and their many dependencies. The Raspberry Pi provided WiFi connectivity and gigabit ethernet to facilitate this. A micro-SD card slot is provided, where a card can be inserted containing a burned Operating System (OS) image of choice [32]. A 64

GB SD card was used to provide abundant storage space for all the necessary packages and process files created. With its impressive capabilities it is sold at an affordable price of £44 [33]. The Raspberry Pi required a 5 V power supply at 2.5 A.

3.1.3 Battery

Due to the Raspberry Pi's powerful processor and additional connected components, an initial concern was that it might demand too much power if relying on the Roomba's battery and significantly reduce the running time. For this reason, an additional power supply was added solely to power the microcontroller. A Xiaomi Mi Power Bank 3 Pro advertised to be 20 Ah, capable of outputting 5 V and up to 3 A was added to the system [34]. It could provide power for the microcontroller for over 5 hours when fully charged.

3.1.4 Interfacing Circuit

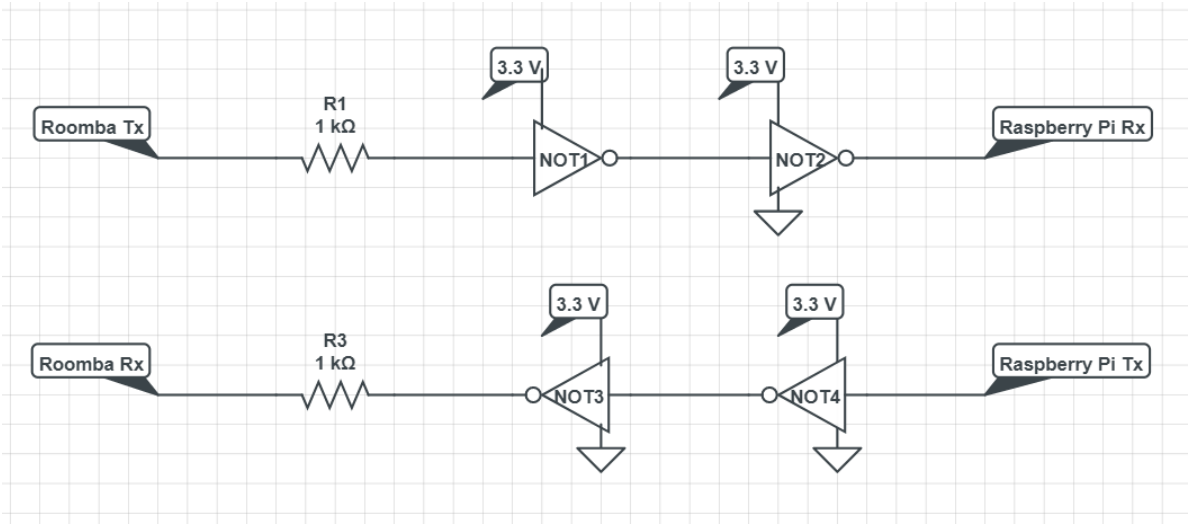


Figure 6: The circuit used to interface between the Roomba and Raspberry Pi serial connections

To facilitate serial communication between the Roomba and the Raspberry Pi, an interfacing circuit had to be designed and built. The Raspberry Pi and Roomba operate off different serial voltage logic levels. If connected directly, the Roomba's 5 V signals could damage the 3.3 V compatible GPIO pins on the microcontroller. This was achieved using the 74HC14, a Schmitt-trigger convertor that acted

as a voltage level shifter in this configuration. The circuit diagram in figure 6 shows how the Tx and Rx pins were connected. The input voltage should not go above the 3.3 V Vcc that the circuit is rated at, unless the current limit is observed. To stay below this maximum value, a 1 kΩ resistor is connected between the Roomba's Tx and Rx pins as a current limiting protection. It prevented the Roomba's maximum 5 V output ever generating enough current to exceed the 20 mA safety limit, which could destroy the 74HC14 device. The circuit will only ever output a maximum of 3.3 V to both Rx pins. This was enough to prevent the Raspberry Pi's GPIO pins from being damaged and was also sufficient to register as a logic high in the Roomba.

3.1.5 Camera

Figure 7: The Asus Xtion Pro Live RGBD camera (source: asus.com)

The AMR's robust vision system was enabled by the dynamic capabilities of the Asus Xtion Pro Live. A stereo vision camera would also have provided very similar functionality, but this camera was already available from a previous project. The device is equipped with adaptive depth detection, infrared sensors, and colour sensing. Depth detection gave the AMR the ability to discern its distance from objects, which was key for mapping and navigation. It calculates this information by projecting infrared light in a known projected pattern and analysing what is reflected. From the disparity of the emitted and detected light pattern, the distance between the object in each pixel and the camera can be measured [35]. Its high-quality colour sensing was an important feature when carrying out loop closures and creating visual words in SLAM, as it provided an important distinction and texture

between features in the environment. This enabled signage and objects to be recognised. The OpenNI2 libraries make it compatible to run on the Linux based operating system used by the Raspberry Pi. The camera costs $300 and was the most expensive component in the AMR but dramatically cut the cost and simplicity of the system through removing the need for LiDAR [36].

3.1.6 Wheel Odometry

The Roomba provides an extremely useful set of wheel encoders. These can be used to discern the odometry of the robot, which is key information when mapping the environment, localising the robot, and navigating. The Roomba's encoders provide square wave signals, containing information about the distance and angle traversed by the AMR. Using an IR beam, a count is incremented during each revolution of the wheel to calculate the distance travelled. When driving forward the count will increase and when driving back it will decrease. The system uses the commanded direction to understand which way the robot is travelling and the knowledge that the Roomba's wheels are 258 mm apart. The formula used is shown below:

$$Distance = \frac{1}{2} * \frac{(Right\ Wheel\ Distance)+(Left\ Wheel\ Distance)*\pi}{Encoder\ Ticks\ per\ roation} * 258$$

The angle is calculated using the difference in distance travelled by the right the left and right wheel since the angle was last requested as follows:

$$Angle\ (radians) = \frac{(Right\ Wheel\ Distance) - (Left\ Wheel\ Distance)}{Encoder\ Ticks\ per\ roation} * 258 * \pi$$

Consequentially, clockwise angles are negative and counter clockwise angles are positive, with the information provided in radians.

3.2 Software Design

The majority of the time spent on the project comprised of software development for the system. Several substantial problems had to be solved; These include the initial set-up of the Operating system and installation of various packages, mapping, localisation, and navigation.

3.2.1 Operating System (OS)

To interact with the Raspberry Pi efficiently, an OS had to be added to manage the microcontroller's hardware, and software resources. It was important that the OS supported the packages that would be used in this assignment, such as ROS and RTABmap. Initially the pre-installed Raspbian Buster OS was chosen. However, after a prolonged period of the project was spent trying to unsuccessfully install packages on this OS, it was realised that this had been a poor choice. Though it is stated by the developers that they were compatible with Raspbian, many issues and dependencies had not been resolved, as there is a relatively small community of users. ROS and most SLAM packages were primarily launched on Ubuntu and its strong community regularly publishes updates and quickly fixes reported bugs. Instead, the recently released Ubuntu Mate 20.04 edition was burned onto the SD card, allowing the necessary packages to be installed seamlessly in most cases. This limited the number of errors that had to be solved. The initial set up requires a mouse, screen, and keyboard to be connected to set up SSH, log-in credentials and Wi-Fi connection, so the interface could be connected to over a computer, enabling future mobile use. Initially the SSH allowed the computer and Raspberry Pi to communicate over a common network through a terminal. This required logging into the Raspberry Pi, using the created credentials through the command prompt window to initiate connection. When XRDP was installed on the microcontroller, this facilitated Windows Remote Desktop connection, providing access to the Ubuntu user interface. The processes for installing some important packages on the OS are detailed in the appendix. The UART port on the Raspberry Pi had to be enabled by configuring Ubuntu's firmware using the following steps:

$ nano /boot/firmware/config.txt

In this file the lines "*enable_uart = 1*" and "*dtoverlay = uart4*" were added. This configuration sets up the GPIO pins, so pins 8 and 9 were Tx and Rx respectively. Ubuntu assigned the serial port to the address */dev/ttyAMA1*. This was used to communicate with the Roomba.

3.2.2 ROS driver for Roomba

As ROS was chosen as the platform on which the AMR's SLAM and navigation would be built, it was necessary to choose a package that would facilitate communication with the Roomba. The create_robot package by AutonomyLab served this purpose. It published the sensor data as a ROS topic and allowed motion commands to be sent over the 'cmd_vel' topic. The odometry information from the wheel encoder was published as 'odom', giving information on the Roomba's distance in metres for the x and y direction and yaw in radians. The teleop_twist_keyboard package could use the 'cmd_vel' topic to control the Roomba with a keyboard, which was useful when driving the robot around the environment manually during the mapping process. To run the create_robot package, the following process was used. As the library is present in a catkin workspace the setup.bash file had to be sourced in the create_robot folder:

$ source ~/create_ws/devel/setup.bash

By default, Ubunutu disables the permissions for the serial ports each time the machine is switched on. Read and write permissions must be allowed to enable communications with the Roomba through the */dev/ttyAMA1 port*:

$ sudo chmod 666 /dev/ttyAMA1

The library is then launched and communication with the Roomba is enabled:

$ roslaunch create_bringup create_2.launch

3.2.3 Visual Odometry

Visual odometry is the process of estimating the position and orientation of the robot in real time, based off a stream of images. Though this technology has been researched since the 1980s, in recent times NASA spurred on its development for use in the Mars Rover. It was beneficial to their mission, as the surface it would be traversing on the planet is very rough and uneven. Consequentially, wheel odometry would not have been a reliable option. Visual odometry was not affected by these

conditions and delivered an effective alternative. RTABmap provided an in-built capability to perform this function, using the image stream from the RGBD camera. The system relies on tracking features. Identical feature points in sperate frames are matched and linked into image trajectories for the given rate of the video. The transformations between several different frames are computed to estimate the motion of the camera. The current pose is then updated [37].

3.2.4 Mapping

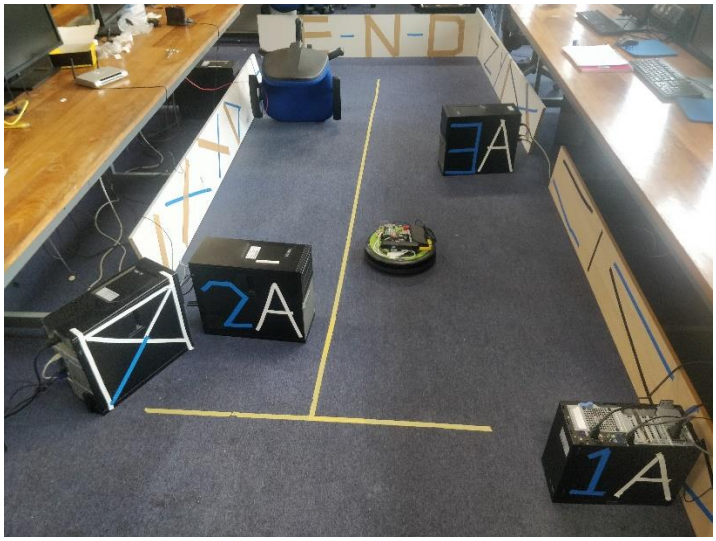


Figure 8: The constructed environment for AMR mapping and navigation

Mapping was carried out on a specially constructed obstacle course in the project's laboratory, seen in figure 8. Three static obstacles labelled 1-3 and a chair were used to add features. The environment was designed with distinctive signage on objects and patterns on the wall to assist with loop closing, by creating a robust visual bag of words. The process also uses depth imagery to produce a 2D occupancy grid map of the obstacle course. The static mapping process takes point cloud inputs. These are data points in space that represent the 3D position of features in the camera's line of sight, relative to its current location. The camera's whereabouts is calculated in relation to its base position, using odometry data. OpenNI2 is launched at the beginning of the process to publish point cloud data, using the Asus Xtion Pro Live's depth images. The ground segment is filtered out and the point cloud is down sampled by a voxel grid filter to make it easier to

process. This filter is a 3D mesh that separates the point cloud into cubic cells and labels them as occupied, unknown, or empty. Others are removed, thereby reducing the size of the point cloud, which delivers information that is far too dense to compute in its entirety. The point cloud is divided into planes, and the angle and distance from the camera for each plane is calculated. The height of the camera above the ground is published in ROS and its position is constantly calculated and updated. The relative angle, height and distance of each point cloud is processed and found. A 3D occupancy map is created, allowing objects to be dimensionally perceived. This map is projected on the floor plane as a 2D grid map that will be used to assist with localising and navigating the AMR. It consists of grids marked as occupied, free and unknown. For localisation and navigation, the generated map is used in conjunction with the database, containing images of previously recorded locations. This assists in identifying where within the 2D map the AMR is located using loop closing.

The mapping process was performed in a single session, until a satisfactory representation of the environment was developed. The strength of RTABmap is that if changes were made to the environment or it needed to be expanded, an existing map could be updated by running an old database and exposing the AMR to areas that need to be added or amended. This ability makes it very practical for industrial settings that change frequently. To begin, RTABmap was launched from the terminal using the rtabmap.launch file. Various configurations could be used, which mainly depended on what type of odometry was being deployed and their respective transform frames. These are listed below:

$ rosrun tf static_transform_publisher 0.1 0 0.08 0 0 0 base_footprint camera_link 100

Firstly, the relationship between two different coordinate frames is established. The fixed relative distance between the centre of the robot and the RGBD camera must be asserted. The camera was placed at the front AMR, positioned 0.1 m from the centre of the robot in the x-direction and 0.12 m above in the z-direction. It is important that the SLAM algorithm is aware of this relative distance, as

the location of point clouds in the map are found in relation to the camera. Using the transform, the position of point clouds can be found from the centre of mass of the AMR.

Wheel Odomtery only- *$ roslaunch rtabmap_ros rtabmap.launch rtabmap_args:="--delete_db_on_start" frame_id:=base_footprint visual_odometry:=false odom_topic:=/odom*

The rtabmap.launch file located in the rtabmap_ros folder is initiated in terminal and some parameters are altered. The default database is deleted at start-up to begin a new mapping process. The base_footprint is the base_frame ID of the robot, that publishes transform information on the 2D position and orientation of the AMR's centre point. The last two parameters tell RTABmap that the default visual odometry will not be used and to instead use wheel odometry, which is published by create_robot as /odom.

Visual Odometry only- *$ roslaunch rtabmap_ros rtabmap.launch launch rtabmap_args:="--delete_db_on_start" frame_id:=base_link*

The only alteration from the launch file is that the base_frame is set to its new ID called base_link because the odometry information is no longer arriving from the Roomba. This parameter must also be changed in the static transform publisher.

The Roomba can be driven around the environment at slow speeds by launching the teleop_twist_keyboard package. This gives RTABmap time to process as many images as possible of each section of the environment and process loop closure based on previously viewed locations. When the end of the obstacle course is reached, the robot turns around and returns to the starting point, ensuring it is completely familiar with every part of the environment from many orientations and positions. After the AMR has received sufficient exposure to its environment Rviz is opened. This is a ROS graphical interface that enables visualisation of many different types of topics. It is a convenient means by which to view the created map. The 'rtabmap/grid_map' is selected, showing the environment as a coloured 2D map, with black areas representing object, white as free space

and grey as unknown. If the map is satisfactory, the current RTABmap database must be saved as another file to prevent the information in the default database being overwritten in a fresh mapping process. The grid map must also be saved in a .pgm and .yaml format to be used for the navigation stack. This can be carried out using the following terminal command:

```
$ rosrun map_server map_saver map:=rtabmap/grid_map
```

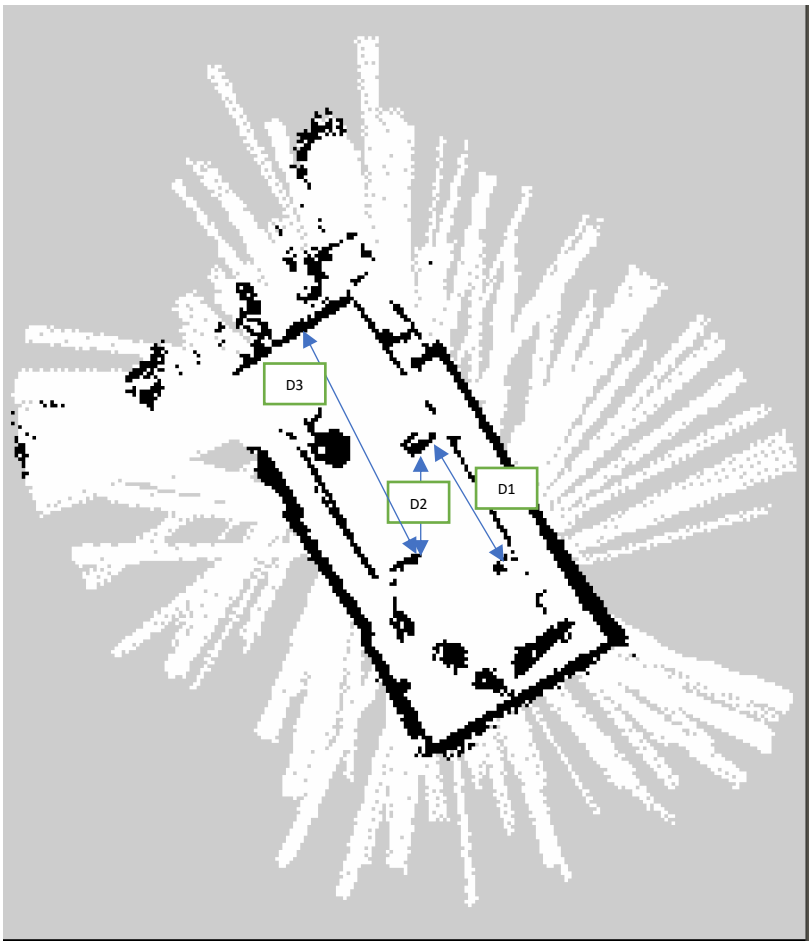


Figure 9: The map obtained using wheel odometry

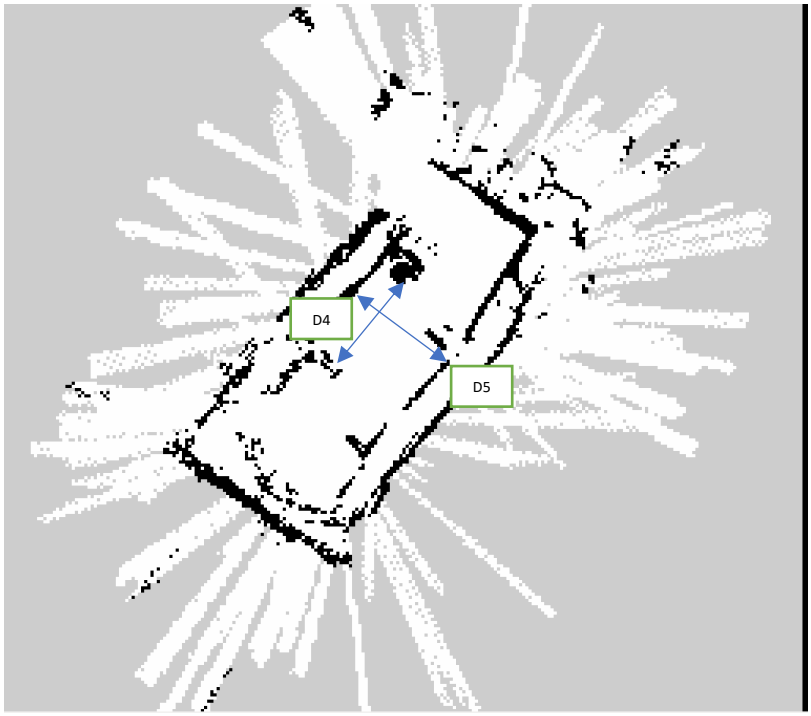


Figure 10: The map obtained using visual odometry

The maps obtained from wheel and visual odometry are shown in figures 9 and 10. A series of measurements are carried out to test the accuracy of both maps. The distances between objects in the map were measured physically and using the measurement tool in Rviz. The results of this experiment are shown in table 1.

Table 1: Distance measurements from both maps and corresponding physical measurements to calculate map accuracy

	D1 (cm)	D2 (cm)	D3 (cm)	D4 (cm)	D5 (cm)	Average % error
Physical Measurement	201	148	350	154	185	
Wheel Odometry	206	150	354	156	182	1.6
Visual Odometry	195	141	345	161	177	3.6

Evidently wheel odometry provides optimal accuracy at 1.6 % error on average versus 3.6 % when using visual odometry. The obtained result was not assumed beforehand, as wheel odometry is prone to wheel slippage that creates position drift overtime. As a result, it was unknown how the Roomba's wheels would perform on the carpeted floor of the projects' laboratory. The course laid out for the AMR is quite small but in a larger more realistic testing environment visual odometry could prove to have a superior performance. This is because the wheel odometry errors increase proportionally with the total distance travelled but visual odometry is not affected by this inaccuracy. The drawback of visual odometry is that it adds extra computational cost and is affected by lighting conditions such as sunlight or shadows, which can lead to vehicle positioning errors [38]. The accuracy of the map is key for precise localisation to keep track of its position over time. In different settings, that have a floor more prone to slipping or if the AMR was expanded to a larger environment, visual odometry could be a more viable option.

To exploit the benefits of both odometries, while reducing the effect of their limitations, the data published from the visual and wheel topics were fused together. This was carried out using the robot_localization package on ROS. It uses an Extended Kalman Filter to fuse data from two or more sensors to perform accurate state estimates. This is a non-linear version of the standard filter that makes a linearised approximation of the system and is often used in navigation. The odom_fusion.launch file carried out this operation, the output of which was the topic "/odom/filtered". This was formed by a combination of the visual and wheel data that should provide odometry with improved accuracy. The file launches the robot_localization package and sets the visual odometry as odom0 and wheel odometry as odom1. In the first two matrices there are 15 possible states that can be integrated, by setting them to true. For this application, only five states are used, corresponding to what data is available from the odometrys; distance (x and y), linear velocity (x and y) and yaw. For the covariance matrix the default values remained unchanged, as they worked reasonably well. If time constraints were not an issue these could be tuned in a relatively complicated process to improve the output. Covariance is an expression of the correlation

between two variables and their tendency to vary together. The mapping process using fused odometry was launched as follows:

$ roslaunch rtabmap_ros rtabmap.launch rtabmap_args:="--delete_db_on_start" frame_id:=base_link odom_topic:=odometry/filtered cfg:="~/rtabmap.ini"

$ roslaunch robot_localization odometry_fusion.launch

The map shown in figure 11 was obtained:

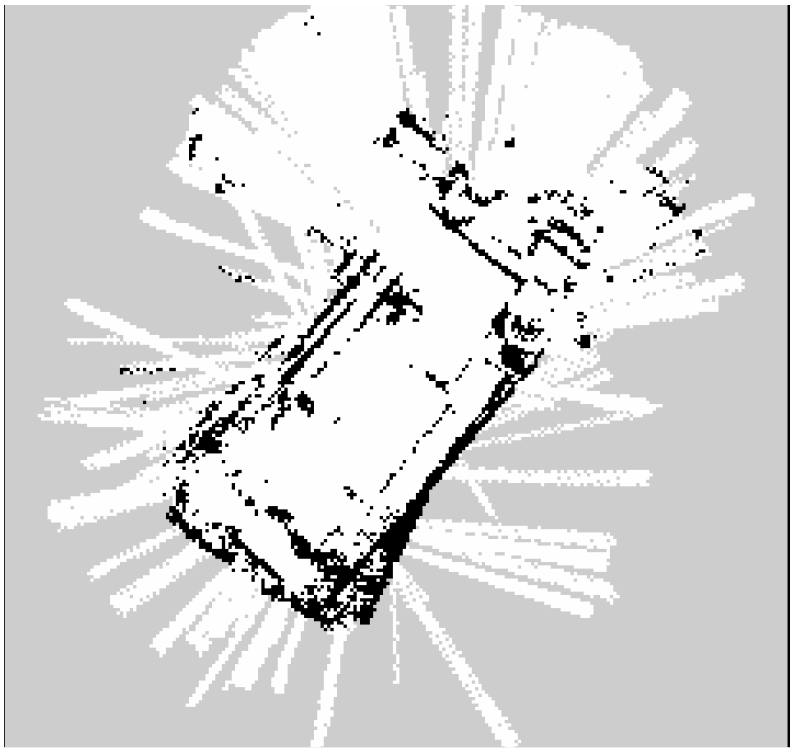


Figure 11: The map obtained using fused odometry

As before, the same 5 measurements were carried out between objects on the map to establish how accurate the mapping process was when using fused odometry.

Table 2: Distance measurements from fused odometry map and corresponding physical measurements to calculate map accuracy

	D1 (cm)	D2 (cm)	D3 (cm)	D4 (cm)	D5 (cm)	Average % error
Physical Measurement	201	148	350	154	185	
Fused Odometry	203	146	353	156	183	1.2

The results in table 2 demonstrate that fusing odometry does improve the accuracy marginally at 1.2 % versus 1.6 % when using wheel odometry. It is expected that if mapping was carried out over a larger area this disparity would increase drastically in favour of fused odometry. While wheel odometry would suffer from a positional drift overtime, this would be mitigated in the fused data by visual odometry.

3.2.5 Localisation

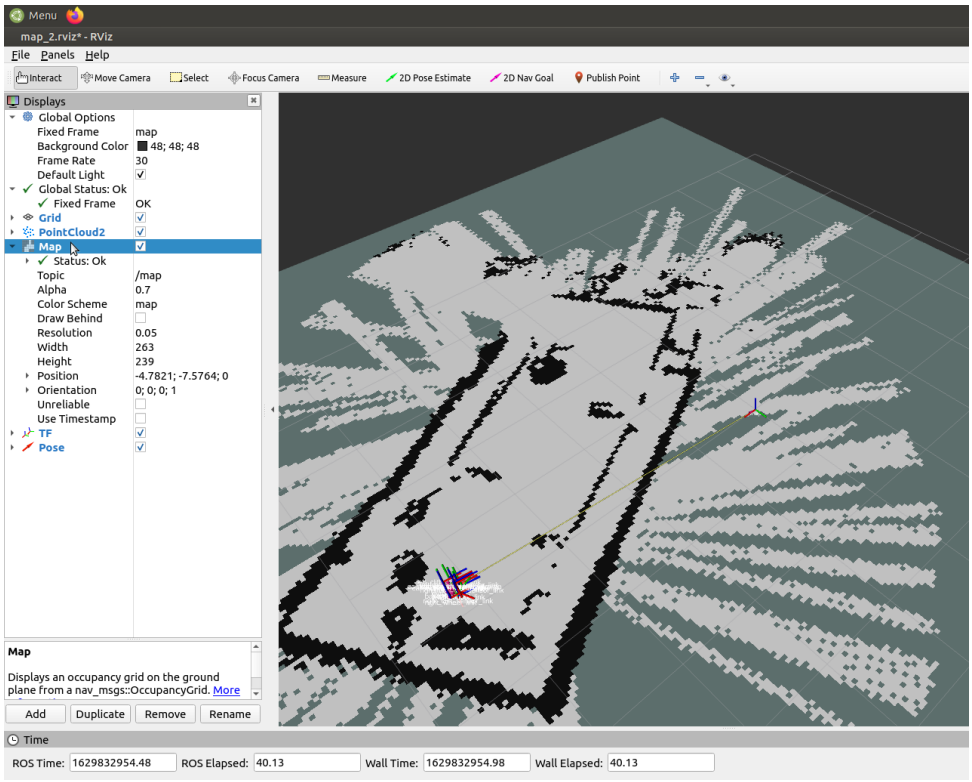


Figure 12: 2D grid map on Rviz with RTABmap in localisation mode, the robot's pose is shown

This localisation process can begin after a satisfactory map has been created. Employing odometry and loop closures that use the visual bag-of-words and proximity detection, the SLAM algorithm can constantly correct the position of the AMR as it moves around the 2D grid map. Tests must be carried out to ensure that the AMR is localising effectively within the environment. The running process is similar to mapping but with the following changes:

$ roslaunch rtabmap_ros rtabmap.launch localization:=true frame_id:=base_footprint visual_odometry:=false odom_topic:=/odom database_path:=/old_database_path

RTABmap is set to localisation mode and the database that was used for the relevant instance of mapping is addressed. By opening Rviz and subscribing to the map topic the 2D grid map should now be visible. The tf topic is also added and shows the current position and orientation of each frame in the co-ordinate system, e.g base_link, camera_link and odometry frames. This is displayed in figure

12 and provides a valuable insight into the overall pose of the robot. A few tests are carried out to ensure that the robot is present in a position that corresponds to where it is located on the map. This can be achieved by measuring the distance moved by the robot in the environment and comparing it to scaled measurements taken from Rviz. When The AMR is localising in all regions of the surroundings with satisfactory accuracy, the navigation problem can be addressed. The system had a robust ability to find its location anywhere within the confines of the environment when switched on or moved.

3.2.6 Navigation

The navigation challenge built upon what had been developed in the mapping and localisation stages of the project. The AMR needed a reliable and accurate map that it could use to set navigation goals and plan a collision free path to reach its destination. It was essential that it could robustly localise, to compute its point of origin and when transiting that it was constantly aware of its location, in order to regularly course correct and supress any navigation errors. This problem was addressed by developing a ROS navigation stack, using the move_base package. It combines a global and local planner to complete its global navigation assignment. This is achieved by intelligently integrating and processing multiple nodes, such as the map, RTABmap localisation, the RGBD senor information, odometry and a navigation goal. The output of this process are velocity commands to the Roomba that guide it to the destination.

To achieve the goal of navigating the Roomba autonomously, there was a lengthy design process and consideration of various other methods. It was not immediately clear that a ROS navigation stack would be the chosen solution. At the beginning, attempts were made to write python scripts that used the Roomba's IR and bump sensors to detect objects and navigate the course without the camera. Though these assignments were ultimately not used in the final project, an understanding was gained on how to control the Roomba and read messages from its sensors at a fundamental level. As the robot was an older piece of equipment from previous projects, it had a slight

malfunction where it veered to the right whenever it was given a drive straight commands, at approximately 0.1m per 1m driven. Methods were explored to compensate for this drift, which could become substantial over long distances. It was expected that the camera would be the most effective method at rectifying this defect. Though extensive experience and understanding were gained about the robot's mechanics and sensors, these python scripts were not used in the final implementation. The mapping and localisation systems were developed in ROS, so there would be a great deal of compatibility issues to overcome when they needed to be integrated outside of the environment. In addition, ROS offered all the functionality that could be achieved when communicating with the Roomba at a lower level through Python scripts, using the create_robot libraries. The next step was gaining experience at using these libraries. This involved sending velocity terminal commands to the Roomba over ROS using 'cmd_vel' to drive it around the room, while publishing and interpreting sensory information from the Roomba. This process formed an early development stage in building the navigation stack.

Costmaps

The global and local costmaps formed integral components of the navigation stack and both characterised the environment differently and at separate scales. The parameter tuning for these maps is essential for the success of the path planner. They shared some common parameters in the costmap_common_params.yaml file. Due to the AMR's rounded shape, its radius is stated. This is used to calculate the radius of an inscribed and circumscribed circle, which are utilised to inflate obstacles in a way that is appropriate for this robot. The transform tolerance parameter sets the maximum amount of latency that is acceptable to the system. This had to be increased after the processor sustained damage. The transform frame ID of the mobile base is also set.

The global map is generated from the static map, and it is used to produce the initial path from the base to the destination. It inflates obstacles on the map as seen in figure 13. The transform frame of the map and robot are asserted. The obstacle inflation radius was set as 0.4 m. This projects objects

outwards at enough distance to avoid collisions, while also not invading too much of the limited free space in the obstacle course. Two plugins are added to the map, which combine to layer the map. The static map layer provides the 2D occupancy grid from the mapping process. The inflation layer expands the area of obstacles to deter the path planning algorithm from approaching too closely. The update rate was kept low at 2 Hz due to the limitations of the processor.

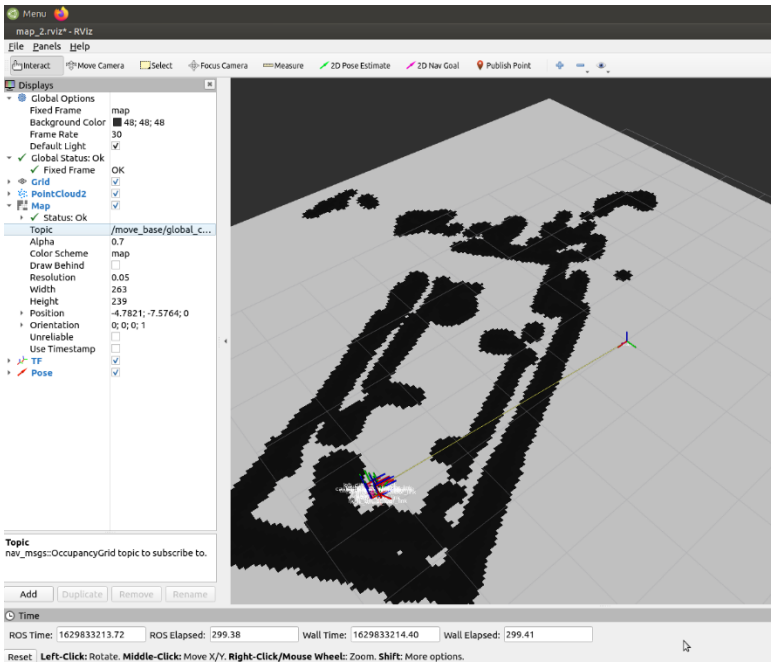


Figure 13: Global costmap displayed on RVIZ

The local costmap represents the environment immediately around the AMR, using the static map, obstacles, and inflation layer. Creating a map that incorporates the live RGBD camera stream, with the nearby section of the static map, imaging its current field of vision and inflating any visible obstacles. Figure 14 shows this representation, with obstacles not present during mapping visible in the costmap. The maximum obstacle height is set to a height 0.25 m, which provides sufficient clearance above the AMR. This is essentially the height of the voxel grid, and it will discard point clouds from regions of objects above this height, as they are not going to affect the robot's trajectory. This filtering of unnecessary data points also improves real time processing speeds. A minimum height of 0.02 m prevents the system mistaking inconsistencies on the ground as objects.

The obstacle range parameter is set to 2.5 m, which means that the map will only update obstacles that are within this distance. The rolling window parameter is set to true to centre the map around the robot in a 4 m X 4 m size, which is updated at 2 Hz. A smaller inflation radius of 0.15 m is used to prevent the limited space in the map being taken over. While the inflation radius dictates where the zero-cost point is from the radius, the cost_scaling_factor is inversely proportional to the cost of the cell and will cause the cost curve to decay quicker, the larger the value shown in the following equation:

*exp(1.0 * cost_scaling_factor * (distance_from_obstacle - inscribed_radius)) * (costmap_2d::INSCRIBED_INFLATED_OBSTACLE - 1)*

The optimal costmap decay curve has a relatively low slope, such that the optimal path is as far away from the obstructions on each side as possible. The benefit is that the robot prefers to move in the middle of the obstacles.[39] A cost_scaling_factor of 10 achieved this goal.

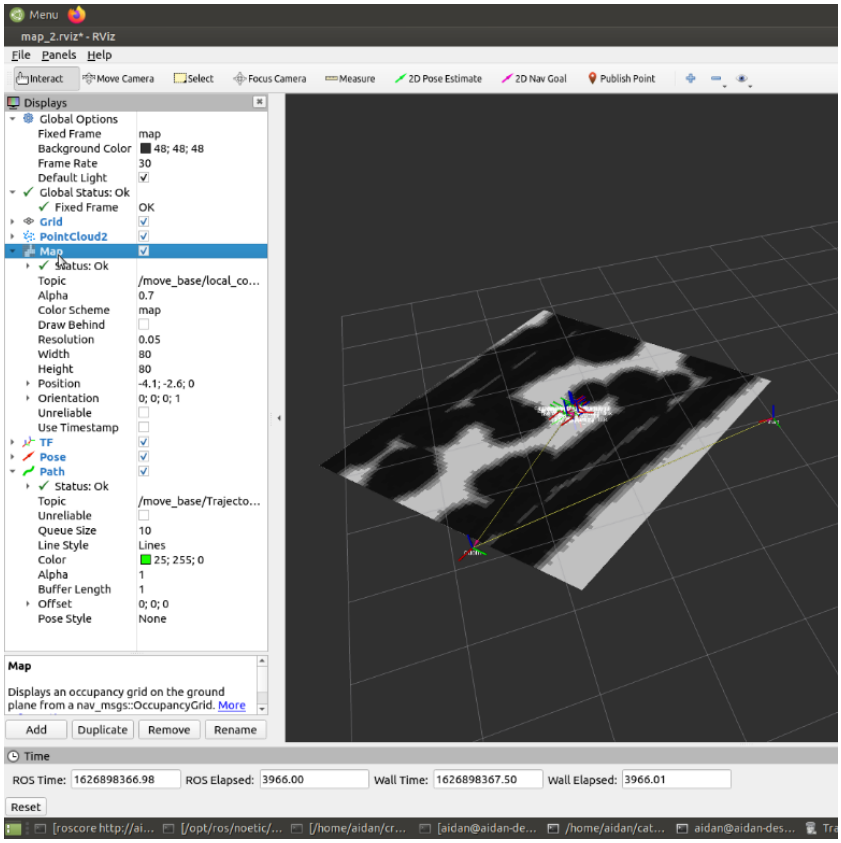


Figure 14: Local costmap displayed on Rviz

Global Planner

The Navfn package was integrated in the stack and tuned to facilitate global planning. The system relies on Dijkstra's algorithm to identify a global path with the lowest cost between a start point and an end point. It is designed to be used with a circular robot, making it suitable for this application. It is a fast navigation function to create the initial plan for the AMR based off the static map.

Local Planner

The base_local_planner was the package chosen to plan the AMR's local route through the environment. The planner utilises the local costmap. Using this small-scale representation of 2D space, it constructs a kinematic trajectory for the robot, to travel from a starting point to a nearby destination. Along the journey, the planner generates a value function, which is represented as a grid map locally around the robot. This value function represents the expense of traversing the grid

cells and facilitates obstacle avoidance. This is a form of trajectory rollout algorithm that provides control for a mobile base in 2D space. It starts by discretizing the AMR's control space and simulating a series of forward velocities from the current position to project where the robot would end up. These trajectories are evaluated using a grid created from the local costmap, in terms of how close it takes the AMR to obstacles, the destination, and the global path. Paths that result in collisions are rejected before choosing the highest scoring route [40]. The base_local_planner_params.yaml is configured with a set of maximum and minimum motion parameters that were obtained through testing the robot to discern what was suitable for the restricted space in the environment. As the Roomba is differential drive, the holmonic_robot parameter is set to false.

Move_base

The move_base package links together the global and local planners, as well as the various configuration files that have been discussed. It also integrates the static map and launches RTABmap in localisation mode with the relevant mapping database for loop closures. RTABmap inherently provides the sensory data from the RGBD sensor. The odometry and base frame id are also set within the move_base.launch file. Move_base is responsible for instructing the creation of the global and local path plans and costmaps, serving as an orchestrator of the entire process, as shown in figure 15.

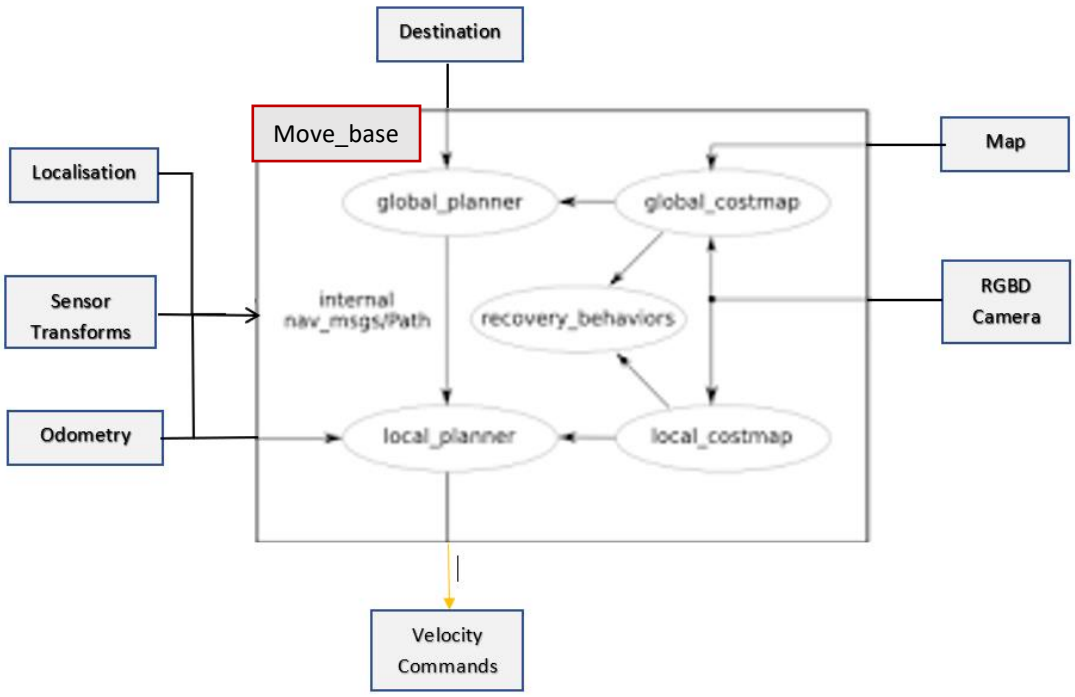


Figure 15: The structure of the AMR's navigation stack

A recovery behaviour plugin is integrated with move_base. When no valid velocity command is generated by the base_local planner for some time, implying that it is probably stuck, the recovery behaviour is activated. The local costmap is first cleared out to revert to the static map. The robot then executes a rotate recovery to check its surroundings again with a clear map and then see if it can find a valid path. It repeats this twice and will abort navigation if none is found after this time, as it considers the goal infeasible. The AMR will often reverse away from objects during this time to obtain a less obstructed field of vision. This routine is a very basic method of freeing the robot.

The entire navigation stack is launched using the move_base.launch file. It was configured with all the necessary components as shown in figure 15. The ROS graph in figure 16 displays all the ROS nodes and topics in this particular navigation system and their inter-communication. This overview was a useful tool at debugging the system as it facilitated node connections to be checked when an issue arose. All move_base's inputs and its outputted velocity commands to 'cmd_vel' topic are

shown, corresponding to what is seen in figure 16. The '*/tf* 'node links several sections and maintains the relationship between their coordinate frames overtime. The graph was found using the following terminal command while the navigation stack was running:

$ rosrun rqt_graph rqt_graph

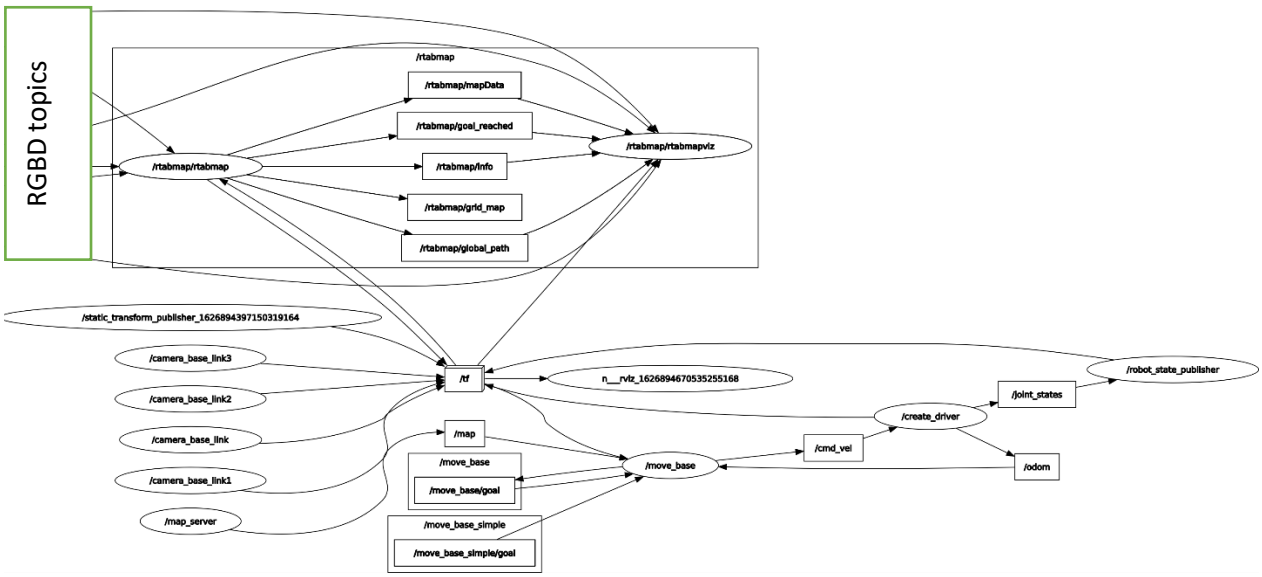


Figure 16: RQT graph showing the connecting nodes in the navigation system

4. Results

A variety of trials were carried out in the testing environment to establish:

1. The accuracy of delivery of the AMR when a navigation goal was set.
2. How this accuracy was affected by obstacles placed in the global path that would cause the AMR to reroute quite significantly.
3. How the AMR coped with obstacle avoidance situations of varying complexity.
4. The number of times the AMR could travel collision free through a relatively onerous environment, with two consecutive obstacles obstructing its global path.
5. If the system can negotiate its way out of a cul-de-sac.

In the final week of completing the project's practical work, the processor was partly damaged. This caused the performance of the Raspberry Pi to degrade rapidly after around half an hour of run time, after which it heated up and became overloaded. Figure 17 shows the CPU usage after the navigation stack had been running for some time. As a result, the following results were obtained before this deterioration had occurred during each power up. After the performance degenerated, the device had to be switched off and left cool down before a new round of testing could commence.

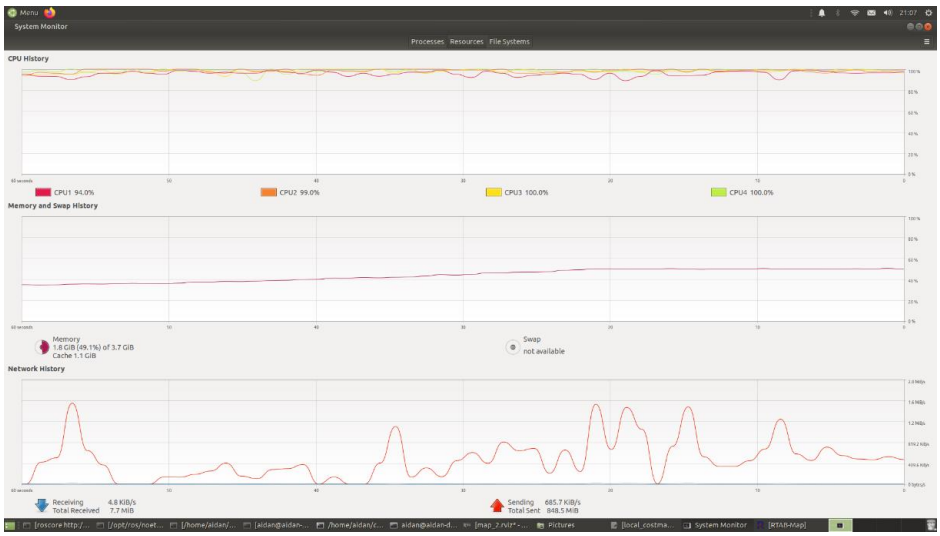


Figure 17: The CPU overload during navigation after the processor was damaged

4.1 Robot Delivery in Static environment

Going to a destination goal with a high degree of accuracy is a critical ability expected from AMRs in most applications. The robot must be as close as possible to the target and arrive there in a reasonable amount of time without colliding. To test the AMR's ability to execute this task several target destinations were set on the map, these are shown in image 18. When these locations on the map were chosen, vertical and horizontal distances were found from known points on nearby objects and/or the wall, exemplified by point 1. The distances were first measured on Rviz using the measuring tool. In this test no obstacles were added to the map that were not present in the static map. While some destinations were set to parts of the map that were outside the AMR's initial field of view, this test had a lower complexity as its global path would not be obstructed. The Euclidean distance was found between the physical point the Roomba arrived at and its set destination on Rviz. This accounted for the distance error:

$$Distance\ Error = \sqrt{(Difference\ in\ vertical\ distance)^2 + (Difference\ in\ horzontal\ distance)^2}$$

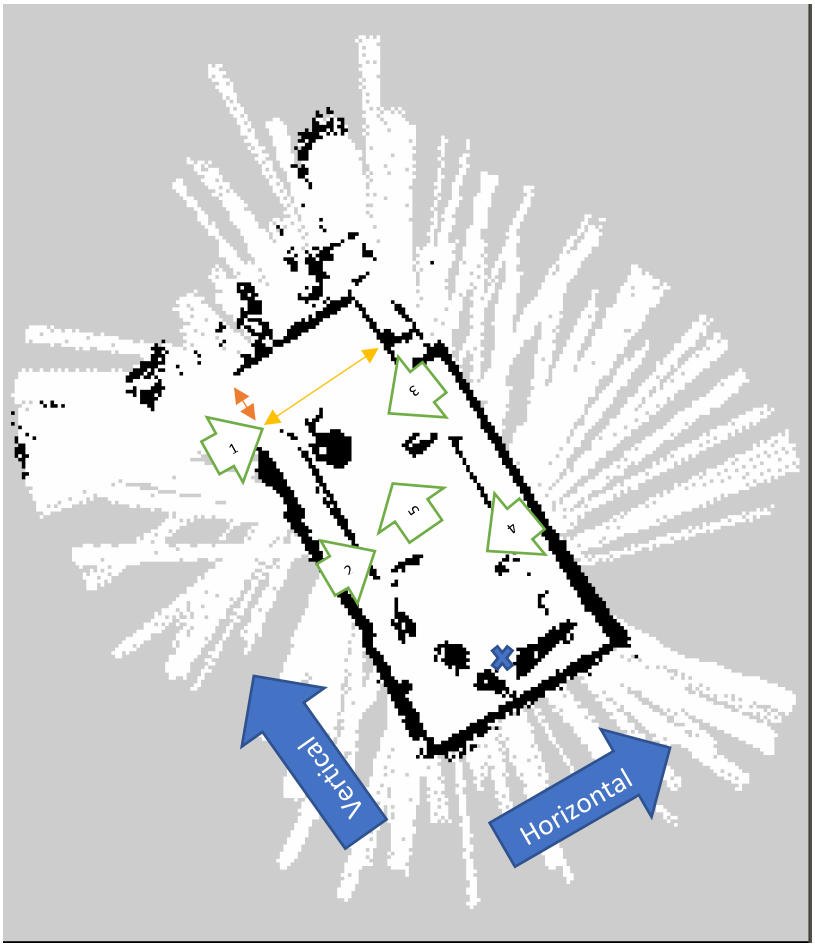


Figure 18: The locations of the navigation goals set during testing

A navigation goal is set to these precise locations. The AMR travels to the destination, always beginning at the point marked as 'X'. When the robot stops, the corresponding physical measurements are taken to calculate the delivery error. The results are shown in table 3:

Table 3: The measurements taken when the AMR travelled to the navigation goal to assess its performance

	Map		**Physical**				
Destination	Horizontal (m)	Vertical (m)	Horizontal (m)	Vertical (m)	Error (mm)	Collision	Time (s)
1	1.92	0.68	1.93	0.64	41	No	27

2	0.25	0.83	0.28	0.82	32	No	12
3	0.47	0.47	0.51	0.47	40	No	23
4	0.58	0.55	0.54	0.57	45	No	9
5	2.13	0.51	2.20	0.52	70	No	16

The results demonstrate that the AMR is very competent at precisely travelling to a point in the static environment and within an acceptable timeframe. The system is robust and reliable in these conditions and the local path planner corrects the inherent steering veer present in the Roomba. It accurately follows the global path and the base_local planner intelligently slows down and makes small precise movements when the AMR is near the endpoint. The mean error in distance is 45.6 mm, which is not substantial when considering the radius of the Roomba is 170 mm. The chosen inflation radius ensures that the robot's path does not cut too closely to objects and cause collisions. In all cases the system was able to guide the Roomba back to the starting point before commencing the next round of testing.

4.2 Robot Delivery in Dynamic environment

In a realistic operational environment, the AMR will encounter non-permanent objects that were not present during the mapping process. The delivery goals detailed in the previous tests were again used but this time an obstacle of size 0.35 m X 0.12 m was placed length ways across the global path, that would result in an altering of its course from the initial plan. The same parameters were examined to discern whether the AMR had carried out a successful delivery. This experiment examined the AMR's obstacle avoidance, ability to re-route and cope with a dynamic environment.

Table 4: The measurements taken when the AMR travelled to the navigation goal to assess its performance with obstacle avoidance

	Map		**Physical**				
Destination	Horizontal (m)	Vertical (m)	Horizontal (m)	Vertical (m)	Error (mm)	Collision	Time (s)
1	1.92	0.68	2.01	0.59	142	No	39
2	0.25	0.83	0.30	0.88	71	No	18
3	0.47	0.47	0.55	0.51	85	No	33
4	0.58	0.55	0.51	0.59	80	No	15
5	2.13	0.51	2.22	0.54	95	No	21

The obtained results in table 4 demonstrate that the AMR is capable of obstacle avoidance and can reroute effectively around the obstacle. The navigation goal is missed by a higher margin in all cases, with the mean error increasing to 94.6 mm, implying the accuracy is adversely affected during this operation. As the navigation stack used wheel odometry, the extra distances traversed and forced turning when avoiding the obstacle may have caused slightly more wheel slippages and increased the accumulated error. This is the probable cause of the reduction in accuracy. To overcome this inaccuracy, the developed fused odometry system should have been integrated with the navigation stack. This improvement would have been added if time had allowed.

4.3 Obstacle Avoidance Scenarios

Three obstacle avoidance situations were created in the environment to demonstrate how the AMR planned its alternative route when confronted with an obstruction. These were designed to establish how capable the system was at adapting to new navigation challenges.

Scenario 1

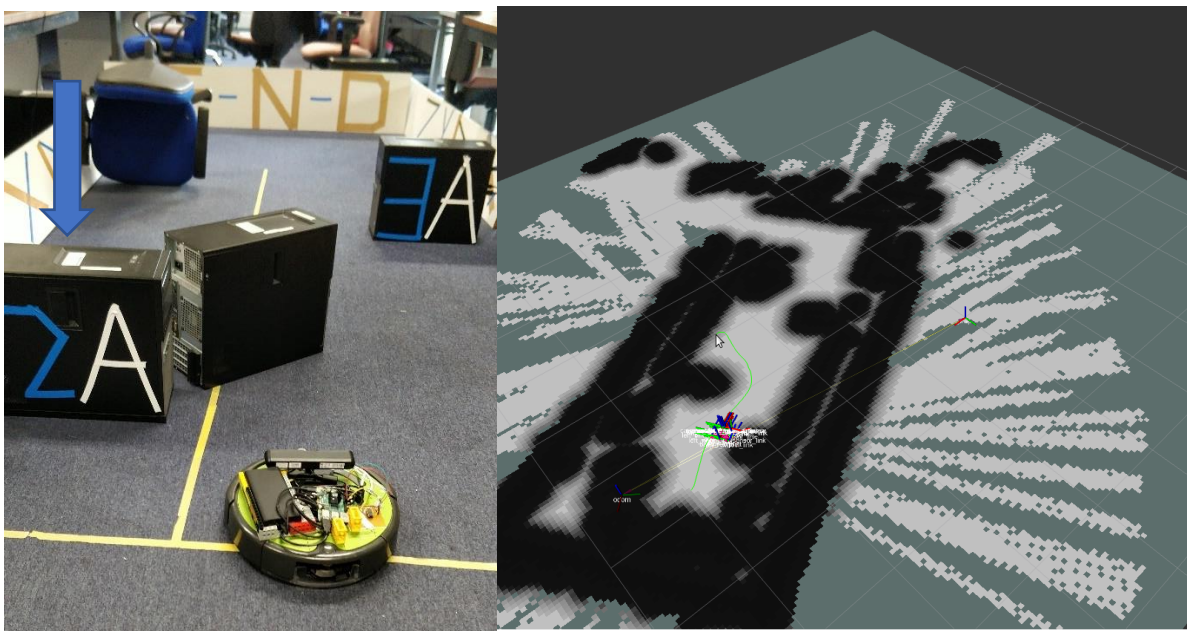


Figure 19: An obstacle placed directly in the global path, forcing the robot to change its path

Figure 19 shows the first scenario, where an obstacle is placed in the global path created by the navfn algorithm to reach the destination. The AMR must now recognize the object and represent it in the local costmap as occupied space, so a new route can be planned. The robot autonomously decides to change direction away from the target destination and its initial route, after identifying an obstacle in its path that will result in a collision, as seen in figure 20. The diversion results in the AMR successfully reaching its destination after taking the new shortest possible path in free space, that avoids an impact with any obstruction.

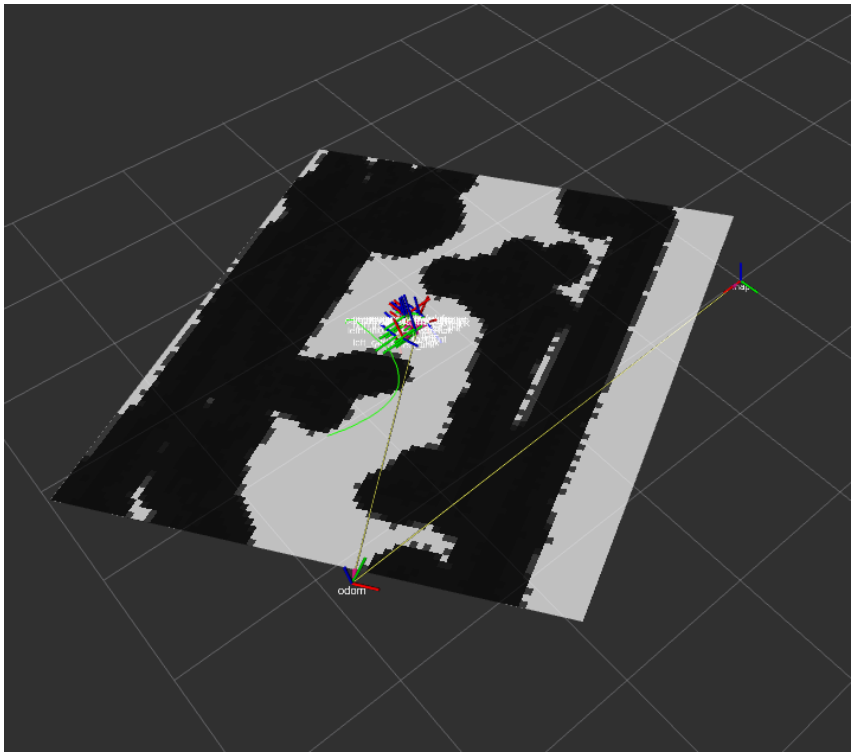


Figure 20: Local costmap showing the obstacle being mapped and an alternative route created to avoid a collision

Scenario 2

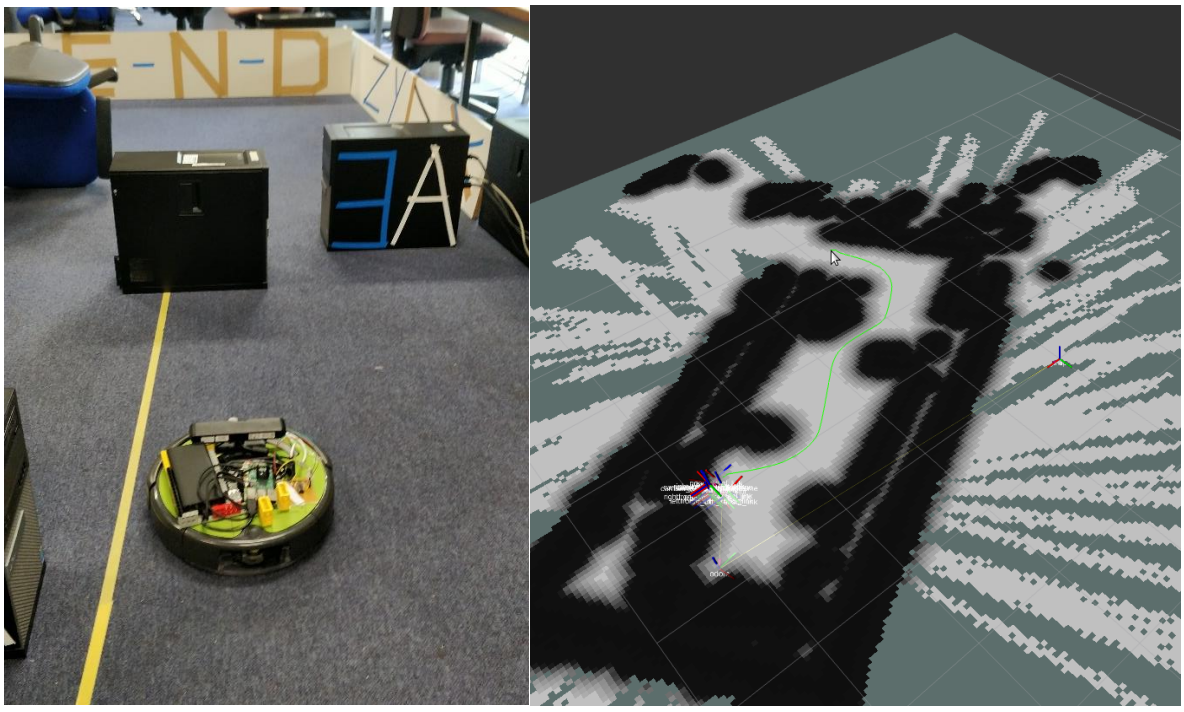


Figure 21: An obstacle placed directly in the global path, forcing the robot to change its path.

Figure 21 demonstrates the scenario whereby the AMR's global path crosses a new object, not present in the static map. The obstacle has free space on either side, forcing a decision to be made as the robot approaches. The right side will probably result in a collision and the other side offers enough clearance to safely navigate to the destination. The AMR must decide which is the lowest cost route and reassess its route accordingly. Figure 22 exhibits the AMR's robust adaptability, where it changed its path to the left of the obstacle, avoiding a potential collision. This was repeated with the object placed at slightly different positions and the robot demonstrated a consistent ability to identify the safest path when two options were available.

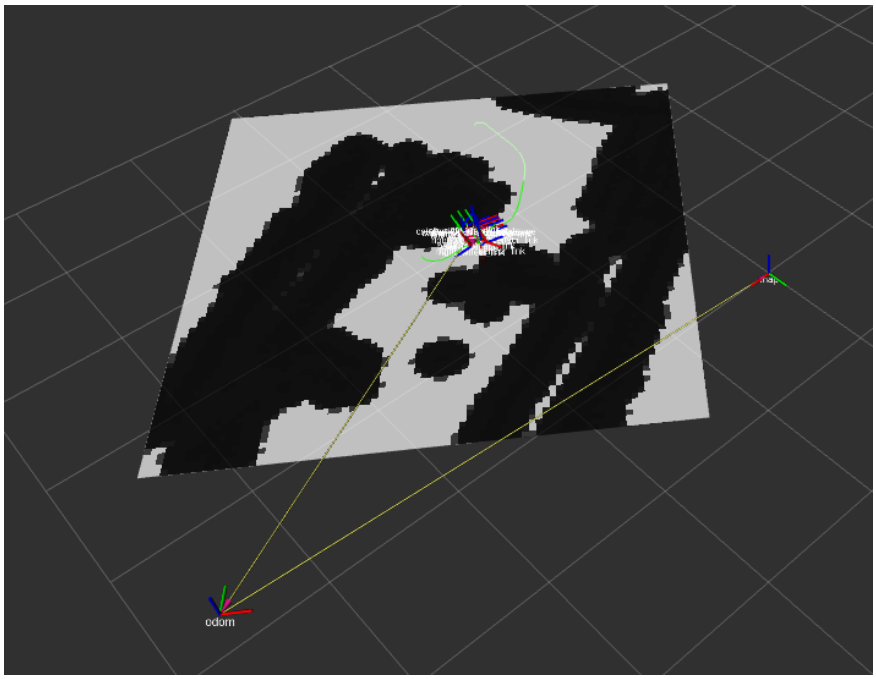


Figure 22: Local costmap showing the obstacle being mapped and an alternative route created to avoid a collision

Scenario 3

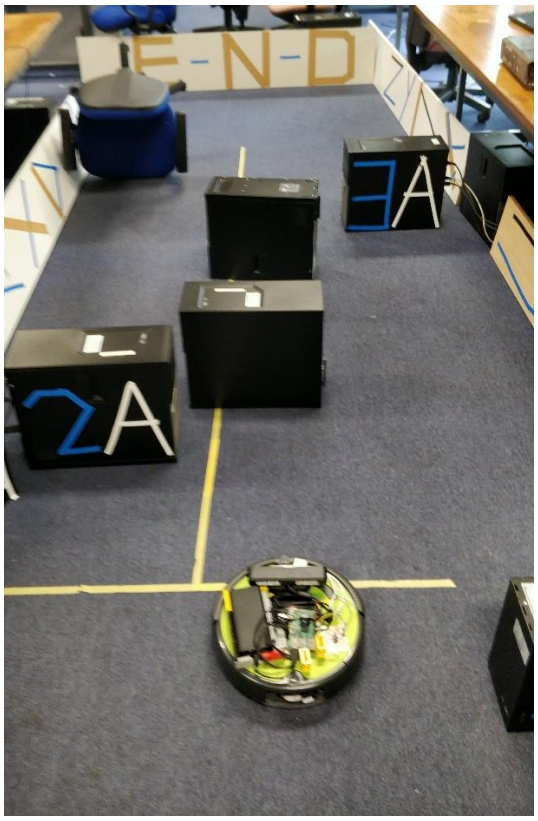


Figure 23: Two obstacles placed directly in the global path, forcing the robot to change its path

Scenario 3 used the same starting point and destination as the previous experiment. Two obstacles were placed in the environment, shown in figure 23. The AMR encountered both in quick succession and the second obstacle is obscured from view until the first has been cleared. This presents a challenge, as the robot must make a very quick decision and rapidly update the map. As with scenario 2, the second obstacle has free space on both sides, forcing the AMR to decide on the safest or quickest path. The result is shown in figure 24 below, where the initial obstruction has just been cleared successfully. The inflation of the objects causes the system to realise that a right turn will likely result in a collision, and it intelligently opts for a left turn. This choice provides enough free space to traverse collision free.

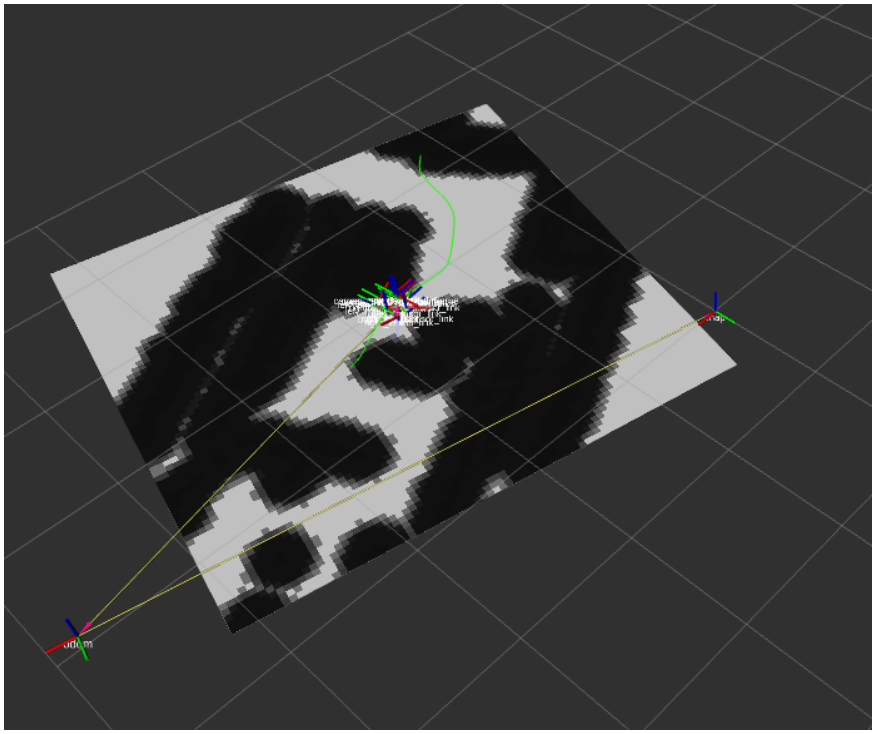


Figure 24: Local costmap showing the two obstacles being mapped and an alternative route created to avoid a collision

To test repeatability, the experiment was carried out 10 times. If the AMR did not collide with an object, it was considered a success. For eight out of ten attempts the system was able to guide itself collision free to the destination. While 80 % shows the robot is mostly functional, to be trusted and deployed in the field this rate would need to be 100 %. As any mistake could result in injury or damage. Optimisations would be carried out on the existing system if more time were available to reduce the probability of crashing to almost 0 %. Additional sensors, such as the Roomba's IR beams, would be integrated to detect objects at all angles. The current set up has a limited field of view, particularly at its peripherals. Most collisions occurred when the AMR cut too close to a corner and the edge of its wide front section collided with a part of the object outside the camera's field of vision. The Roomba's side facing IR sensors would be extremely beneficial for detecting an imminent collision of this type. Attempts were made to mitigate this risk by increasing the inflation radius, but a trade-off had to be met, as if it was set too large, the limited free space in the environment would be excessively invaded.

Scenario 4

The final experiment provided the most complexity and was used to investigate the full capability of the system. An object was placed in the middle of the corridor, offering two viable routes for the AMR. Whichever route was chosen was then blocked as the robot rounded the corner. The only way of progressing was to backtrack and instead use the alternative route. When presented with this problem the AMR reversed away from the obstructions and rotated around looking for other options, as it did in other scenarios when stuck. However, it never drives back along its path, as the situation would require if it were to find a route to the destination. Instead, the robot continues looking for a route in the forward and lateral directions, between its current position and the destination. This demonstrates a limitation in the system that would require further development and more intelligent path planning to overcome this problem. A solution to this issue is improving the recovery behaviour in move_base, requiring customisation and improvements by including more sophisticated plugins and optimising parameters.

5. Limitations

The main limitation in the design was the limited number of sensors deployed in the system. While the camera provided rich data about the surroundings, it was limited to a 58° horizontal view. The AMR had no way of knowing if the environment had changed to its rear or peripheral since the section was last observed and mapped. This would present considerable safety concerns in a dynamic environment with many moving parts. If the system was expanded to a larger area the wheel odometry data used in the navigation stack would introduce a cumulative error, worsening with increasing distance travelled. This would adversely affect the system's ability to accurately localise. The Raspberry Pi's performance capabilities would also limit the system in this respect. Prior to the processor's performance degrading, it could just about cope with the demands of the system

in mapping, localising and navigating in the small, controlled environment. In fact, it may have been damaged because of the navigation task overloading the hardware. This implies that the Raspberry Pi may have struggled to facilitate the AMR running in a larger environment, that would require far larger processing power with incrementally more demanding image processing. While the robot could react well to most obstacle avoidance scenarios, it became trapped when entering a cul-de-sac that would require backtracking. An intelligent solution would need to be found to overcome this shortfall, as it is a situation that could easily arise in the real-world applications. Another notable weakness was that the AMR had not been field tested, making it difficult to predict how it might react in a more realistic environment.

6. Future Work

To make the system safer and more aware of its surroundings, object detection sensors should be spaced all around the circumference of the Roomba. IR beams and detectors could be used to sense hazards that were outside the camera's field of view since the area was last reserved. When attempting to escape from an area that the AMR had become stuck, it often reversed blindly. These sensors would provide information on obstructions to its rear that it cannot see. They would almost completely remove the risk of collision with slow moving obstacles and could be integrated with the current navigation stack by publishing data through ROS. The limitations of the Raspberry Pi's processing power could be overcome by choosing a microcontroller with higher specifications. Also, due to the 8 GB version not being available, the 4 GB device was bought. It would be recommended to use an 8 GB version to speed up the processing speed, to better manage real-time tasks and update the map more rapidly. Due to the demands of the image processing tasks involved in the system, a more powerful processor and at least 8 GB of RAM would facilitate the operational environment to be expanded significantly.

The overall performance of the navigation stack would have been improved if time had permitted the integration of the fused odometry. Furthermore, when the system was expanded to larger areas the cumulative error introduced by the wheel odometry would have been mitigated. The parameters in the Kalman filter could have been optimised to ensure that the data would be as precise as possible to the real-world behaviour of the AMR. A customised alteration of the recovery behaviour could allow the AMR to intelligently escape from the cul-de-sac problem. This would involve integrating more sophisticated plugins and adding routines for the robot to use in certain situations after all other options had been exhausted.

The final weakness could be overcome if the AMR was tested in a larger and more dynamic environment, with a large variety of potential scenarios simulated. After a satisfactory performance was achieved and the system was considered robust enough to put into the field, the AMR should be trailed in an industrial indoor environment. Feedback from the user could be very beneficial in enhancing its performance to a point where it could be considered adequate to be deployed.

7. Conclusion

The project achieved the goal of creating a low-cost AMR capable of mapping, localising, and navigating in an environment. Obstacle avoidance was successfully implemented on the system, but some limitations exist, with several solutions proposed that could be carried out if more time had been allowed. The robot exhibited a robust ability to adapt to a changing environment, showing its potential to be optimised and deployed in an industrial setting. While the mapping and localisation sections were satisfactory and adequate for a small environment, if the scope was increased a more powerful processor would be required to scale up the existing system. The navigation stack was effective, but some limitations may have an adverse impact on the success of the project if deployed in the field. It is advised that these limitations be solved for the project to develop further.

References

[1] Grand View Research, "*Autonomous Mobile Robots Market Size, Share & Trends Analysis Report By End-use (Wholesale & Distribution, Manufacturing), By Type (UAVs, Goods-to-Person Picking Robots), By Battery Type, And Segment Forecasts, 2020 – 2027*", Grand View Research, (2021).

[2] Alessandro Mulloni et al, "*Indoor Positioning and Navigation with Camera Phones*", IEEE, (2009).

[3] Malik Haris and Jin Hou, "*Obstacle Detection and Safely Navigate the Autonomous Vehicle from Unexpected Obstacles on the Driving Lane*", MDPI, (2020).

[4] Emmanuel A. Oyekanlu et al, "*A Review of Recent Advances in Automated Guided Vehicle Technologies: Integration Challenges and Research Areas for 5G-Based Smart Manufacturing Applications*", IEEE, (2020).

[5] Giuseppe Fragapane et al, "*Increasing flexibility and productivity in Industry 4.0 production networks with autonomous mobile robots and smart intralogistics*", Springer Link, (2020)

[6] M.Brettel et al, "*How virtualization, decentralization and network building change the manufacturing landscape: An Industry 4.0 Perspective*", ResearchGate, (2014).

[7] A.Liaqat et al, "*Autonomous mobile robots in manufacturing: Highway Code development, simulation, and testing*", Springer Link, (2019).

[8] Oron Products page, [Online]. Available: *https://industrial.omron.co.uk/en/products/mobile-robot*, (Visited on August 9th 2021).

[9] American Shipper article on Omron mobile robots, [Online]. Available: *https://www.freightwaves.com/news/special-coverage-robot-maker-omron-adept-faces-unique-logistical-*

challenges#:~:text=The%20mobile%20robots%20cost%20between,other%20countries%20were%20early%20adopters, (Visited on August 9th, 2021).

[10] Omron's account of its Robots deployed at Skoda, [Online]. Available: *https://industrial.omron.co.uk/en/solutions/blog/skoda-auto-mobile-robot*, (Visited on August 9th, 2021).

[11] AngelaRojas et al, "*A dense disparity map of stereo images*", ScienceDirect, (1998).

[12] Technical information about SMP Robotics' products, [Online]. Available: *https://smprobotics.com/* , (Visited on August 10th, 2021)

[13] Shibo Zhao and Fang Zheng "*Direct Depth SLAM: Sparse Geometric Feature Enhanced Direct Depth SLAM System for Low-Texture Environments*", NCBI, (2018).

[14] Mary B Alatise, "*A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods*", IEEE, (2020).

[15] Naveen Appiah and Nitin Bandaru, "*Obstacle detection using stereo vision for self-driving cars*", Semantic Scholar, (2018).

[16] J Levinson, "*Traffic light mapping, localization, and state detection for autonomous vehicles*", IEEE, (2018).

[17] Jelena Ketocić et al, "*Sensors and Sensor Fusion in Autonomous Vehicles*", IEEE, (2018).

[18] Morgan Quigley, "*ROS: an open-source Robot Operating System*", ICRA workshop on open source software, (2009).

[19] Chao Duan et al, "*Deep Learning for Visual SLAM in Transportation Robotics: A review*", Oxford Academic, (2020)

[20] Khalid Yousif et al, "*An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics*", Springer Link, (2015).

[21] Mathieu Labbe, "*RTAB-Map as an Open-Source Lidar and Visual simultaneous localization and mapping library for large-scale and long-term online operation*", IEEE, (2018).

[22] Mathieu Labbe , "*Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation*", IEEE, (2013).

[23] Sung-Hyeon Joo et al, "*Autonomous Navigation Framework for Intelligent Robots Based on a Semantic Environment Modeling*", MDPI, (2020).

[24] Illah Reza Nourbakhsh and Roland Siegwart, "*Introduction to autonomous mobile robots*" , pg.257-291, (2006).

[25] Daniel R. Lanning, "*Dijkstra's Algorithm and Google Maps*", Semantic Scholar, (2014).

[26] Description of Dijkstra's and A* algorithms, [Online]. Available: *http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html*, (visited on August 14th 2021).

[27] Documentation on ROS costmaps, [Online]. Available: *http://wiki.ros.org/costmap_2d*, (visited on August 15th 2021).

[28] Aquino-Jr, Fagner de Assis Moura Pimentel and Plinio Thomaz , "*Evaluation of ROS Navigation Stack for Social Navigation*", Semantic Scholar, (2021)

[29] iRobot Roomba 600 Open Interface Spec, [Online]. Available: *https://www.irobotweb.com/~/media/MainSite/PDFs/About/STEM/Create/iRobot_Roomba_600_Open_Interface_Spec.pdf,* (visited on August 17th 2021).

[30] Mohan Rajesh Elara, "*Design principles for robot inclusive spaces: A case study with Roomba*", IEEE, (2014)

[31] IRobot Roomba prices, [Online]. *https://store.irobot.com/*, (visited on August 17th 2021).

[32] Raspberry Pi 4 Model B specifications, [Online]. *https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/*, (visited on August 17th 2021).

[33] Raspberry Pi 4 Model B prices, [Online]. *https://uk.rs-online.com/web/p/raspberry-pi/1822096/,* (visited on August 17th 2021).

[34] Battery pack specifications and price, [Online]. *https://www.mi.com/global/20000mAh-mi-power-bank-3-pro*, (visited on August 17th 2021).

[35] Information on depth sensing, [Online]. *https://www.intelrealsense.com/beginners-guide-to-depth/*, (visited on August 18th 2021).

[36] Asus Xtion Pro Live information and price, [Online], *http://xtionprolive.com/asus-xtion-pro-live*, (visited on August 18th 2021).

[37] David Nistér et al, "*Visual odometry for ground vehicle applications*.", Journal Field of Robotics, (2006).

[38] Mohammad O. A. Aqel et al, "*Review of visual odometry: types, approaches, challenges, and applications*", Springer Open, (2016).

[39] Kaiyu Zheng , "*ROS Navigation Tuning Guide*", Semantic Scholar, 2016

[40] Documentation on base_local_planner, [Online]. *http://wiki.ros.org/base_local_planner*, (visited on August 23rd 2021).

Appendix

A.

The following procedure was used to install ROS Noetic on the device:

Always begin by ensuring the system is up to date as possible:

$ sudo apt update && sudo apt upgrade

To make the OS accept the ROS package use the following line:

$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

To allow the device to communicate with the server a package must be installed, and the keys set up:

$ sudo apt install curl

$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add –

Install the desktop version as it comes with useful software such as Rviz but does require more memory:

$ sudo apt install ros-noetic-desktop

When ROS is used it must be sourced each time to work. This can be done automatically with the following commands:

$ echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

$ source ~/.bashrc

Various dependencies must be installed to facilitate the download of ROS add-ons:

$ sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential

Before installation can be carried out all the ROS dependencies must be acquired using the following command:

$ sudo apt install python3-rosdep

Not all dependencies are satisfied and libconsole must be made from source in order for ROS to work:

$ ~ros comm# git clone https://github.com/ros/console bridge

$ ~ros comm# cd console bridge

$ ~ros comm\console bridge# cmake src

$ ~ros comm\console bridge# make install

$ ~ros comm\console bridge# export CMAKE PREFIX PATH=/home/ .. root/ros comm/console bridge

After the next two commands are ran successfully, ROS Noetic should be operational

$ sudo rosdep init

$ rosdep update

To verify that it has been installed run and 'Noetic' should be returned:

$ rosversion -d

B.

The following guide details the procedure to install the create_autonomy libraries:

As the package is built using catkin the following installation is carried out and the libboost dependency added:

$ sudo apt-get install python-rosdep python-catkin-tools

$ sudo apt-get install build-essential cmake libboost-system-dev libboost-thread-dev

Firstly a C++ library for integrating with the IRobot system is installed called libcreate:

A catkin workspace is created where libcreate will be installed:

$ mkdir -p create_ws/src

$ cd create_ws

$ catkin init

$ cd src

The package is downloaded from Github into the workspace and build using catkin:

$ git clone https://github.com/AutonomyLab/libcreate.git

$ catkin build

Ensure to run this command line prior to installing the package with catkin or it will only be able to 50 % complete the process:

$ sudo usermod -a -G dialout $USER

Now the create_autonomy can be installed when this is satisfied. The repo is cloned to the folder:

$ cd ~/create_ws/src

$ git clone https://github.com/autonomylab/create_robot.git

Now create_autonomy can be built using catkin:

$ cd ~/create_ws

$ catkin build

C.

RTABmap is quite simple to install when ROS is fully operational and can be completed with a single command:

$ sudo apt-get install ros-noetic-rtabmap-ros

Another problem was encountered with the Wi-Fi disconnecting when the ethernet cable was plugged out, to allow mobile connection the robot. This had to be solved by changing the priority order of the WLAN and ETH0 connections, so that the Wi-Fi would take priority. The first command shows the unique identification address of the Raspberry Pi on both networks.

$ route-n

$ sudo route delete default gateway Raspberry_Pi_WLAN0_address

$ sudo route add default gateway Raspberry_Pi_WLAN0_address