Micael Alves Pires

NATURAL NAVIGATION SOLUTIONS FOR AMRS AND AGVS USING DEPTH CAMERAS

Dissertation in Engineering Physics

September of 2021

1 2 9 0

UNIVERSIDADE Ð COIMBRA

University of Coimbra

Faculty of Science and Technology

Department of Physics

Natural navigation solutions for AMRs and AGVs using depth cameras

Micael Alves Pires

A dissertation presented for the degree of Engineering Physics

Coimbra, September of 2021

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Natural navigation solutions for AMRs and AGVs using depth cameras

Supervisor:
Dr. Fernando Santos

Co-supervisor:
Dr. José Paulo Pires Domingues

Jury:
Dr. João Manuel Rendeiro Cardoso
Dr. David Bina Siassipour Portugal
Dr. Fernando Santos

A dissertation submitted in partial satisfaction of the requirements for the degree of Master in Engineering Physics

Coimbra, September of 2021

Contents

Acknowledgements iii

Abstract v

Resumo vii

List of Acronyms ix

List of Figures xi

List of Tables xiii

I Introduction 1

1.1 Context and Motivation 2

1.2 Objectives 3

1.3 Outline of the Dissertation 3

II State of the Art 5

2.1 Sensors 5

2.2 Algorithms 6

2.2.1 Fundamental Algorithms 7

2.2.2 Implementations available in ROS 11

2.2.2.1 Hector-SLAM 11

2.2.2.2 Gmapping 13

2.2.2.3 RTAB-Map 14

2.2.2.4 Algorithm Comparison 15

2.2.3 Pathfinding 15

III Implementation 17

3.1 Hardware Description 17

3.2 First Attempts 21

3.3 ROS-based Implementation 22

3.3.1 Brief description of ROS 22
3.3.2 Implementation of SLAM algorithms 23
3.3.2.1 HectorSLAM 23
3.3.2.2 IMU fused RGB-D Odometry 24
3.3.2.3 Gmapping 26
3.3.2.4 RTAB-Map 27
3.3.2.5 Navigation Stack 27
3.3.3 Object Detection 29

3.4 Tests Description **32**

IV Experimental Results **39**

4.1 Mapping **39**
4.1.1 Static Environments 41
4.1.1.1 HectorSLAM 41
4.1.1.2 Gmapping 42
4.1.1.3 RTAB-Map 43
4.1.1.4 Comparison between Depth Camera and LIDAR 46
4.1.2 Dynamic Environments 46
4.1.2.1 Gmapping 48
4.1.2.2 RTAB-Map 49

4.2 Pathfinding **51**

V Conclusion and Future Work **57**

References **59**

.

Acknowledgements

I would like to start by thanking my supervisors Dr. Fernando Santos and Dr. José Paulo Domingues for the help provided during the development of this work. I would also like to thank Active Space Technologies for giving me the tools and the opportunity to study this subject and write this dissertation. I extend my thanks to the colleagues in Active Space Technologies who helped performing the tests for this work.

I also want to thank my friends for accompanying me during my time in University. Last but not least, I would like to thank my parents for all the help and support that made me come this far and that will keep allowing me to reach all my goals.

Abstract

Automation is one of the most fast developing fields of robotics. With the ever growing interest in automating processes, both for efficiency or safety, the ability to provide robots with autonomous navigation capabilities is of paramount significance. The first approaches to enable autonomous robot navigation involved magnetic strips on the floor, with the robot following a pre-established path. However, problems arise whenever it is necessary to change the trajectory of interest or if the magnetic tape is damaged. With the development of depth sensors, namely LIDARs (Light Detection and Ranging) and depth cameras, a new way of tackling this challenge has emerged: SLAM (Simultaneous Localisation and Mapping). The SLAM problem is far from trivial in the way that it implies two seemingly paradoxical questions ("How to build a map without knowing the robot's location?" and "How to locate the robot without a map?").

Throughout this work, some SLAM methods (HectorSLAM, Gmapping and RTAB-Map) are implemented and evaluated using ROS (Robot Operating System). An Intel RealSense D435i depth camera is used as the main sensor and various practical tests are made using an AGV (Automated Guided Vehicle) developed by Active Space Technologies. Tests consist in recording depth and IMU (Inertial Measurement Unit) data from the depth camera with the AGV serving as a mobile platform, exploring our test location. Measurements are then available for offline analysis through ROS bag files. These files allow multiple runnings of the same test and direct assessment of the different algorithms by comparing the resultant maps. Furthermore, pathfinding is implemented, which allows for full robot autonomy.

In order to optimise mapping techniques, some object detection algorithms are added to the system to deal with various obstacles that can appear in the robot's sight and degrade the quality of the results, e.g., moving people. Through the use of object detection algorithms, people are detected and removed from the depth image. Then, the resulting maps with and without the application of this depth image filter are compared. Object detection algorithms are also used to add another layer to navigation, yielding the ability to stop and resume the velocity commands provided by the pathfinding algorithm through the interpretation of traffic lights.

This work is done as part of the AMR-UVC project developed by Active Space Technologies. This is a project that seeks to expand the AGV offering of the company by developing a fully autonomous AGV with disinfection capabilities. The project is motivated by the ongoing pandemic and the final product can be used in any medical or industrial setting that requires decontamination from the SARS-CoV-2 or other biological agents.

Keywords: SLAM, AGV, AMR, ROS, Depth Camera, Navigation, Pathfinding, Object Detection

Resumo

A automação é um dos campos de desenvolvimento mais rápido da robótica. Com o crescente interesse na automação de processos, tanto para eficiência como para segurança, fornecer aos robôs a capacidade de navegação autónoma é de suprema importância. As primeiras abordagens para possibilitar navegação autónoma de robôs envolveram a colocação de bandas magnéticas no chão, traçando uma trajetória pré-definida. No entanto, surgem problemas sempre que é necessário alterar a trajetória de interesse ou quando a banda magnética se danifica. Com o desenvolvimento de sensores de profundidade como os LIDARs (Light Detection and Ranging) e câmaras de profundidade (depth cameras), surgiu uma nova maneira de lidar com este desafio: o SLAM (Simultaneous Localisation and Mapping - Localização e Mapeamento Simultâneos). O SLAM é um desafio que está longe de ser trivial pois implica duas questões aparentemente paradoxais ("Como construir um mapa sem saber a localização do robô?" e "Como localizar um robô sem ter acesso a um mapa?").

Ao longo deste trabalho, alguns métodos SLAM (HectorSLAM, Gmapping e RTAB-Map) são implementados e avaliados usando ROS (Robot Operating System). Uma câmara de profundidade Intel RealSense D435i é usada como o sensor principal e vários testes práticos são feitos usando um AGV (Automated Guided Vehicle) desenvolvido pela Active Space Technologies. Estes testes consistem em gravar dados de profundidade e dados provenientes de uma IMU (Inertial Measurement Unit - dispositivo de medição de inércia) contida na câmara com o AGV servindo de plataforma móvel para explorar o local de testes. As medições ficam disponíveis para análise offline através de ficheiros ROS bag. Estes ficheiros permitem várias execuções do mesmo teste e avaliação direta dos diferentes algoritmos através da comparação dos mapas resultantes. Para além disso, um algoritmo de pathfinding é também implementado, o que permite total autonomia do robô.

Com o intuito de otimizar técnicas de mapeamento, alguns algoritmos de deteção de objetos são adicionados ao sistema para lidar com um tipo de obstáculos que pode aparecer à frente do robô e diminuir a qualidade dos resultados: movimentação de pessoas. Através do uso da deteção de objetos, as pessoas são detetadas e removidas da imagem de profundidade. Em seguida, comparam-se os mapas resultantes, com e sem a aplicação deste filtro da imagem de profundidade. Algoritmos de deteção de objetos são também utilizados para adicionar outra camada à navegação, adicionando a capacidade de parar e retomar a publicação de comandos de velocidade fornecidos pelo algoritmo de pathfinding através da interpretação de semáforos.

Este trabalho é feito no âmbito do projeto AMR-UVC desenvolvido pela Active Space Technologies. Este projeto visa expandir a oferta de AGVs da empresa através do desenvolvimento de um AGV totalmente autónomo com capacidades de desinfeção. O

projeto é motivado pela pandemia atual e o produto final poderá ser usado em qualquer ambiente médico ou industrial que requeira descontaminação do SARS-CoV-2 ou outros agentes biológicos.

Palavras-chave: SLAM, AGV, AMR, ROS, Câmara de Profundidade, Navegação, Pathfinding, Deteção de Objetos.

List of Acronyms

AGV Automated Guided Vehicle.

AMR Automated Mobile Robots.

API Application Programming Interface.

DOF Degrees of Freedom.

EKF Extended Kalman Filter.

FPS Frames Per Second.

HSV Hue Saturation Value.

ICP Iterative Closest Point.

IMU Inertial Measurement Unit.

KF Kalman Filter.

LIDAR Light Detection and Ranging.

NDT Normal Distributions Transform.

RGB Red Green Blue.

RGB-D Red Green Blue - Depth.

ROS Robot Operating System.

SLAM Simultaneous Localisation and Mapping.

UAV Unmanned Aerial Vehicle.

UKF Unscented Kalman Filter.

USV Unmanned Surface Vehicle.

List of Figures

2.2.1 Example of an Occupancy Grid Map [1]. 8
2.2.2 Pose-graph, the circles represent the nodes of the graph, connected by edges (or links) [2]. 10
2.2.3 Map before (a) and after (b) a loop-closure/pose graph optimisation process has occurred [2]. 11
2.2.4 Overview of the mapping and navigation systems used in HectorSLAM [3]. . . 12
2.2.5 Difference in particle distributions on different environments using the improved proposal distribution [4]. 14
3.1.1 Intel RealSense D435i. 17
3.1.2 Colour image and pointclouds of a flat wall. 19
3.1.3 Active Space developed ActiveTwo AGV. 20
3.1.4 AGV in test conditions. 21
3.3.1 Flow chart for our HectorSLAM implementation. 24
3.3.2 Flow chart for our Gmapping implementation. 26
3.3.3 Flow chart for our RTAB-Map implementation. 27
3.3.4 Flow chart for our Navigation Stack implementation with traffic light obedience. 29
3.3.5 Flow chart for the implementation of Yolact. 32
3.4.1 Depth and colour images from the D435i depth camera in front of a window. . 33
3.4.2 Pictures of ongoing SLAM tests. 35
3.4.3 Object Detection API tests. 37
4.1.1 Map built using a high precision Velodyne Puck VLP-1614 laser scanner. . . 39
4.1.2 Pictures of the four rooms of the test site. 40
4.1.3 Static environment map from HectorSLAM. 41
4.1.4 Static environment map from Gmapping. 42
4.1.5 Typical Gmapping failures. 43
4.1.6 Example of a loop closure pose graph optimisation in RTAB-Map. 44
4.1.7 Static environment results from RTAB-Map. 45
4.1.8 Removal of people from depth image using Yolact. 47
4.1.9 Gmapping dynamic environment results for the full lap. 48
4.1.10 Gmapping dynamic environment results for the eight-shaped trajectory. . . . 49
4.1.11 RTAB-Map dynamic environment results for the full lap. 50
4.1.12 RTAB-Map dynamic environment results for the eight-shaped trajectory. . . . 51
4.2.1 Trajectory calculation and update according to the map. 53
4.2.2 Corrected trajectory according to a new obstacle. 54
4.2.3 Red traffic light effect on navigation. 55
4.2.4 Yellow traffic light detection. 56

4.2.5 Green traffic light effect on navigation. 56

List of Tables

2.2.1 Comparison between the discussed SLAM implementations available in ROS. 15
3.1.1 Depth measurements from the camera. Every measurement corresponds to the depth at the centre of the depth image. 18

Chapter I

Introduction

How to find our way through the world? How to navigate to a certain place whilst knowing how to come back efficiently? What route should we take? These are questions the human race has always thought about since the beginning of existence. Be it in hunter-gatherer times when humans had to find a place to take shelter, the best hunting spots or the most fertile grounds for agriculture, or nowadays, whenever we want to plan a trip to a new country, navigate through traffic to the supermarket or just having a leisurely walk through the park.

Nowadays, navigation is not just a problem for us humans but it is also a developing field in robotics. With the increasing interest in automating processes and their efficiency comes the need to attribute this ability to robots. Robots may have the most variable forms and applications, they can be terrestrial AMRs (Autonomous Mobile Robot) or AGVs (Autonomous Guided Vehicles), aquatic USVs (Unmanned Surface Vehicles), or even flying UAVs (Unmanned Aerial Vehicles). Applications are endless, for every process that involves a human controlling a vehicle and navigating through a certain area there is an opportunity for automation. Examples include the operation of a forklift, driving a car, oceanography and military and patrol operations in land, sea and air.

The way mankind has performed navigation has evolved during the many centuries of our existence, beginning with the position of the Sun during the day and the position of the stars at night, evolving to compasses and the more advanced methods we use today like GPS and RADAR navigation. Some of these methods and a large number of different sensors are available for robots, but in this dissertation we will focus on the use of depth cameras with an IMU (Inertial Measurement Unit) module.

For many years, the most popular method for robot navigation purposes was navigation by means of magnetic strips placed on the floor tracing the path the robot should follow. But this method becomes cumbersome whenever we want to change the robot trajectory for any reason or if the magnetic tape is somehow damaged.

Another solution to the navigation problem is the use of lasers and installation of fixed, strategically placed reflective targets as reference points, but this also requires preinstallation of these targets and it reduces the AGV's movement to a particular path.

The natural evolution to this previous approach is to develop a system that guides the robot through vision without any previously placed landmarks, in a process called natural navigation. To perform natural navigation we can use different types of sensors, the main

ones in terms of vision being depth cameras and LIDAR sensors, but, additional sensors like IMUs and/or odometry provided by wheel encoders are usually required.

Using the data provided by the depth camera and IMU we can build a map of our surroundings and locate ourselves on that map, in a process called SLAM (Simultaneous Localisation and Mapping). For this, three different algorithms, HectorSLAM, Gmapping and RTAB-Map, are implemented and optimised to a real-world data-set recorded in the premises of Active Space Technologies. An additional algorithm is used to perform path planning and navigation. Furthermore, some object recognition algorithms are added to the system to try and improve our results and also to implement control of the robot navigation through traffic lights.

Some tasks of this work have been fulfilled as part of the NavCam contest, an open competition organised by Active Space Technologies to develop SLAM technology. This contest was open to a national scale and various teams from different portuguese universities (Lisbon, Porto, Braga,...) have participated. This work, part of a broad approach to address SLAM technologies, was awarded the first place in the competition which consisted in a monetary award. The public award ceremony took place June 10th, 2021 in the Physics Department of University of Coimbra with presence of the teams, professors, students and the organisers.

This section serves as an introduction to this thesis and its chapters and defines how they will be organised, highlighting this work's goals and explaining the context and motivation for this project.

1.1 Context and Motivation

There is an ever-growing interest and need to automate tasks performed by humans with the assistance from robots. This trend strives for efficiency with the aim of doing things safer and faster, by removing the limits of the human being from the equation.

The ongoing pandemic has shown proof that this is a subject which should be invested in. When the Covid-19 epidemic started, many people had to be put at risk of infection to do their jobs, particularly healthcare workers. There should be a way of disinfecting work areas without subjecting people to direct contact with viruses. This is true for Covid-19 or any other situation where these professionals are subjected to risk of infection to provide medical care to a person or to do scientific research.

This is where Active Space Technologies' project "AMR-UVC[1]" comes in. This project consists in developing an AGV equipped with a UV light for disinfection of possibly infected

[1]AMR stands for Autonomous Mobile Robot and UVC for UltraViolet radiation in the C-band

areas. This product has hospitals as its main potential clients but can be sold to any interested organisations. For this purpose it is important that the AGV has the capability to navigate on its own and unaided throughout the whole area of interest, so that is why implementing natural navigation is so important. Moreover, the AGV will not only have this application in mind but can also be used in any industrial or commercial setting that relies on specific tasks such as logistics. The AGVs are easily adapted to these contexts and have been sold with similar intentions before, although with magnetic navigation. Thus, by developing a natural navigation system that is applicable to existing AGVs developed by Active Space Technologies, the company can diversify and extend their product offering for industrial automation.

1.2 Objectives

The main objective of this dissertation is to develop a natural navigation solution for AGVs and AMRs, allowing the robot to localise itself and navigate unaided in an unknown environment. The system shall build a map of its environment, localise itself on the map and define the optimum trajectory to any destination. This work will take advantage of the sensors provided by the Intel RealSense D435i depth camera module (depth camera and IMU).

Additionally, extra features will be developed through the usage of object detection algorithms in order to add an extra layer to the navigation. This includes responding to traffic lights and improvement of the maps, removing some possible spurious landmarks. An example of these false landmarks comes from people walking across the camera's field of view.

This project emerged as a response to the SARS-CoV-2 pandemic to help with the fight against the spread of the virus by building an autonomous disinfection robot and also as a diversification of the autonomous AGV offering already provided by Active Space Technologies.

1.3 Outline of the Dissertation

Chapter II (State of the Art)

This chapter discusses and compares the operation of available methods to perform SLAM. The main focus is on the three algorithms available in ROS (Robot Operating System) used in this thesis: HectorSLAM, Gmapping and RTAB-Map, and on some of the more more fundamental tools used to implement these algorithms.

Chapter III (Implementation)

The third chapter describes the hardware, the first MATLAB-based approach for SLAM and the ROS-based implementation of the various algorithms used across this work. The SLAM algorithms and their key parameters are described as well as the implementation of object detection techniques and their integration with SLAM. The description of the various tests performed during the development of this work is also included in this chapter.

Chapter IV (Experimental Results)

In Chapter IV, the most important results are presented and analysed. Mapping and pathfinding results are split into two main sections. The mapping results are further divided between static and dynamic environments. For both types of environments, some resulting maps and trajectories are presented for the SLAM algorithms discussed. In the dynamic environments section, the addition of object detection techniques for people removal from the camera data is also discussed. In the pathfinding part we present the results of trajectory planning and the integration of a traffic light interpretation algorithm to control the AGV's navigation.

Chapter V (Conclusions and Future Work)

In the final chapter the conclusions are presented. There is also a summary concerning the algorithms that offer best performance for SLAM implementation. Pathfinding optimisation and the addition of object detection algorithms is also evaluated. Some suggestions for future work are also underlined in this chapter.

Chapter II

State of the Art

SLAM, as the name clearly defines, is the process of building a map while simultaneously knowing the robot's position on that map. It is obvious why this problem is so difficult because it has the characteristics of a ”chicken-and-egg” problem. How to build a map when we do not know where we are, and how to localise a robot if we do not have a map are the two seemingly paradoxical questions that are the foundation to the SLAM problem.

There are a few main ways of performing SLAM and a large amount of different sensors. To this project, specific hardware has been made available: the Intel RealSense D435i depth camera. But there are other popular sensors for SLAM like LIDARs and tracking cameras.

2.1 Sensors

There is a wide variety of different sensors suitable to implement SLAM, the most powerful ones being 3D cameras and LIDARs. Many of these systems include other sensors that can be used to increase the accuracy of the SLAM algorithm.

Vision sensors are normally separated into two main groups: passive and active. Passive sensors are defined by a methodology that consists in getting depth information from images obtained through conventional cameras. Active sensors, on the other hand, are defined by radiation emitting modules which calculate depth by emitting and re-absorbing radiation. The aforementioned LIDARs are active sensors, conversely, depth and tracking cameras are examples of passive sensors [5, 6].

LIDARs work in the same way as RADARs and SONARs, but, instead of using radar waves or sound, laser pulses are emitted. These laser pulses are backscattered by the surrounding space and then detected by the LIDAR receiver. The time elapsed between every emission and absorption (Time of Flight) [5] is measured. This allows us to measure the distance to a target via a simple formula using the speed of light.

Depth cameras are split into three categories: Natural Light, Projected IR (Infrared) and Coded Light [5].

In the first group, the module comprises two RGB cameras, separated by a known dis-

tance. Depth is calculated by matching common features in both images and performing geometric calculations.

Projected IR cameras are similar to the previous group but they have an additional infrared module, which projects a texture in the scene in order to help finding common features in both images.

In Coded IR cameras, instead of having two separate colour cameras, we substitute one of those cameras by an IR projector. In this type of sensor we can derive depth using the angle of projection of the infrared radiation and the position of the infrared dots projected in the image.

Tracking cameras do not give us depth information but are extremely precise on measuring the movement of the camera and identifying its position in the three-dimensional space. These modules use two fisheye cameras and an IMU for this purpose. In order to use this device to perform SLAM, it might be needed to use another type of visual sensor, like the ones discussed previously, as the main tool. Because of this, tracking cameras are mainly used to improve the system and not as a stand-alone SLAM device [6].

Every one of the three previous types of modules has or may have an IMU installed. IMUs can be composed by accelerometers, gyroscopes and/or magnetometers and they are used to measure the orientation of a system. Accelerometers measure linear accelerations and gyroscopes measure angular velocities. The fusion of these sensors provides us the relative orientation of the device. If an IMU includes a magnetometer we can obtain the global orientation of the robot, relative to North.

Other interesting sensors that can be used in SLAM are: GPS, for outdoor applications, ultrasonic sensors, for short-range measurements, and odometry provided by wheel-encoders. Using odometry sensors is useful for relative localisation but the reconstruction of trajectory using these measurements quickly diverges.

2.2 Algorithms

In this section, we will discuss the fundamental algorithms commonly used in SLAM and some implementations of these algorithms available in ROS[2] (Robot Operating System).

[2]ROS is a robotics framework used throughout this work. Further details about ROS are discussed in Chapter III, Section 3.3.1.

2.2.1 Fundamental Algorithms

Over the years, many different methods have been developed to solve the SLAM problem. The first method was the Kalman Filter Slam (KF) that dates back many decades. In the Kalman Filter SLAM family there is also the Extended Kalman Filter SLAM (EKF) and the Unscented Kalman Filter SLAM (UKF). The last two algorithms can be applied in environments where the classic KF SLAM fails. Beyond these, Particle Filter SLAM and Graph-Based SLAM approaches are also very important.

Kalman Filter, Extended Kalman Filter and Unscented Kalman Filter-based SLAM: Every single piece of data has associated noise and this extends to the algorithms. So, we may need to apply certain methods to model these uncertainties in order to filter the data and obtain better results. The most popular algorithm to perform this is the Kalman Filter.

The Kalman Filter: The KF was one of the first ideas to solve the SLAM problem. This filter takes previous poses[3] of the robot and tries to predict the next pose using the dynamic control given to the robot and the observation provided by a visual sensor [7]. To do this, it takes advantage of two models: the *motion model* and the *observation model*. The motion model takes previous positions into account and predicts subsequent poses using the dynamic control. The observation model corrects the prediction by comparing expected observations and the actual observation obtained at every single step, after the requested movement has taken place. As implied, these two models are implemented sequentially, making up the prediction and correction steps.

The Extended Kalman Filter: As the name states, is an extension to the classic Kalman Filter, meaning that it elongates its prediction and correction capabilities to non-linear systems, where the Kalman Filter fails. The EKF linearises non-linear functions using Taylor Series Expansions.

The Unscented Kalman Filter: The UKF, described by Julier *et al.* [8], is another way of applying the Kalman Filter in non-linear systems, but, instead of using a Taylor series expansion like the EKF for linearisation, it uses the unscented transform. More details on the unscented transform can be consulted in [8]. The UKF has similar results to the EKF in linear systems, but it is better in non-linear environments. Even though the UKF has advantages, it is generally slower than the EKF.

[3] In this context, pose is defined as the robot's position and heading.

The family of KF-based SLAM algorithms usually describes the world using landmarks. These landmarks represent real world objects, either static, like corners of a room or furniture, or mobile, like people and animals. The location of these landmarks is described in matrices that are updated in every single step.

A consequence of this is that these methods become computationally expensive as the number of landmarks increases, so it might be useful to use other methods after a certain point. One alternative is the Particle Filter discussed next.

Particle Filter Based SLAM: In Particle Filter based SLAM [9, 10], the map is represented by Occupancy Grids. Occupancy Grids are a way of mapping a space by dividing it into a large number of cells, each cell holding the probability of being occupied [1]. Occupancy Grids can be graphically represented by an array of equal squares (or cells) in grayscale where darker tones mean higher probability and vice-versa.

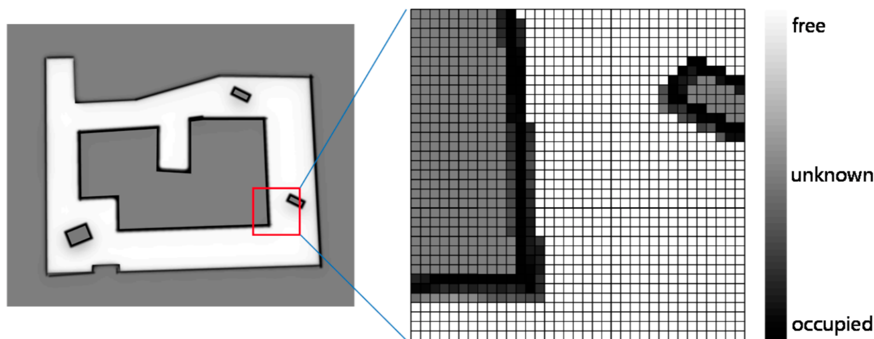


Figure 2.2.1: Example of an Occupancy Grid Map [1].

.

The Particle Filter's defining characteristic is the random generation of particles over the whole possible space, each particle representing a pose hypothesis for the robot. The way this algorithm is composed is detailed in the following steps:

1. Random sample of particles, with random poses;
2. Translation and/or rotation of every particle according to the command;
3. Calculation of particle weights;
4. Re-sampling according to their weights;

The first step consists of randomly sampling a given number of particles over the considered area.

Step two is defined by applying whatever the robot command is at the current time, be it a translation or a rotation or a combination of the two, to every sampled particle from step one.

The weights are then calculated for every particle. The more similar the particle's observation represents the robot's real observation, the higher its weight will be. For instance, when the robot is in front of a corner of a room, particles that more closely resemble that feature are given higher weights than particles that are observing a flat wall.

After the previous step, lower probability particles are eliminated and the others survive. The Particle Filter then re-samples the same number of particles as in the first step but, instead of sampling uniformly across the space, the algorithm will place a higher number of particles in zones that contained particles with heavier weights. And it is back to step two. This process is repeated until the most accurate set of particles is found, this set will expectantly define an approximation to the true robot's pose.

This algorithm, much like the EKF, can deal with non-linear systems but is computationally less expensive [11]. The nature of particle re-sampling in a "survival of the fittest" philosophy allows this method to have a multi-modal belief about the robot's pose [11], meaning it can have multiple simultaneous beliefs for the pose of the robot within the particle set, which can reduced model instability.

Every particle holds an hypothesis for the robot's current pose and trajectory. Once the most accurate particle is found, the map is then built using the observations coupled to each instant of the trajectory.

Graph-based SLAM: Graph SLAM [12] uses a different method of defining the robot's poses and trajectory. As the name suggests, the dynamics of the robot is defined by a graph. The graph is made of nodes and edges (or links, represented by lines connecting the nodes). The nodes represent various recorded poses at different times and the edges are the constraints between the nodes.

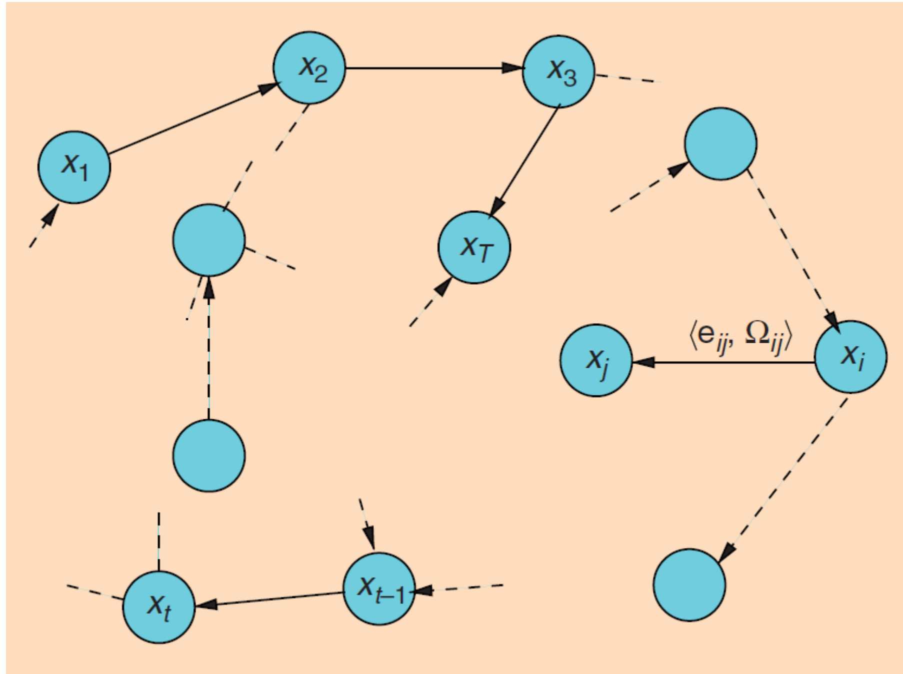


Figure 2.2.2: Pose-graph, the circles represent the nodes of the graph, connected by edges (or links) [2].

The constraints can have many different forms, and can result from different types of data and sensors. We can have constraints resulting from the visual sensors (like cameras or LIDAR), from the IMU or from any other type of sensor that binds a previous state of the robot to the next. These constraints give information between consecutive moments in time by defining the difference in the pose of the robot between those moments. IMU related constraints are easily understandable but, in the case of visual information, scan-matching algorithms [13] are used in order to register (align) pointclouds[4] or laser scans provided by the sensor. The spatial and angular difference between successive pointclouds or laser scans determines the movement of the robot. By setting enough constraints between the succeeding poses we can build the full trajectory and a map of the environment, considering that every pointcloud or laser scan is attached to a node of the graph.

Another important feature of graph-based SLAM is the ability to carry out delayed trajectory and map optimisation based on loop-closures. This is an attribute that allows us to have future corrections to our map and trajectory of the robot by re-observing a part of the robot's surroundings. When the system loops back to a previously explored part of the room, it sets a new constraint between our current pose and a preceding node of the graph. This may result in a correction to a number of previous nodes, considering the fact

[4]set of points defined by three-dimensional Cartesian coordinates that composes an object or shape.

that every node is connected by constraints to one or several other nodes. Thus, by closing the loop, we can correct possible sources of noise, for instance, odometry drift or errors in pointcloud/laser scan registration. This procedure can correct errors in the trajectory and in the map through the alignment of different sightings of the same features.

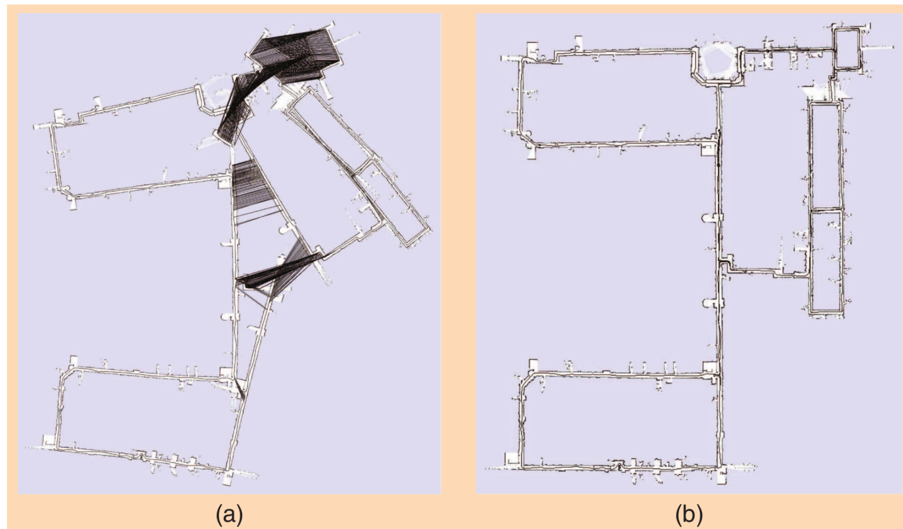


Figure 2.2.3: Map before (a) and after (b) a loop-closure/pose graph optimisation process has occurred [2].

2.2.2 Implementations available in ROS

As discussed, there are many different types of algorithms to perform SLAM. The next step is to find a way to computationally implement these algorithms.

One popular way to do this is to use ROS, which is an open-source platform that contains many tools for robotics projects, including state of the art implementations like the ones discussed here.

2.2.2.1 Hector-SLAM

The first ROS algorithm to be presented is the HectorSLAM [3]. HectorSLAM is an implementation that builds maps using 2D Occupancy Grids without the need of external odometry. This method, though, is not able to perform delayed map/trajectory optimisation. Figure 2.2.4 a system overview of HectorSLAM:

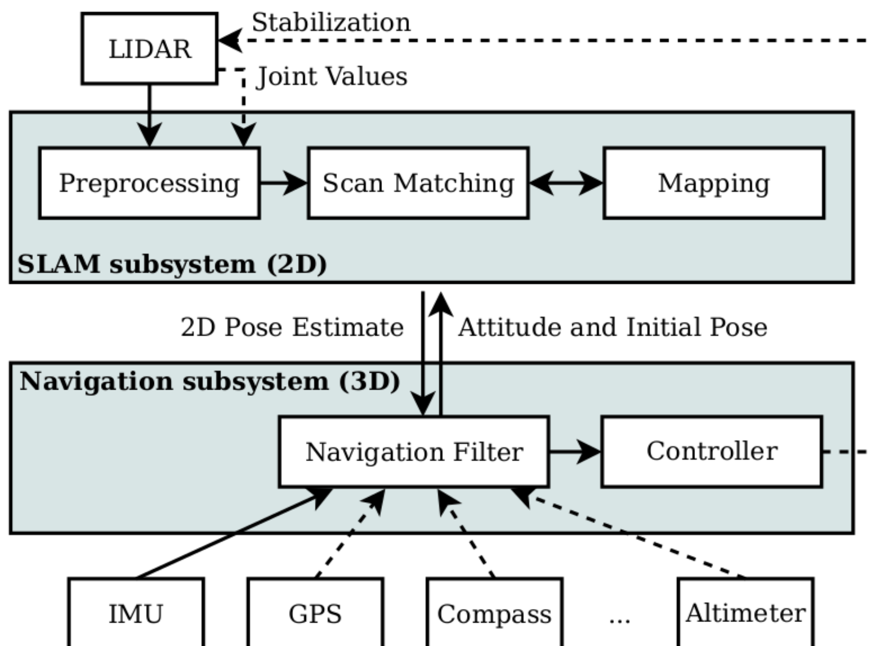


Figure 2.2.4: Overview of the mapping and navigation systems used in HectorSLAM [3].

As shown in Figure 2.2.4, HectorSLAM is composed by two subsystems that work together to provide its results. We have a navigation subsystem that takes input from the non-visual sensors in the robot and performs sensor fusion. Additionally, there is the SLAM subsystem that takes laser scans and performs scan-matching.

The navigation filter within the navigation subsystem takes the data from the IMU and other sensors (GPS, compass, altimeter,...) and estimates the 3D state of the system in the six degrees of freedom (DOF). On the other hand, scan-matching calculates the robot's pose in the 2D ground plane.

Scan matching in HectorSLAM works by aligning laser scans with the existing map by optimising the alignment of the beam endpoints with the map learnt so far [3]. This implicitly means the scans are aligned between them. The nature of the optimisation method used in HectorSLAM means that it might get stuck in local minima[5], possibly yielding false results but practical results show that this is improbable

To mitigate the local minima problem, HectorSLAM uses multiple maps with varied resolutions at all times. All maps are updated at every single step, starting with the lower resolution maps and ending in a finely defined map. In the transition between maps, the

[5]When a laser scan gets aligned in a local minimum, it means that it gets aligned in a plausible location which is not the true intended alignment for the scan (for instance, aligning a corner of a room in a laser scan with a different corner of the room in another laser scan).

pose estimate calculated for the previous map is used as the starting estimate for the next.

The six DOF estimate for the robot's pose is performed by an EKF. This is required because the algorithm is not linear as it estimates angles such as roll, pitch and yaw. In the case of the position and velocity estimates, the system integrates the measured accelerations provided by the IMU. The estimates provided by the EKF and the integration of the velocities are used as start estimates for scan-matching to improve the results.

2.2.2.2 Gmapping

Gmapping is a SLAM approach based on Rao-Blackwellised particle filters developed by Grisetti *et al.* [4]. Rao-Blackwellisation is a technique which allows us to factorise the joint posterior distribution of the map m and the trajectory $x_{1:t} = x_1, ..., x_t$ given laser scan observations $z_{1:t} = z_1, ..., z_t$ and the odometry measurements $u_{1:t-1} = u_1, ..., u_{t-1}$ in the following way:

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(m|x_{1:t}, z_{1:t}) \cdot p(x_{1:t}|z_{1:t}, u_{1:t-1}) \quad (1)$$

Using this factorisation, we can estimate the trajectory of our robot before building the map given that trajectory. More information on Rao-Blackwellisation can be found in Murphy *et al.* [14]. The map is usually computed through a process of "mapping with known poses" and the trajectory is evaluated through a particle filter that uses a probabilistic odometry motion model as the proposal distribution and that resamples at every single step. Only using the odometry as a proposal distribution is proved to not get the best results and resampling whenever we get a new observation is computationally inefficient.

In order to limit the computational effort and improve the quality of the results in comparison to the classical Particle Filter, Gmapping uses an improved proposal distribution and applies an adaptive resampling technique. The improved proposal distribution of Gmapping not only considers the odometry information but also the current observation, using a scan-matcher to determine the meaningful area where particles should be sampled. For example, in an open space, the probability distribution of the particles should have a very different shape than in a open corridor that, in turn, should have a different shape than the distribution in a dead end corridor. Whenever the scan-matching process fails, the algorithm reverts to using odometry information only, until it can restart using both types of data. Given this, odometry information is mandatory in order to use Gmapping. A comparison between using the improved proposal distribution in different scenarios and the odometry based variant is pictured in Figure 2.2.5.

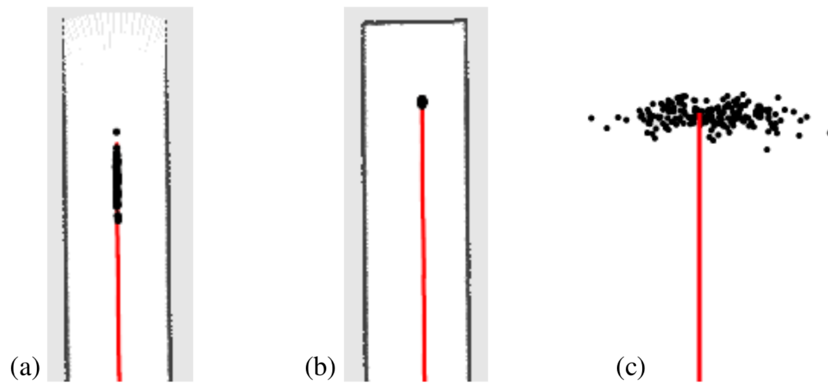


Figure 2.2.5: Difference in particle distributions on different environments using the improved proposal distribution [4].

In Figure 2.2.5(a) the particles are transversely well defined but less so longitudinally. This is due to the open nature of the corridor. In Figure 2.2.5(b), a dead end corridor, the particles are much more focused. In Figure 2.2.5(c) the particles are very scattered because they are in an open space. In this case, only the odometry motion model is being used.

The second improvement Gmapping introduces to other Particle Filter approaches is the adaptive resampling method developed by Grisetti *et al.* [4]. Resampling is an extremely important step because it removes particles with low weights, which poorly describe the environment, allowing for the generation of new particles. This is key because the algorithm works with a finite number of particles at all times. However, some unintended consequences can appear if we resample at every single step. One of these phenomena is called particle impoverishment or particle depletion where the resampling step removes good particles from our particle set. In order to mitigate this effect, Grisetti *et al.* [4] developed an adaptive resampling method using a measure of how disperse the particle weights are. This measure is called the effective sample size (N_{eff}). Resampling is performed when the value of N_{eff} drops below the value of $N/2$, which represents half of the number of particles used. The authors found that this reduces the number of resampling operations and the number of good particles dropped.

2.2.2.3 RTAB-Map

RTAB-Map is a pose graph based SLAM implementation developed by Labbé and Michaud [15]. This algorithm requires odometry as an input and it is able to output both 2D and 3D maps. It also offers the capability to perform RGB-D image based odometry, i. e., calculate odometry based on the colour and depth data provided by the sensor. This

is useful for applications with no other form of external odometry.

As a graph-based SLAM algorithm, RTAB-Map has the ability to perform pose graph optimisation. The optimisation is based on detected loop closures and it is a very powerful tool to correct any drift introduced on the map and trajectory calculated by the system.

RTAB-Map also provides us with the ability to perform localisation in previous drawn maps as a separate feature.

2.2.2.4 Algorithm Comparison

Table 2.2.1 compares some defining characteristics between the previously mentioned algorithms.

Algorithm	Type of Algorithm	Needs External Odometry?	Performs Map/Trajectory Optimization/Correction?	Map Output
HectorSLAM	EKF/ Scan-matching	No	No	2D Occupancy Grid
Gmapping	Particle Filter	Yes	Yes, by choosing the most optimal particles	2D Occupancy Grid
RTAB-Map	Graph-based SLAM	Yes	Yes, by performing loop closure	2D Occupancy Grid/ 3D Pointcloud Map

Table 2.2.1: Comparison between the discussed SLAM implementations available in ROS.

2.2.3 Pathfinding

Marder-Eppstein *et al.* [16] developed an open-source navigation solution for mobile robots. This is available in ROS as the Navigation Stack, as described in Section 3.3.2.5. This implementation uses sensor, map and odometry data and a navigation goal set by the user and outputs velocity commands so as to control the robot.

In order to navigate, this system builds a three-dimensional voxel grid to describe the world around the robot. Much like occupancy grids, the voxel grid is divided into a number of different cells, each one with the possibility to have one of three states: free, occupied or unknown. On a 64-bit machine, Marder-Eppstein *et al.* [16], state that this voxel grid can have 4.1 cm vertical and 2.5 cm horizontal resolutions [16]. It is important to have a three-dimensional voxel grid because we need to not only take into account objects that sit on the floor but every object that may impact the robot's travel, considering robot height.

The navigation algorithm can be started with a previously constructed map, provided

by any SLAM algorithm, or without any initialisation. If an *a priori* map is not provided, the navigation system purely uses the current observations provided by the sensors and the robot's pose provided by wheel odometry and by possible sensor fusion with IMU. In this scenario, the algorithm will estimate a path towards the navigation goal according to all previously seen obstacles. This path is progressively adjusted according to new observations. If a map is provided, the system calculates an initial guess for the required trajectory using such map. As with the previous case, the trajectory is then recalculated if/when new obstacles are detected.

The position and size of the obstacles is collected on a cost map, which is a 2D map that is basically a flattened version of the voxel grid. In this map, each cell is attributed a cost, which is high if the cells are occupied and it decreases exponentially according to a user defined inflation[6] as the distance to the obstacles increases. High cost means that no part of the robot must be in contact with the corresponding cell, zero cost represents free path. The cost map is initialised with the previously built map, if provided.

In order to achieve obstacle avoidance the system needs to have the robot footprint defined, meaning its size and geometry as seen from above. The robot's footprint is defined by the user.

With the purpose of planning trajectories, the Navigation Stack has two planners at its disposal: the global planner and the local planner. The global planner is used to draw the full trajectory from the robot to the navigation goal, considering the robot's position and the obstacle data in the cost map. As a consequence of the underlying algorithm of the global planner and to keep the system efficient, the global planner assumes the robot is circular and does not consider the kinematics and dynamics of the robot. Because of that, the path may contain bends that are too sharp to be taken by the robot or might try to lead the robot too close to certain obstacles, which would cause a collision. This means that the robot might not be able to follow the proposed path. The local planner, on the other hand, takes the kinematics and dynamics of the robot into account, as well as the position of obstacles from the cost maps. The local planner's function is to make the robot follow the global planner's proposed path in a feasible way while avoiding collisions. The local planner's output is velocity commands.

[6]Inflation is a value that sets how much clearance relative to the obstacles is desired for the robot's path. If, for example, an inflation value of 20 cm is set, the navigation algorithm will trace trajectories which avoid obstacles by this distance.

Chapter III

Implementation

3.1 Hardware Description

We start the implementation chapter by describing the experimental apparatus. The main tool used in this work is the Intel RealSense D435i stereoscopic depth camera, pictured in Figure 3.1.1. Being a stereoscopic camera, the Intel RealSense D435i is equipped with two RGB cameras separated by a known distance and an IR blaster. The IR blaster helps matching common features between the left and the right cameras. It also contains an IMU, which is very helpful for sensor fusion. The D435i is easily connected to any computer through USB.

Figure 3.1.1: Intel RealSense D435i.

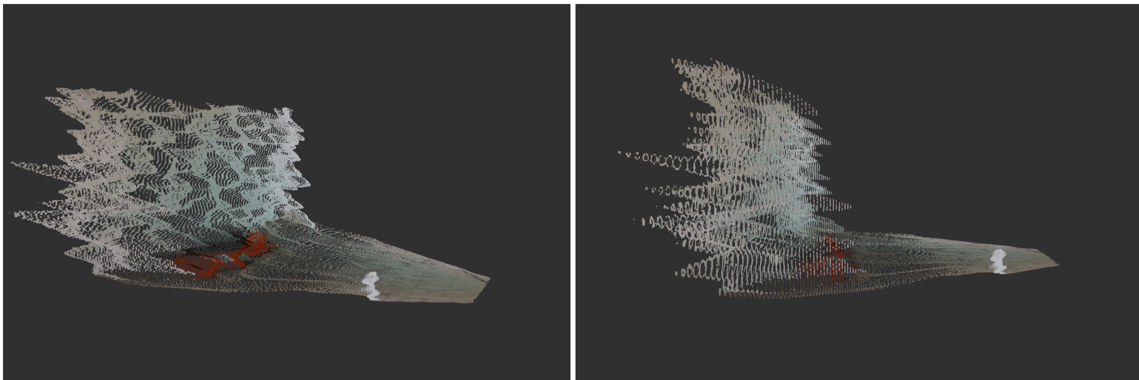
Table 3.1.1 compiles some distance measurements from the camera compared with the real values from a tape measure.

	Illuminated Room		Dark Room	
Real Distance (m)	Mean Distance Measured (m)	Standard Deviation (m)	Mean Distance Measured (m)	Standard Deviation (m)
0.5	0.5063	0.0007	0.5073	0.0004
2.5	2.6716	0.0174	2.6501	0.0405
5	5.3962	0.2211	5.2875	0.1613
7	7.3585	0.5594	7.4708	0.5503
9	10.9986	0.3732	10.9210	0.9759

Table 3.1.1: Depth measurements from the camera. Every measurement corresponds to the depth at the centre of the depth image.

As can be seen in Table 3.1.1, the depth measurements tend to deviate from the truth for larger distances. Other conclusion that can be obtained is that the measurements in an illuminated room are better than in the dark. Both results were expected considering that camera measurements are extremely dependent on the use of colour images. In contrast, in a LIDAR sensor, it is expected that the measurements in the dark are much closer to those in daylight (or with good illumination). Another important camera characteristic has been found, which relates to how the depth measurements vary across the depth image. Figure 3.1.2 shows a pointcloud of a flat wall and we can see that results are very fuzzy, with variable values of depth across the image. This fuzziness affects map quality.

(a) Colour image.



(b) Perspective view of the pointcloud.

(c) Side view of the pointcloud.

Figure 3.1.2: Colour image and pointclouds of a flat wall.

Throughout the development of this work it has been necessary to conduct several tests in order to assess the robustness of the results from the SLAM algorithms. These tests imply having a way to move the camera in a realistic fashion considering the objective of this work. Therefore, an ActiveTwo AGV previously developed by Active Space Technologies is used. Figure 3.1.3 presents the AGV used in this work.

Figure 3.1.3: Active Space developed ActiveTwo AGV.

This AGV uses a localisation algorithm provided by a separate company (Navitec), which specialises in the development of natural navigation algorithms. The robot itself is equipped with two LIDAR scanners that, in conjunction with the system from Navitec, allow the AGV to navigate autonomously.

The full apparatus involved taping the Intel RealSense D435i depth camera to the front of the ActiveTwo with a computer right behind it for data collection. The camera was placed 70 centimetres in front of the centre of the robot and aligned with the AGV. Figure 3.1.4 is a picture of the prototype in testing conditions.

Figure 3.1.4: AGV in test conditions.

This setup allows us to input trajectories into the Navitec localisation system and record data using the camera for offline analysis. These data are then processed by the SLAM methods developed throughout this thesis. More details on these tests will be discussed in Section 3.4.

3.2 First Attempts

At the start of this work, the chosen approach was to use MATLAB. MATLAB has very thorough and extensive documentation regarding many topics of computer science, and SLAM is no exception. During this time, some of the concepts regarding SLAM are studied and put into practice. This is done using the Intel RealSense MATLAB wrapper as well as associated functions. Some rudimentary SLAM systems are developed using scan-matching and IMU information but these do not show great results.

The first mapping algorithm is based on the full 3D world. The input for this algorithm is pointclouds and IMU information. The first step is to align successive pointclouds through a process called Pointcloud Registration. MATLAB offers some pointcloud registration functions in its Computer Vision Toolbox[7] that use some of the most popular algorithms like ICP (*pcregistericp* function) and NDT (*pcregisterndt* function). IMU information is also used in order to approximate the initial transform between pointclouds. At each step there

[7] `https://www.mathworks.com/help/vision/`

is a calculation to see which of the pointcloud registration algorithms produces the best result; then, the best transform is chosen. Finally, the two pointclouds are merged according to how the transform dictated.

The data are provided by rosbag files and this process is repeated until the rosbag file ends. The map is built incrementally, with every pointcloud hopefully being aligned and merged correctly with the other pointclouds.

Some additional changes are made to the algorithm in order to improve results. The first change is to not consider every single recorded pointcloud but to define an interval between processed pointclouds. This is a common practice in many SLAM methods, reducing computer allocated resources while still improving map definition. The other change is to segment out the area of the pointclouds that is not interesting for the registration (e.g., floor and ceiling). To do this, a part of the upper and lower volumes of the pointclouds is extracted according to a predefined percentage. In a final try, the pointclouds are flattened into two dimensions and a similar process was applied to try to build 2D maps. Unfortunately, these approaches did not give very good results. The algorithm was able to build short maps in very controlled conditions but eventually would fail.

At a certain point it became clear that continuing going down this path would be unproductive and achieving an acceptable level of robustness in the time available for the development of this work would be tough. Thus, it was decided that a new approach would be necessary for SLAM implementation, and ROS emerges as a robust option.

3.3 ROS-based Implementation

This section describes the stage of this work after abandoning MATLAB and discovering ROS. A brief description of ROS is presented along with the description of implementation details for the various algorithms used and their key parameters.

3.3.1 Brief description of ROS

As already mentioned, ROS [17] stands for Robot Operating System and is a popular robotics framework, which includes numerous state-of-the-art algorithms for a wide variety of robotic applications.

ROS allows for the execution of various pieces of software that can be written in C++ or Python through ROS nodes. These nodes can communicate with each other, allowing for the operation of a robot. Nodes are organised in ROS packages, that can contain one or multiple nodes. Packages, in turn, can be organised in Stacks.

Communication between nodes is done through messages that can contain any type of information for the operation of the robot like sensor readings, images, pointclouds, etc. The nodes publish these messages in ROS topics, which in turn can be subscribed by other nodes. This enables the communication between different scripts located in different nodes.

ROS uses some convention coordinate frames to define robot navigation. These are the *base_link*, *odom* and *map* frames [18]. The *base_link* frame is attached to the robot and moves with it, usually acting as the centre of mass of a robot. Some robot components position can be defined in relation to the *base_link* frame, like the position of a camera. The position of the robot in the *odom* frame is derived from odometry measurements. Within the *odom* frame, the pose of the robot is continuous but can drift over time. In the *map* frame, the pose of the robot is not continuous, meaning that the pose may jump over time as it is being corrected by SLAM algorithms. However, the pose of the robot in the *map* frame should have a low amount of drift.

This work relies on ROS Noetic on Ubuntu 20.04.

3.3.2 Implementation of SLAM algorithms

Enabling the execution of SLAM algorithms available in ROS is a matter of downloading the required package, with the desired nodes, and defining the required inputs and outputs in a launch file. The inner workings of each node may be treated as a black box, as long as the required communication between each node is properly defined.

Some aspects of SLAM related nodes are common to every implementation, for instance, the definition of the *world*, *odometry* and *base_link* frames. However, each node is different and, for every node, a specific set of input topics must be defined according to its needs. Additionally, it is common for every node to have a number of different parameters to fine tune in order to improve the results.

The requirements and parameters for each node are defined in the corresponding package's dedicated page on the ROS website.

3.3.2.1 HectorSLAM

HectorSLAM[8] uses laser scans to perform SLAM so, beyond the gmapping package, an extra package that converts the camera's pointclouds to laser scans is also needed. This package is simply called *pointcloud_to_laserscan*[9].

[8] `http://wiki.ros.org/hector_slam`

[9] `http://wiki.ros.org/pointcloud_to_laserscan`

The *pointcloud_to_laserscan_node* has a number of different parameters to be fine tuned and its job is to convert a pointcloud to a laser scan by slicing a volume of the pointcloud and flattening it along its height, simulating the output from a laser sensor by returning a two dimensional set of points. The thickness of the pointcloud volume used is defined by the user by setting the *min_height*, *max_height*, *range_min* and *range_max* parameters. In the range parameters, it is necessary to take into account the minimum and maximum range of the camera.

HectorSLAM's *hector_mapping* node subscribes to the *scan* topic provided by the previous package and returns an occupancy grid map in the *map* topic. It also requires the definition of a transform between the frame attached to the laser scans and the *base_frame* with the difference in position between these two frames. This is provided by the *static_transform_publisher* node from the *tf2_ros*[10] package. To improve the map, it is possible to fine tune the *map_update_distance_thresh* and *map_update_angle_thresh* parameters, which set linear and angular thresholds for the map to be updated, in metres and radians, respectively.

Figure 3.3.1 displays a flowchart with the most important topics, nodes and parameters for our HectorSlam implementation. The hexagon represents the depth camera, boxes represent the ROS nodes, links and ellipses represent ROS topics. A similar figure code is used for the flow charts corresponding to the other implementations.

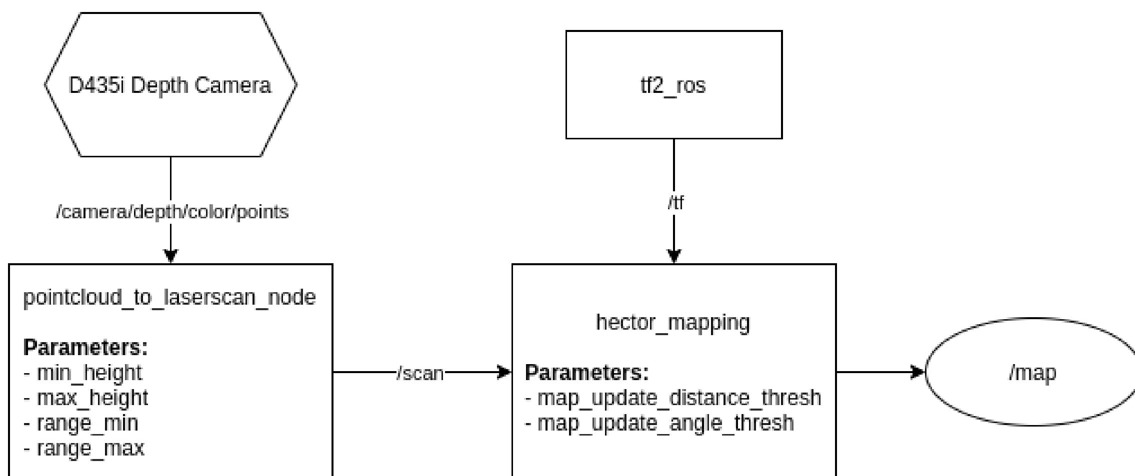


Figure 3.3.1: Flow chart for our HectorSLAM implementation.

3.3.2.2 IMU fused RGB-D Odometry

The next implementations of SLAM require external odometry so it is important to find a robust way of providing odometry with the capabilities provided by the camera.

[10] http://wiki.ros.org/tf2_ros

An interesting way to generate odometry from a depth camera is to use the successive RGB-D images provided by the camera. This is called RGB-D Odometry.

For this purpose, the RTAB-Map[11] package offers the *rgbd_odometry*[12] node that can perform just that. This node subscribes to the depth and colour image topics and the RGB camera info topic from the camera and outputs odometry messages containing the robot's position and orientation. While using this node, it is recommended to set the depth image topic as the D435i provided *aligned_depth_to_color* topic or else the localisation of features in the colour image will be different from the colour image. This is due to the different fields of view of the two images. It is also necessary to describe the position of the *camera_link* frame in comparison to the *base_link* frame. This is done through a static transform publisher node from the *tf2_ros* package. The AGV has a length of 150 cm so, in the tests run as a part of this work, the camera was located around 70 cm in front of the robot's centre and centred along the robot's width. This is the transform defined in the node.

This method provides reliable results by itself but, knowing that the D435i is equipped with an IMU, there is the possibility to perform sensor fusion in order to further improve the accuracy of the generated odometry. The work developed by Moore and Stouch [19] is available as the *robot_localization*[13] ROS package. This package offers sensor fusion nodes for robot state estimation based on the Extended Kalman Filter and the Unscented Kalman Filter. It has the ability to fuse data from an unlimited number of sensors of many different types. In this work, the UKF localisation node was chosen.

For this implementation, it is needed to fuse the RGB-D Odometry with the IMU data, which is easily supported by the *robot_localization* package. Thus, all that is needed to do is to provide the *map*, *odom* and *base_link* frames and set the input topics as the output topic names from the RGB-D odometry and the camera's IMU data. The example launch file provided in the package has templates for the configurations of every type of sensory data it might be needed to fuse so the process is straightforward.

Another important aspect available in the *robot_localization* package is the ability to flatten the data into two dimensions using the two_d_mode parameter, which can eliminate some unwanted drift in height caused by the RGB-D odometry uncertainty. All tests were made in horizontal environments so any movement in the *z* direction is undesirable.

[11] `http://wiki.ros.org/rtabmap`
[12] `http://wiki.ros.org/rtabmap_ros#rgbd_odometry`
[13] `http://wiki.ros.org/robot_localization`

3.3.2.3 Gmapping

Similarly to HectorSLAM, Gmapping[14] also takes laser scans as input, so the same package (*pointcloud_to_laserscan*) is used to convert the pointcloud data from the depth camera.

Within the *slam_gmapping* node from the Gmapping package, the main parameters to improve the quality of the maps are the *maxUrange* parameter, the *iterations* parameter and the *linearupdate*, *angularupdate* and *temporalupdate* parameters. The *maxUrange* parameter defines the maximum usable range of the scans, the *iterations* parameter sets the number of iterations of the scanmatcher. The *linearupdate*, *angularupdate* and *temporalupdate* parameters define how often should the map update according to linear or angular movements or if a certain period of time has elapsed from the last processed scan. Each of these parameters must be fine tuned according to the nature of the robot's surrounding and respective expected trajectory. For example, if the trajectory contains a large number of rotations, lowering the *angularupdate* value is key.

As mentioned, Gmapping uses the previously described RGB-D odometry fused with IMU as its odometry source and publishes an occupancy grid map in the *map* topic, akin to HectorSLAM.

Figure 3.3.2 shows a flowchart for our Gmapping implementation.

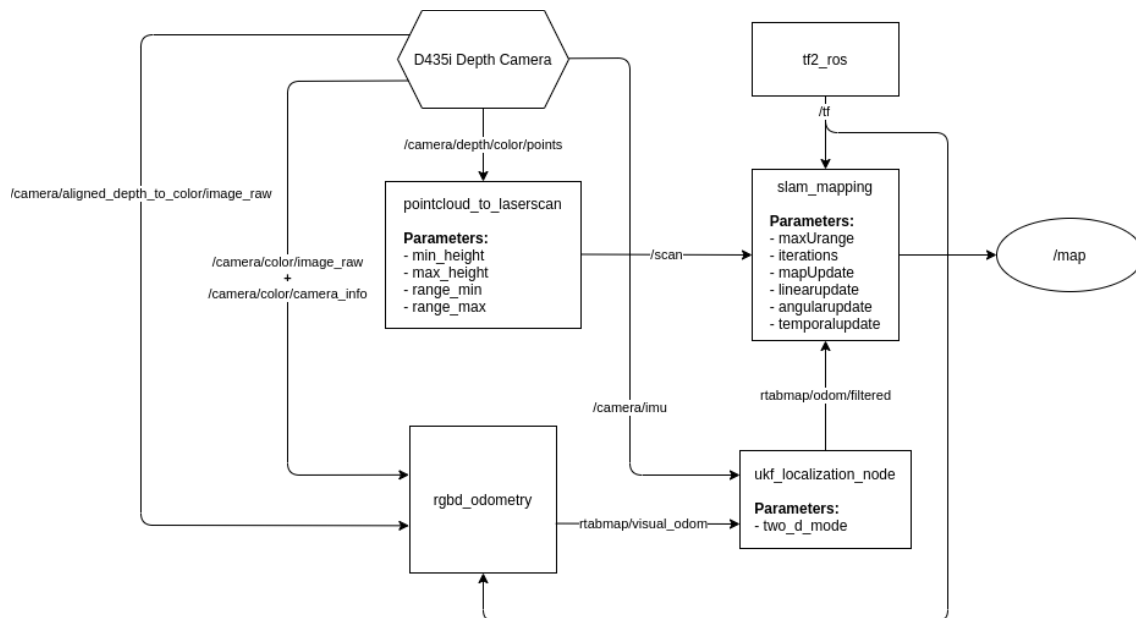


Figure 3.3.2: Flow chart for our Gmapping implementation.

[14] `http://wiki.ros.org/gmapping`

3.3.2.4 RTAB-Map

The setting up of RTAB-Map is very similar to the algorithms already discussed and a large part of the procedure is already made by implementing the all important IMU fused RGB-D odometry. To add this odometry to RTAB-Map it is a matter of defining the *odom_topic* as the output topic from the *robot_localization* package.

The additional required topics are the camera's image related topics: *rgb_topic* to enable loop-closure pose graph optimisation, the *depth_topic* and the camera info topics.

RTAB-MAP outputs the map in the *grid_map* topic and the trajectory in the *mapPath* topic.

Figure 3.3.3 shows a flowchart for our Gmapping implementation.

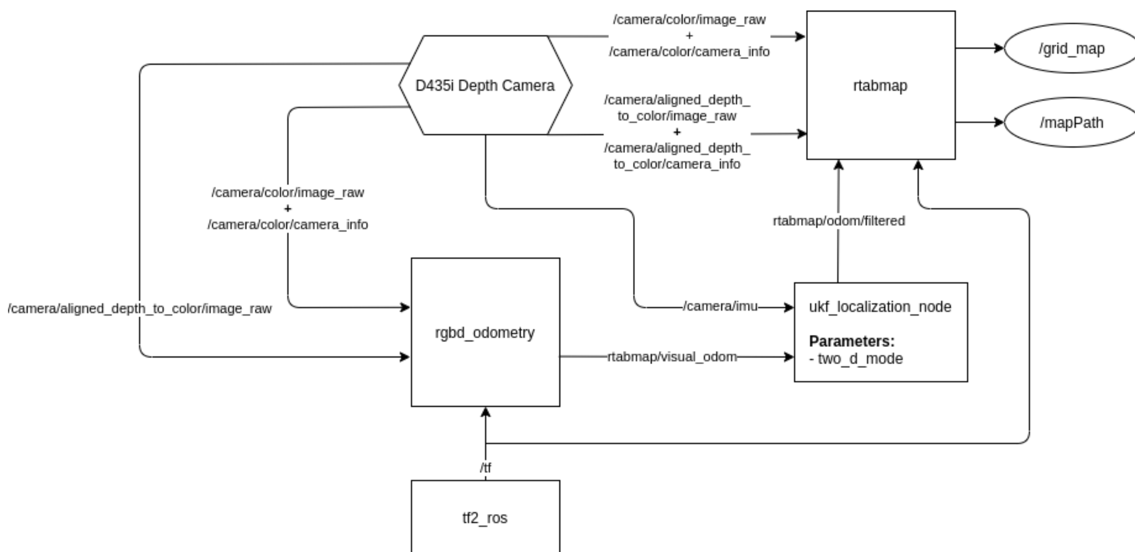


Figure 3.3.3: Flow chart for our RTAB-Map implementation.

3.3.2.5 Navigation Stack

Once the system has the capability to map an environment and localise itself, the next step for autonomous navigation is to implement pathfinding and trajectory planning. For this, the Navigation Stack[15] has been implemented. The Navigation Stack contains multiple important packages to enable navigation which are implemented simultaneously, such as *move_base* and *amcl*. However, for simplicity, the set of all packages will be described as the Navigation Stack in this work.

The Navigation Stack requires three sets of parameters in order to work: the costmap parameters as well as local and global planner parameters. In the local planner para-

[15] http://wiki.ros.org/navigation

meters is where the dynamic characteristics of the robot are fed to the algorithm. These parameters set minimum and maximum velocities and accelerations in both linear and angular movements. It is also defined whether the robot is holonomic and both position and heading tolerances for the navigation goal to be achieved. The local planner parameters are crucial so that the algorithm knows how to follow the global planner's path according to the way the real robot moves. For the global planner parameters, the default configuration has been used.

The costmap parameters define parametrisation for two different maps: the global costmap, used to make a global plan towards the goal, and the smaller local costmap for local trajectory planning according to possible new obstacles.

Some parameters are common for both maps, such as the map topic, which should be the same as the map output topic from the SLAM algorithm that is being used (HectorSLAM, Gmapping or RTAB-Map). Other common parameters are the definition of the sensors that are used to detect obstacles. For each one of these, it is necessary to set the type of sensor data (laser scan or pointcloud), the corresponding ROS topic, the frame where the data is attached and whether the data is used for the addition of obstacles and/or for eliminating obstacles from the map. It is possible to have some sensors adding obstacles, some sensors removing obstacles or sensors that do both addition and removal.

There are some parameters that appear in the configurations of both maps but must have different values. These include map size and origin. The global costmap's size must accommodate the robot's full expected trajectory, so it is normally a large map. On the other hand, the local map is smaller, with dimensions within the range of the camera's data in order to describe the local surroundings. It is important to set the local map's origin as half its dimension along both axis and the *rolling_window* parameter as true so that the robot remains in the centre of the local costmap, as desired.

Other parameters that differ between local and global costmaps are the update and publish frequencies. For the global costmap these values do not need to be very high so they are kept low, with the map being updated at 5 Hz, saving computational effort. In the local costmap's case the opposite applies, because it is important that the map is updated and published at a high rate to account for possible moving obstacles. The local costmap update frequency is set at 30 Hz. A similar approach is applied to the transform tolerance parameter, which sets a maximum delay for an incoming transform to be valid.

In order to use the Navigation Stack, we can use ROS' own visualisation tool (Rviz[16]) to publish a 2D Navigation Goal topic in the map frame. Once the robot destination is set, the

[16] `http://wiki.ros.org/rviz`

Navigation Stack will estimate and publish the global trajectory towards the navigation goal and velocity commands dictated by the local planner guiding the robot along the planned path. The navigation algorithm can also be paused when a red traffic light is detected through the publishing of the *move_base/cancel* topic via a python script (more details about object detection in Section 3.3.3). This python script (”realsensetalker.py”), which also performs object detection, subscribes to the *move_base/goal* topic and saves it in a variable. When a green traffic light is detected, the navigation is resumed by publishing the saved goal in the *move_base/goal* topic.

Figure 3.3.4 displays a flowchart for our Navigation Stack Implementation, containing the most important packages, topics and the integration with the ”realsensetalker.py” python script (represented by a parallelogram).

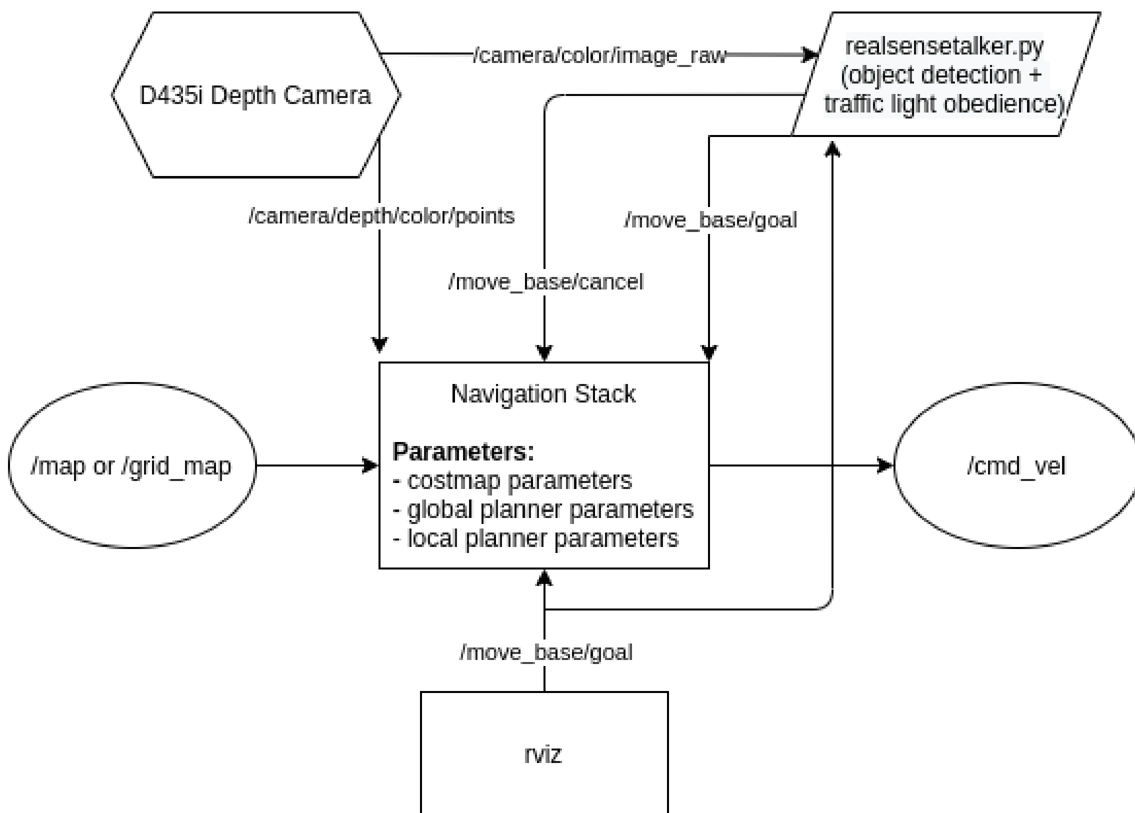


Figure 3.3.4: Flow chart for our Navigation Stack implementation with traffic light obedience.

3.3.3 Object Detection

Since the D435i depth camera is equipped with a colour sensor, there is the opportunity to add object detection algorithms to this work. Object detection is implemented with the objective of adding extra functionalities to the system, like the ability to stop and restart navigation in accordance to a traffic light, and improving the resulting maps by eliminating

people from the camera's data. People represent the most common moving obstacle for most robots in most applications so, by removing the false landmarks imposed in the map by passing humans, it is possible to improve scan-matching and the resulting map will be composed of more relevant features. The rationale behind this strategy is that lack of information ('hole') would be better than presence of false landmarks that can mislead the system in subsequent visits. When the robot revisits the "hole" and the person is not there anymore the map is completed with the right information about the room.

Two implementations of object detection have been evaluated in this work. The first was the TensorFlow Object Detection API[17] (Application Programming Interface). The Object Detection API is compatible with many different pre-trained object detection models, each with different performance characteristics and types of output. Some models prioritise the accuracy in the detection and others have an emphasis on detection speed. Some models output bounding boxes around the detected object, others output masks with the object's shape. The more accurate models tend to be slower and the faster models have a higher probability in returning false positive results.

The chosen model would have to be balanced in terms of performance, since it is important to run in real time but detection of false positives has to be minimal. Considering this, the SSD MobilenetV1 model trained over the COCO dataset was used. This model is able to detect 90 classes of objects, including people and traffic lights, as required.

The TensorFlow Object Detection API detects objects frame by frame, in a loop, returning a list of the detected classes, respective scores and pixel locations of the corners of the bounding box corresponding to each detection. The scores are a percentage metric which determines how certain the model is about the detection.

However, for this application, the detection of 90 different object types is not really necessary so the detections corresponding to undesired classes were removed from the output. Furthermore, a minimum score value of 50% was set to further reduce the detection of false positives.

Once the detections are available, the respective bounding box dimensions are used to extract a sub-image of the object, then, the centre of the object (centroid) is calculated and published as a ROS topic. A topic is published for people centres and another for traffic light centres. These topics were useful for the NavCam contest but are not used in this work, so they will not be discussed further.

For traffic lights, extra processing is carried out in the sub-images in order to detect which light is turned on. A range of HSV (Hue, Saturation, Value) values is defined for the three colours and a mask for each colour is applied to the image. The mask with the

[17] https://github.com/tensorflow/models/tree/master/research/object_detection

higher number of *true* values (corresponding to the number of pixels of a colour from the respective mask's range) determines whether the traffic light is green, yellow or red. For this application it has been important to deviate from the RGB standard into HSV because while red and green are primary colours in RGB, yellow is not. Thus, in order to set value ranges for yellow, it is necessary to choose values that mix the two primary colours, which represent the other important traffic light conditions. This may induce fails in detecting the right colour. Hence, by using HSV, which represents colours by Hue, Saturation and Value, not by mixing primary colours, the system is less prone to error. When the colour is detected, a ROS topic is published with the colour of the traffic light.

After using the last model, it became clear that for the purpose of removing people from the depth image it might be easier and more computationally efficient the use of an object detection algorithm that returned masks highlighting the whole detection instead of bounding boxes. Some alternative models that output masks were tested with the Object Detection API but it was not possible to get a high enough frame rate. Because of this, moving away from the Object Detection API became mandatory.

A new implementation by the name of Yolact[18], developed by Bolya *et al.* [20], was found and implemented. Yolact is a real time object detection algorithm that outputs masks. Another bonus point for Yolact is that it has a dedicated ROS wrapper [21] that facilitates communication with the rest of the ROS environment by subscribing to and publishing ROS topics. The chosen model to use with Yolact was Resnet50-FPN, which is the available model that provides the highest frame rate without a large penalty in the quality of the detections.

Further modifications were made to the Yolact algorithm in order to only consider the detection of people and to black out the background of the detection. The isolated detections are published in the */yolact_ros/visualization/* ROS topic. An additional Python script ("yolact.py") subscribes to the detections and the *aligned_depth_to_color* topic from the camera. Then, a mask is obtained from the image and it passes through an image eroding process to amplify the person's shape so as to make sure the whole body is deleted from the depth image. Finally, a logical "and" operation is performed between the mask and the depth image resulting in a filtered depth image without the detected people. In our tests, up to four people are detected and removed, as it is shown in Section 4.1.2, Figure 4.1.8, but the maximum number of people that Yolact can detect in a frame was not measured. The filtered depth image is published in a ROS topic to be used as input for the SLAM algorithms.

This filtered depth image can be directly inputted into RTAB-Map but the *camera_info*

[18] `https://github.com/dbolya/yolact`

topic to be used in RTAB-Map cannot be the topic provided directly by the camera. This is due to the fact that RTAB-Map needs the depth image topic and the corresponding camera info topic to be synchronised. To get around this problem, the "yolact.py" script also subscribes to the */camera/aligned_depth_to_color/camera_info* topic and republishes it at the same time as the filtered depth image, in the */filtered_camera_info* topic. For Gmapping, which takes laser scans as input, extra processing must be applied to the filtered depth image. The filtered depth image is passed through the *point_cloud_xyz* node from the *depth_image_proc* package, which converts the depth image into a pointcloud. Having a pointcloud, the rest of the process is equal to the already described Gmapping setup (Section 3.3.2.3). In this implementation, the filtered depth images are also fed into the RGB-D odometry when this filter is applied.

Figure 3.3.5 displays a flowchart with the most important topics for our implementation of Yolact in order to filter the depth images.

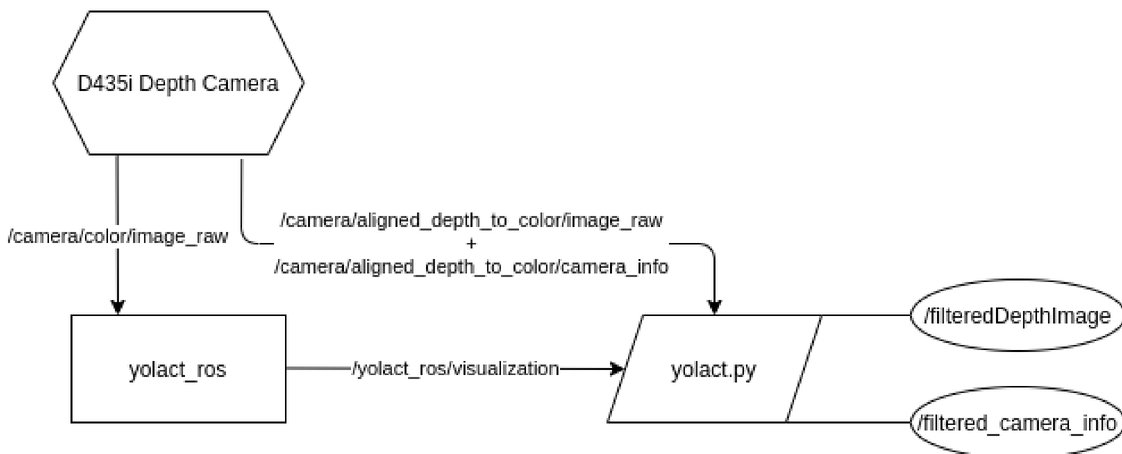


Figure 3.3.5: Flow chart for the implementation of Yolact.

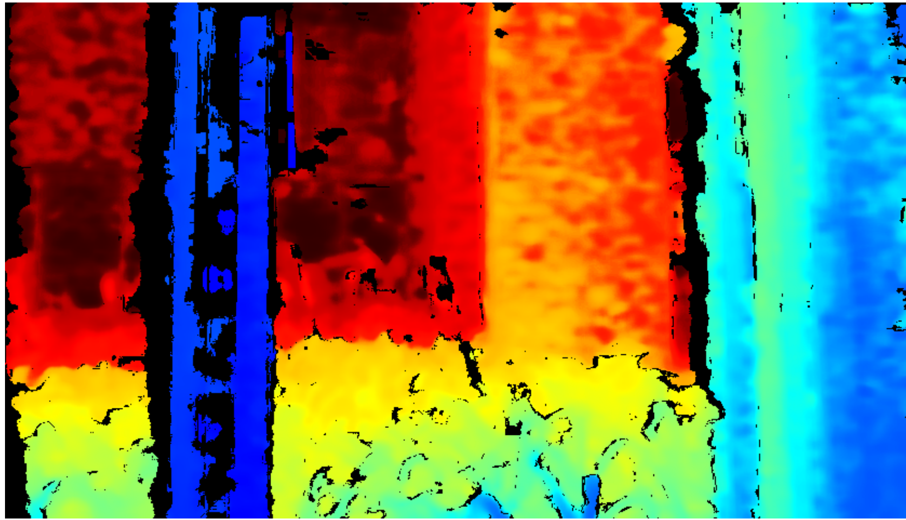
3.4 Tests Description

During the development of this work, three distinctive types of tests have been carried out: depth tests, SLAM tests and object detection tests.

First, it is important to evaluate the precision of the camera's depth measurements and how it behaved according to different types of obstacles. The depth tests involve measuring the depth returned from the camera's measurement and comparing it with a tape-measure. The measurement from the camera is obtained through a MATLAB code that returns a user defined number of depth measurements at a defined pixel location. At the end of the data collection, the script returns the average and the standard deviation of all values. The depth is measured in the centre pixel of the depth image and the results

are presented in Table 3.1.1.

The original plan had been to do this previous test both for opaque and for transparent and translucent objects, but it was immediately discovered that the camera cannot measure distances to transparent objects, namely windows. When a window is in front of the camera, the measured depth corresponds to objects behind the window, as if the window did not exist at all. This can be easily seen in Figure 3.4.1.



(a) Depth Image

(b) Colour Image

Figure 3.4.1: Depth and colour images from the D435i depth camera in front of a window.

The window in Figure 3.4.1 is completely invisible in the depth image. The depth measured for the centre of the image is of 7 metres which corresponds to the wall beyond the window, on the opposite side of the camera.

As the SLAM algorithms are implemented it becomes necessary to assess the level of accuracy of each algorithm, whether it is to fine tune the algorithm's parameters or to compare it to others.

The SLAM tests are performed with the previously shown prototype (Figure 3.1.4) and consist in programming a trajectory into the Navitec's localisation algorithm while recording data using the depth camera. The depth camera's data are recorded in the ROS Bag format, which is a type of file that records ROS topics and that can be played back in real time. These files are really useful in the way that they allow for virtual simulations that are indistinguishable to a real-life running of the robot and permit easy repeatability of the tests.

The next step is to find a test location. According to the results of the depth test, it is clear that the test would ideally have to be run in a not too large, well illuminated room, preferentially devoid of any transparent or translucent barriers like windows. The dimensions of the room should be big enough so that the robot can follow a significantly large trajectory but, in order to operate the camera in a high degree of accuracy, there should be obstacles closer than 10 m from the camera at all times. The terrain must be smooth enough not to induce a very high amount of drift into the IMU measurements. Another important point to some algorithms, specifically RTAB-Map, is the ability to perform loop-closure so the room's size should also allow the robot to perform a loop.

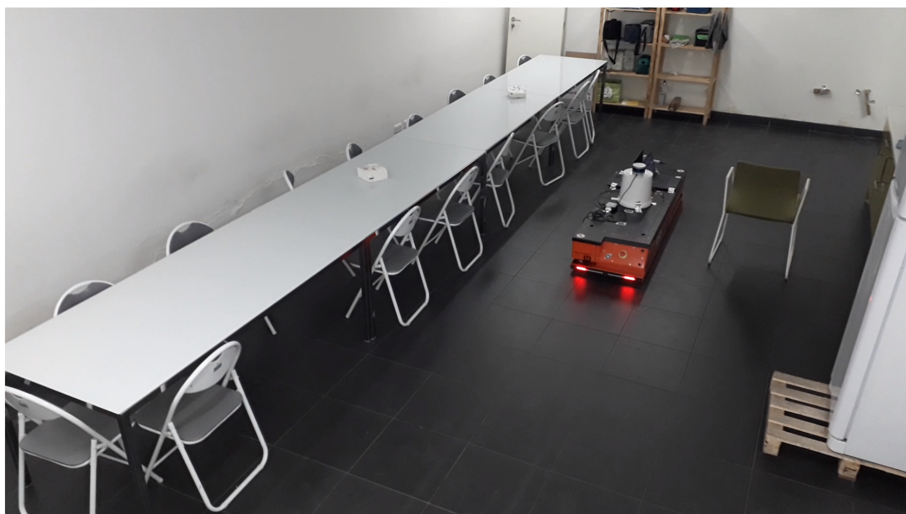
Considering these facts, it was concluded that the Active Space Technologies' basement is a good candidate to perform the tests. This is effectively a 17 m x 13 m usable area that meets most previous requirements. The only characteristic that might cause problems is the room's size versus the accuracy of the depth measurements. To mitigate this, white plastic buckets are added along the robot's trajectory in order to create additional landmarks. The basement has three extra adjacent rooms: a meeting room, a canteen and a laboratory, which are also used during testing.

Now that the test environment is set, the only thing missing is the test trajectories for the robot. The first tests are simple, with the robot driving around an oval trajectory alongside the walls of the rectangular basement. In the second phase, some tests are run with the robot performing an eight-shape trajectory around two of the basement's support pillars. In the final test, all previous tests are combined into a longer, more complex trajectory. This is done in order to test the robustness of the algorithms in a higher level and execute a more complex type of loop-closure. In this test, the robot drives around the oval lap of the basement, then entering the meeting room heading towards the canteen and through the laboratory before re-joining the oval trajectory from the opposite side of the basement. After this, the robot follows the rest of the oval up to the starting point,

where it performed an eight-shape trajectory again, similar to the second test. In this trajectory there are four possible loop-closures: one at the end of the oval lap, another when it rejoins the oval after exiting the laboratory and, finally, two more at the starting point, before and after the eight-shape trajectory.

The first tests are recorded without the presence of moving objects or people but, later, with the introduction of object detection algorithms, some tests are performed with stationary and moving people. People are introduced in the robot's view only when the AGV rejoins the oval lap after exiting the lab. This is done so that an accurate map of the full space could be built before testing the system's behaviour in the presence of people. In Figure 3.4.2, some photographs of the final test are presented.

(a) Basement



(b) Canteen

Figure 3.4.2: Pictures of ongoing SLAM tests.

When object detection algorithms are added, some dedicated tests are also performed. In these tests, the maximum distance that a person could be detected was measured. This is done by simply running the script and checking if the algorithm detects a person at various distances, as shown in Figure 3.4.3(a). It is also discovered that the algorithm does not need the full image of a person's body in order to detect them, Figure 3.4.3(b) shows a person being detected only through the image of their hand.

(a) Long distance detection of a person.

(b) Detection of a person through a part of the body.

Figure 3.4.3: Object Detection API tests.

For the detection of traffic lights, a traffic light image on a mobile phone was shown to the camera and it was tested if the navigation system stopped and restarted its goal according to the traffic light's colour.

For the integration of the object detection algorithms with SLAM, additional SLAM tests have been made for the same trajectories but with static and moving people appearing in sight of the robot at random places, as already mentioned.

Chapter IV

Experimental Results

4.1 Mapping

A crucial part of a SLAM system is the ability to build good maps of the robot's environment, so the resulting maps from each algorithm in static and dynamic environments are provided in this section, as well as the computed resulting trajectories. The results of adding object detection methods to SLAM algorithms are presented in the dynamic environments section.

In order to make the comparison between the maps and the real world, a map made by a Velodyne high precision laser scanner is provided (Figure 4.1.1) as well as some panoramic photographs of the four rooms (Figure 4.1.2). The Velodyne map was done in the scope of another project in Active Space Technologies. The robot did not explore the area to the left of the red dashed line in Figure 4.1.1 so the maps in our results correspond to the area to the right-hand side of the line. During testing, white plastic buckets have been spread across the basement in order to increase the number of features, however, this is not pictured in Figure 4.1.2(a) but can be seen in Figure 3.4.2(a).

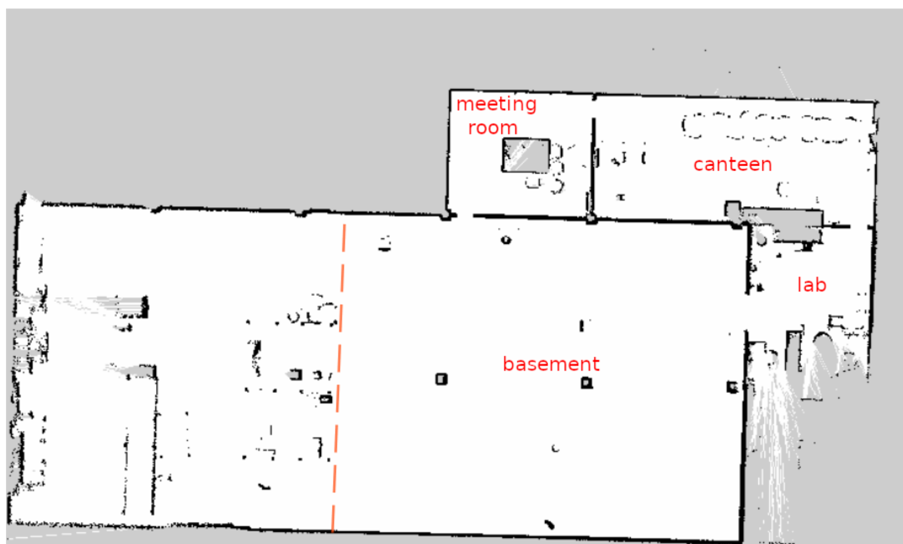


Figure 4.1.1: Map built using a high precision Velodyne Puck VLP-1614 laser scanner.

(a) Basement

(b) Meeting Room

(c) Canteen

(d) Lab

Figure 4.1.2: Pictures of the four rooms of the test site.

4.1.1 Static Environments

4.1.1.1 HectorSLAM

HectorSLAM does not use odometry or perform loop closure map/trajectory optimisation and this was found to be a defining factor in the quality of results. Nevertheless, due to the good quality of HectorSLAM's scan-matcher, it is common that good results are still able to be obtained with a wide variety of laser scanners available in the market. However, the majority of the laser scanners commonly used in SLAM applications have a 360º horizontal field of view and this is not the case with the Intel RealSense D435i, with a mere 87º of horizontal field of view. If a 360º scanner had been used, the amount of features in every frame would be much higher. This would not only provide a larger number of reference points available to match the pointclouds but it would also protect against possible occlusion of some features.

As a result, maps tend to quickly diverge with our setup, and, without the ability to perform loop closure optimisation, the divergence is never corrected and the results get progressively worse. Figure 4.1.3 presents an attempt to map our test site up to the canteen.

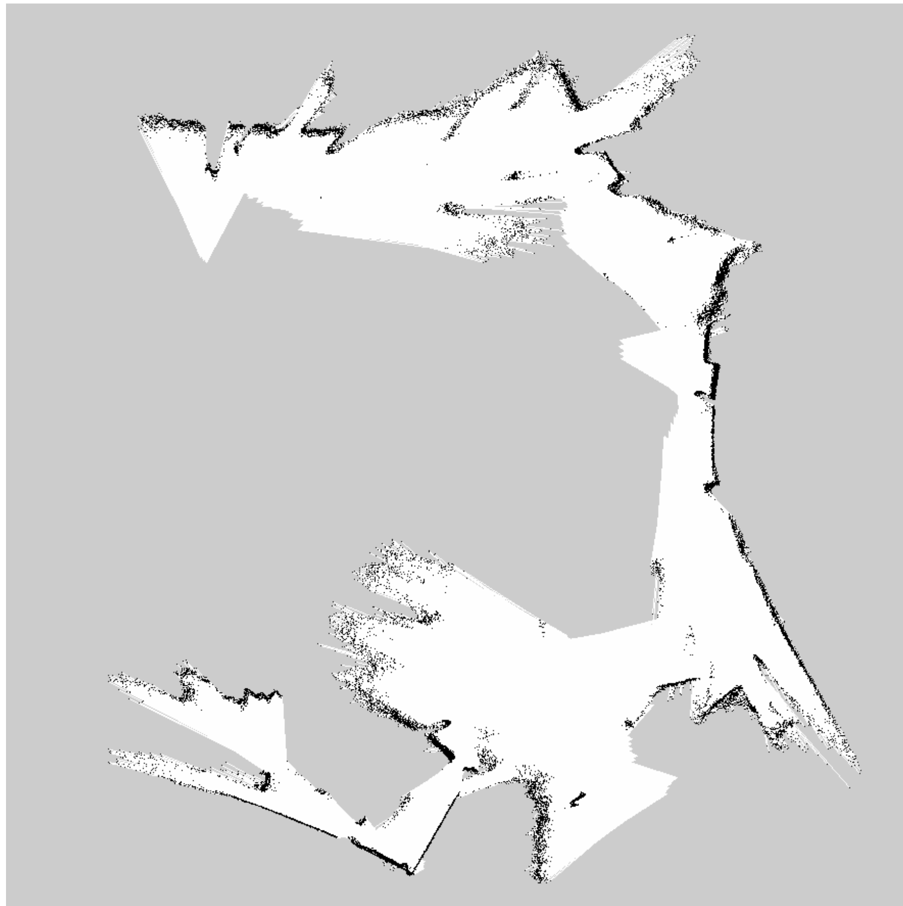


Figure 4.1.3: Static environment map from HectorSLAM.

The grey area represents unexplored space, whereas the dark points represent occupied space and obstacles, and white areas are free space, as described in Figure 2.2.1.

It is clear that the algorithm diverged, mainly during rotations, leading to poor mapping. The algorithm diverged even though the angular speed is kept low in every test since it has been found that if the cornering speed is too high, the probability of failure increases. This is valid for HectorSLAM and the two subsequent algorithms.

4.1.1.2 Gmapping

Gmapping with IMU fused RGB-D odometry generally provides good results. Nonetheless, due to Gmapping's probabilistic behaviour, it may occasionally diverge, resulting in some inaccuracies in the maps. This derives from the Particle Filter every so often choosing particles that wrongly describe the robot surroundings.

Figure 4.1.4 shows a fairly accurate map of the test site, albeit with some divergence in the lower wall and in the path that goes along the meeting room, canteen and lab.

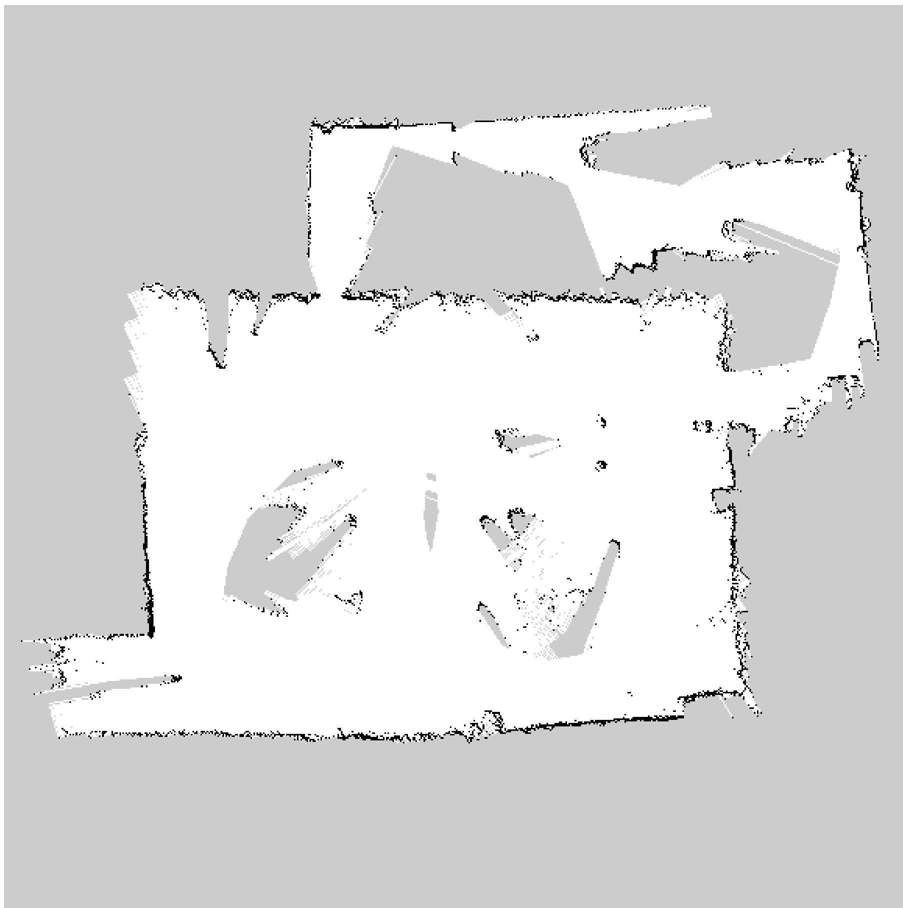
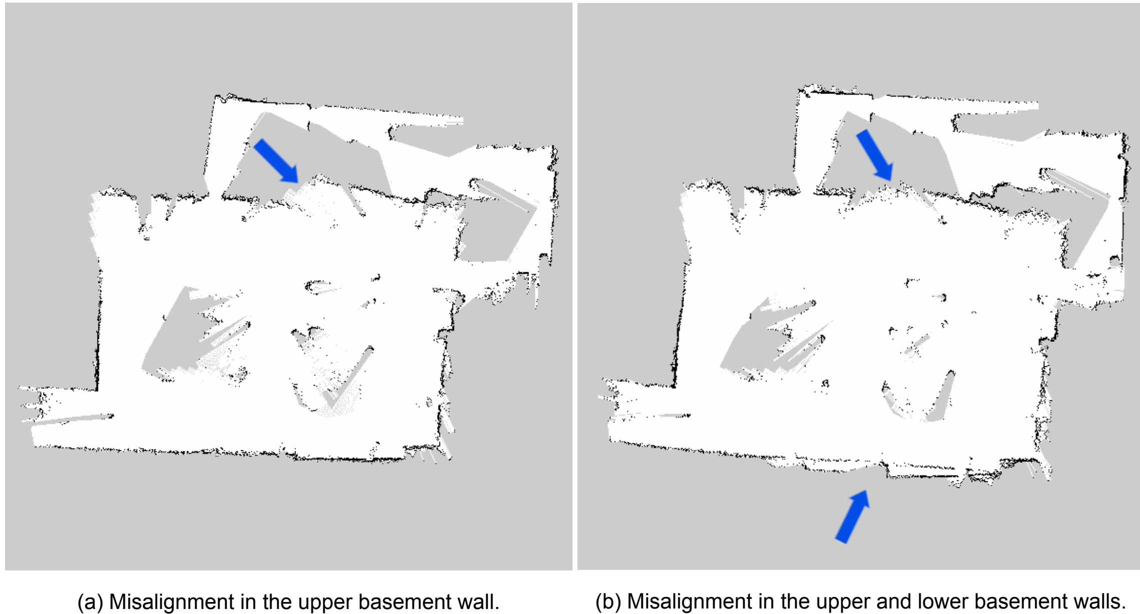


Figure 4.1.4: Static environment map from Gmapping.

The obstacles in the centre of the map are originated from the white plastic buckets that are displaced along the basement in order to increase the number of features and from the basement's support pillars. There are some islands of unexplored space in the centre of the map which result from the camera never being pointed towards those directions.

An example of the Gmapping implementation from this work can be seen in a dedicated video [22]. Figure 4.1.5 shows some examples of typical Gmapping failures (indicated by blue arrows) due to its probabilistic behaviour.



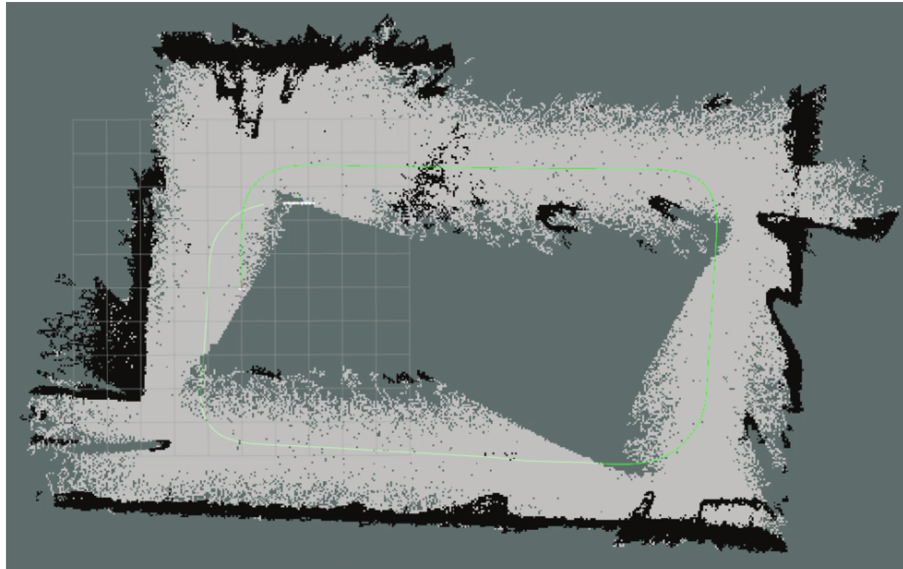
(a) Misalignment in the upper basement wall.

(b) Misalignment in the upper and lower basement walls.

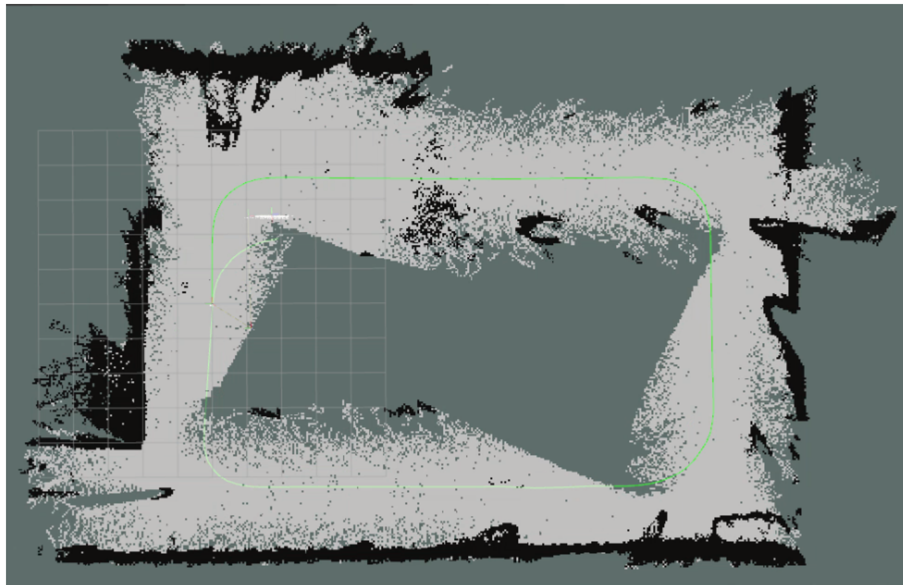
Figure 4.1.5: Typical Gmapping failures.

4.1.1.3 RTAB-Map

RTAB-Map has been revealed as the most reliable and precise algorithm to use with this work's setup. The maps are accurate and there is not a very high variability between different runnings of the same test. Most divergences that happen during the map building process are corrected by loop closure optimisation. Figure 4.1.6 illustrates an example of a loop closure pose graph optimisation in RTAB-Map, we can see that the map had diverged, with the left ”wall” being shorter that in reality. When the loop closure happened, the map was corrected and the room got noticeably squarer. An alignment of the trajectory (green line) is also noticeable.



(a) Map before optimisation.

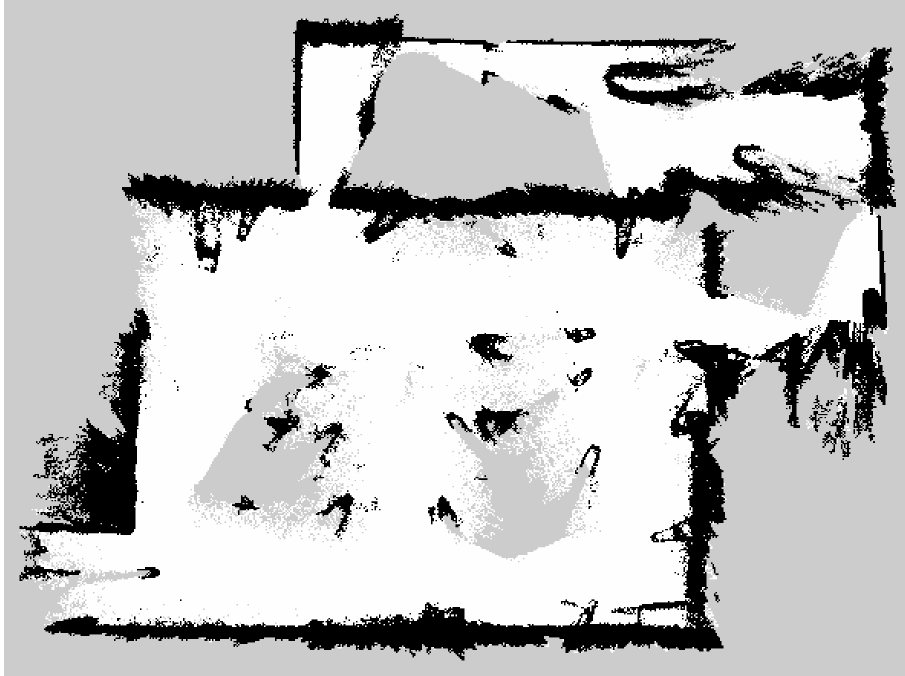


(b) Map after optimisation.

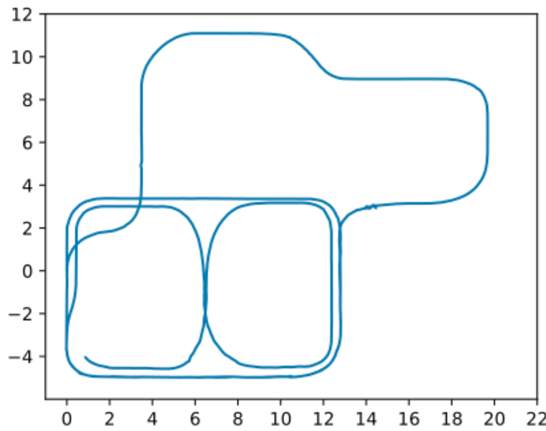
Figure 4.1.6: Example of a loop closure pose graph optimisation in RTAB-Map.

The only drawback in RTAB-Map is a larger wall thickness, which results from the wavy depth measurement, as shown in figure 3.1.2.

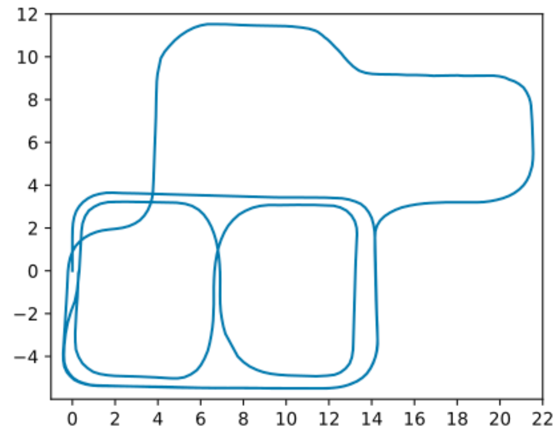
In addition to the maps, RTAB-Map also provides us with the ability to export the full *a posteriori* trajectory, which can be compared with the trajectory from the Navitec localisation system. The map and trajectories are presented in Figure 4.1.7.



(a) Resultant Map from RTAB-Map.



(b) Trajectory from the Navitec localisation system [19].



(c) Trajectory from RTAB-Map.

Figure 4.1.7: Static environment results from RTAB-Map.

From Figure 4.1.7 it is concluded that the map is very good as well as the trajectory, albeit showing some minimal drift. The high fidelity in mapping the test site along with low linear and angular drift and very high reliability makes RTAB-Map the most robust of the algorithms investigated.

A video showing how RTAB-Map is being implemented can be found here [23]. The video description contains timestamps for some loop closure map optimisation events, including the one pictured in Figure 4.1.6.

[19]The Navitec localisation system did not record the last part of the trajectory but the path finished in a similar manner to Figure 4.1.11(e) at around the (0.5,0) coordinates.

4.1.1.4 Comparison between Depth Camera and LIDAR

This work has been carried out in coordination with a similar project that uses the Intel RealSense L515 LIDAR scanner instead of the D435i camera. In some cases, tests have combined both technologies, which allows for a straightforward comparison between depth cameras and LIDARs.

In general, the depth camera measurements are less accurate and present some significant variation across the depth image as shown in Figure 3.1.2. This means objects in the map assembled with the camera are fuzzier and features like walls may have their thickness inflated (Figure 4.1.7a) and flatness reduced. This does not happen with the LIDAR measurements because they tend to be significantly more accurate, meaning the measured distances are closer to the true distance to objects, error does not increase much with distance and accuracy extends across the whole image, with the LIDAR not producing the wavy wall effect. The more accurate measurements from the LIDAR yield a lower failure rate and divergence of the algorithms as well. The LIDAR's measurements in the dark are very similar to the measurements in an illuminated room, which is not the case with the depth camera.

That being said, the Intel RealSense L515 LIDAR also presented some shortcomings. For example, the LIDAR does not work well with direct sunlight, which is not a problem for the camera. The Intel RealSense L515 LIDAR also presents an unidentified source of noise at certain spots in our test room. This noise appears as false detections, resulting in false obstacles being added to the map. This situation introduces uncertainty to mapping and navigation, degrading the reliability of the process, e.g., avoiding non-existent obstacles. Various tests have been made in order to try to identify the source of this problem but, ultimately, the reason was not found. Detailed information can be found in the other work [24].

Considering these facts, we conclude that generally, the LIDAR will return better results due to its higher accuracy in measurements, however, if the scene is well illuminated by sunlight or if the noise from the unidentified source is detected in large amounts, the camera will be the better choice. The LIDAR would also be the sensor of choice for mapping in the dark. Nevertheless, the camera also returns good SLAM results in most environments.

4.1.2 Dynamic Environments

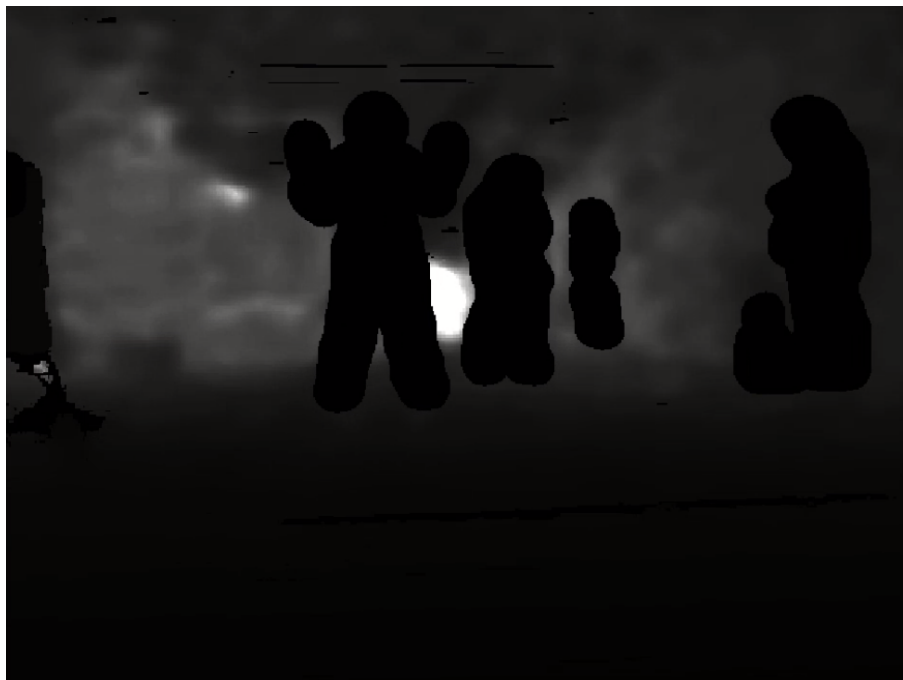
Results in Section 4.1.1 highlight the performance of algorithms without any type of moving objects or people in the room. However, dynamic environments are crucial for a robust implementation. Hence, the tests presented in this section are related to the

performance of SLAM in dynamic environments and the impact of object detection in the final results. Only Gmapping and RTAB-Map are used since HectorSLAM fails in static environments.

Figure 4.1.8 shows the effect of applying the Yolact object detection algorithm for people removal from the depth image. All the four present people are detected and removed (the black colour represents no depth information). A part of the bucket stack on the right is also removed due to the bucket being close to a person and wrongly classified as such. After processing, the filtered depth image is published at around 17 FPS in the computer used in this work.

(a) Colour image.

(b) Original depth image.



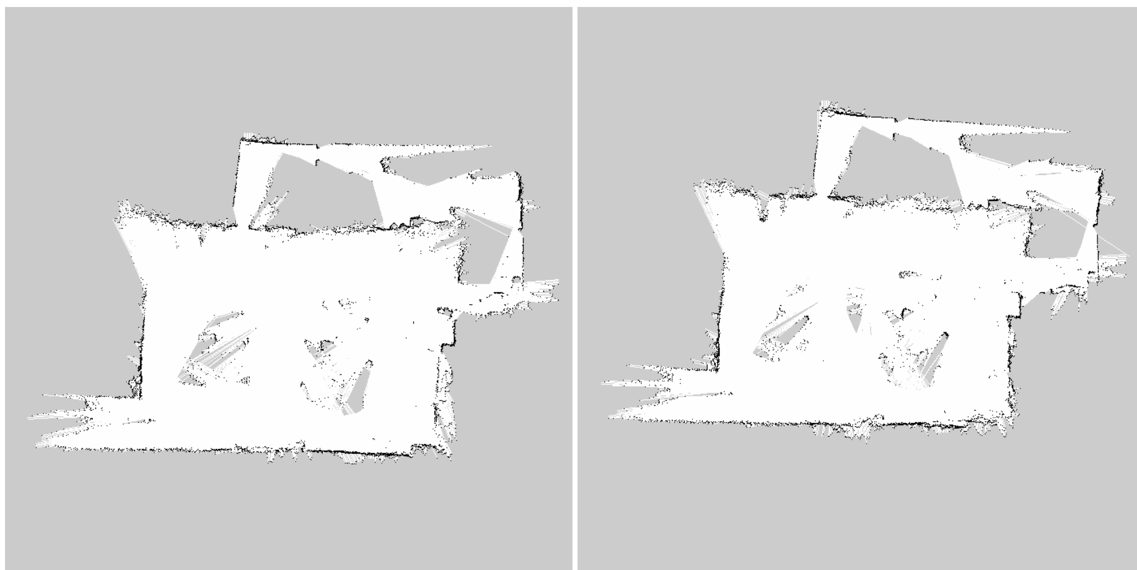
(c) People removed from depth image.

Figure 4.1.8: Removal of people from depth image using Yolact.

For each algorithm, mapping attempts are carried out with and without removing people through the object detection algorithm. This is performed for the full lap of our test site and also for the eight-shaped trajectory (the only part of the trajectory that includes moving people). The next sections present those results.

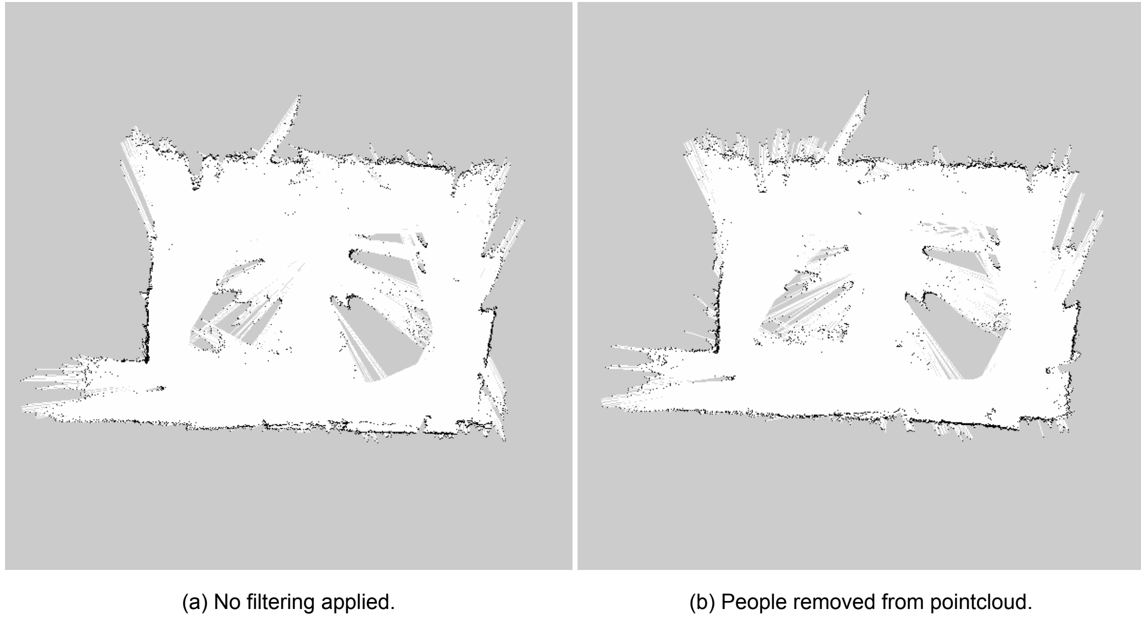
4.1.2.1 Gmapping

It has been confirmed that the presence of moving people in the room increases the probability of map inaccuracy or total failure. However, Gmapping is able to still draw a good representation of the basement with the presence of moving features and without adding them to the final map. Filtering the pointclouds shows no appreciable difference in those results, meaning that Gmapping is robust enough for dynamic environments by itself. Discrepancies among maps are more related to Gmapping's probabilistic behaviour. Figure 4.1.9 presents maps built with and without filtering for the full lap and Figure 4.1.10 shows the results for the eight-shaped trajectory.



(a) No filtering applied.　　(b) People removed from pointcloud.

Figure 4.1.9: Gmapping dynamic environment results for the full lap.

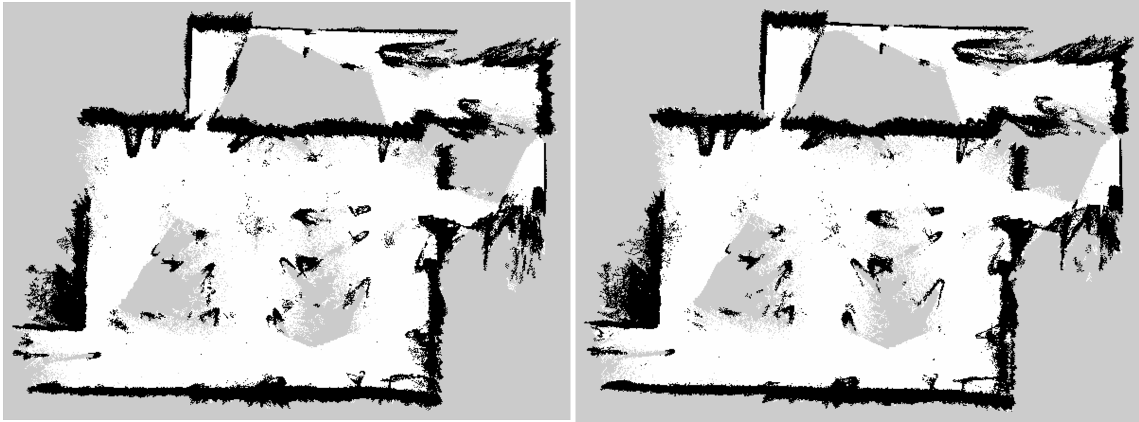


(a) No filtering applied. (b) People removed from pointcloud.

Figure 4.1.10: Gmapping dynamic environment results for the eight-shaped trajectory.

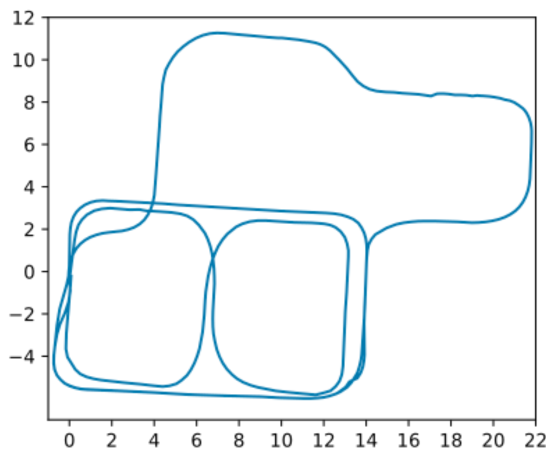
4.1.2.2 RTAB-Map

For RTAB-Map, the presence of people not only represents spurious features but can prevent some loop closures. This is due to the possibility of people appearing in front of the robot at certain locations and then not appear (or appear in a different position) in subsequent revisits. This often implies poor reliability of measurements, with the tests having diverged slightly in some attempts to build the map. Still, it has been concluded that RTAB-Map is also robust enough to handle dynamic environments and filtering the depth image does not provide much difference. There is a particular time that the object detection algorithm fails and a false feature is added to the map but it is fully removed when the AGV revisits that location. Figure 4.1.11 shows the results for the full lap and Figure 4.1.12 presents the results for the eight-shaped trajectory.

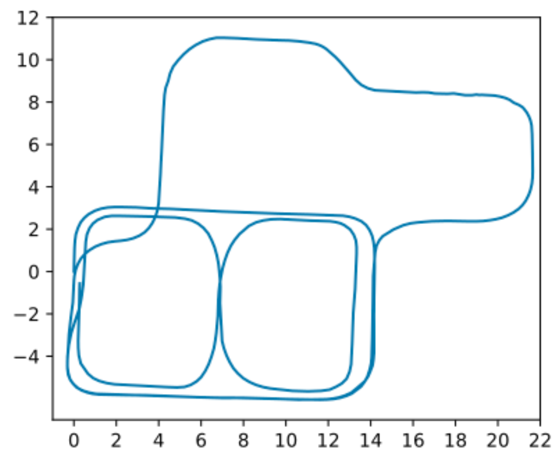


(a) Map with no filtering applied.

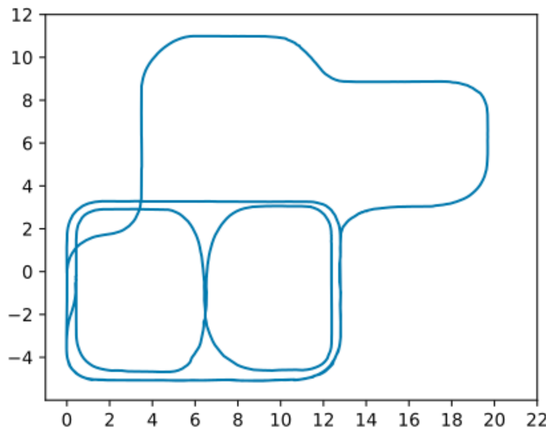
(b) Map with people removed from depth image.



(c) Trajectory with no filtering applied.



(d) Trajectory with people removed from depth image.



(e) Trajectory from the Navitec localisation system.

Figure 4.1.11: RTAB-Map dynamic environment results for the full lap.

One of the differences that can be seen in these results is that when people are removed, the trajectory is a bit smoother in the canteen area, correcting a sudden turn that is present in Figure 4.1.11(c), around coordinates (17,8). Also, the intersection of paths in

the middle of the eight-shaped trajectory is more similar to the real trajectory. That being said, the maps are very similar in both cases.

(a) Map with no filtering applied (b) Map with people removed from depth image.

Figure 4.1.12: RTAB-Map dynamic environment results for the eight-shaped trajectory.

For the isolated eight-shaped trajectory results in Figure 4.1.12 it is also evident that the maps are very similar whether people are removed from the depth image or not.

4.2 Pathfinding

This section presents the pathfinding results and the integration of the TensorFlow Object Detection API to stop and resume the navigation of the robot.

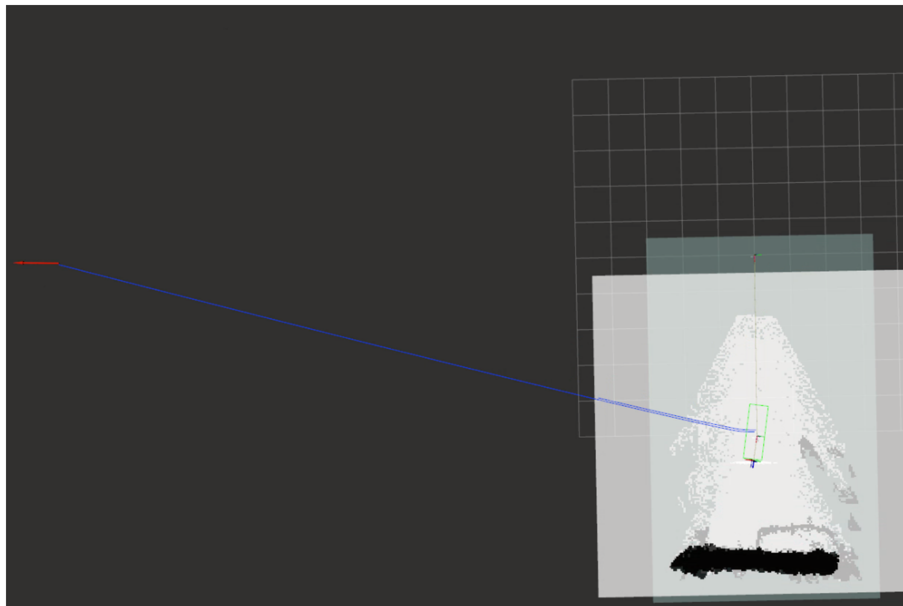
The pathfinding algorithm from the ROS Navigation Stack works as expected both in previously built maps and during the map building process. It generates a global path towards the navigation goal and outputs velocity commands to make the robot follow that path. The trajectory is recalculated according to updates to the map and whenever the robot sees an object that blocks the global navigation plan.

Figure 4.2.1 shows the path being calculated for RTAB-Map in two different conditions. In Figure 4.2.1(a) the path is drawn without knowledge of a part of the map, so the trajectory (in blue) is a straight line to the goal (represented by a red arrow). This path would be impossible because there is an obstacle between the robot and the destination (the shelves on the left of the map - see Figure 4.1.2(a)). When the remaining map is built by RTAB-Map, the trajectory is adjusted correctly, as we can see in Figure 4.2.1(b). In this occasion, the calculated path takes the robot around the shelves. A video related to this subject is presented in link [25].

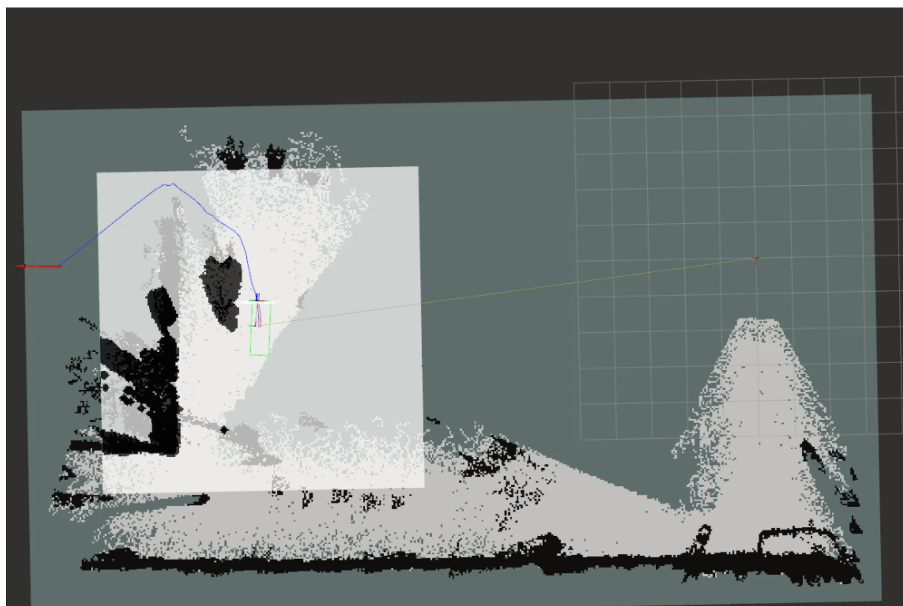
Note that the robot did not follow the suggested trajectory in Figure 4.2.1(a) in the recorded data, which justifies the map not being as expected if that was the case. That is

because we do not have the pathfinding system communicating with the robot's control system. However, in a fully integrated situation, the robot would follow that straight line until it found obstacles that prevented its trajectory, then, the recalculation of the path would take place in a similar fashion.

Figures 4.2.1 and 4.2.2 show two maps overlapping each other; the larger map built by RTAB-Map below the smaller local costmap in black and white, centred in the robot. The green rectangle is the robot footprint.



(a) Straight line trajectory.

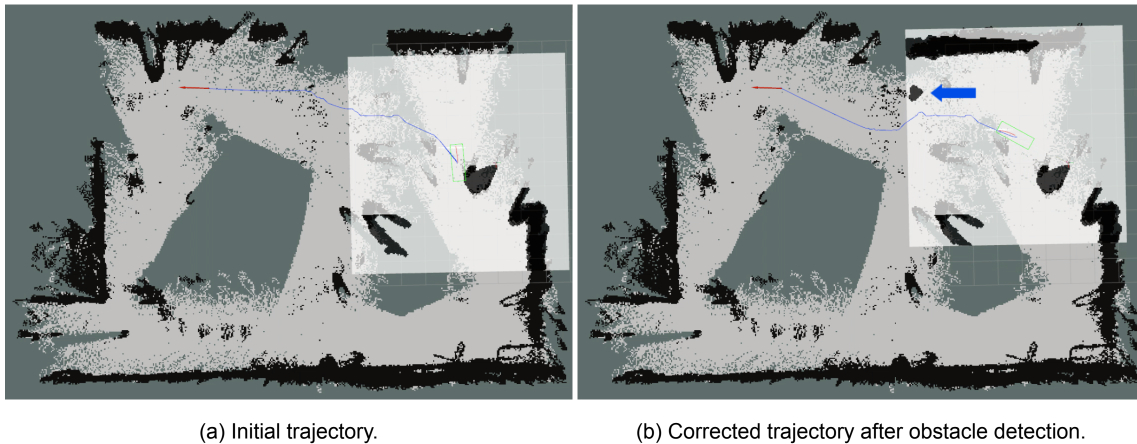


(b) Corrected trajectory.

Figure 4.2.1: Trajectory calculation and update according to the map.

Figure 4.2.2 presents the recalculation of the trajectory according to the position of an unmapped obstacle appearing in sight of the robot. This obstacle is generated by a person, since the algorithm that removes people from the depth image was intentionally neglected in order to generate obstacles, and can be seen in the upper left part of the local costmap (indicated by a blue arrow). The trajectory is deviated downwards so that the robot does not crash into the obstacle. A video is also available showing this feature,

in link [26].



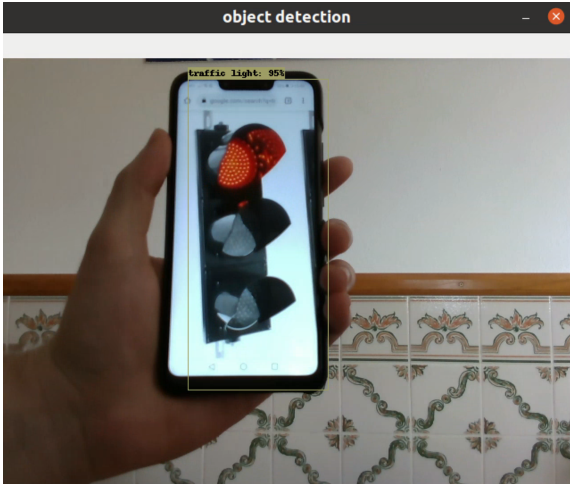
(a) Initial trajectory. (b) Corrected trajectory after obstacle detection.

Figure 4.2.2: Corrected trajectory according to a new obstacle.

The maps in Figures 4.2.1 and 4.2.2 are made using RTAB-Map, but the Navigation Stack also works with Gmapping, with the results being the same. This algorithm would also work with HectorSLAM if good map results could be attainable with our setup.

The final topic related to pathfinding is traffic light obedience. A SLAM test running a traffic light in a television screen has been employed. The television is placed in the robot's sight at a fixed location in the basement. However, a full SLAM validation has not been possible, but the light colour detection is nevertheless attained. This limitation arises from the fact that the object detection model SSD MobilenetV1 model trained over the COCO dataset also detects television sets. Thus, the output from the detection is the television and the traffic light image on the TV screen is seldom detected. Note that the model is not changed from its original form in order to stop detecting irrelevant objects, the irrelevant detections are simply ignored. The only way that this test could be run is if a real traffic light was at our disposal, which it was not. However, testing with a traffic light image on a mobile phone screen has shown that the colours are detected correctly and the appropriate topics to cancel and resume navigation are published.

Figure 4.2.3 shows the algorithm stopping the navigation through the detection of a red light. Figure 4.2.3(a) shows the colour image presented to the algorithm and Figure 4.2.3(b) shows the result of the colour identification. Figure 4.2.3(c) is the console output of subscribing to the *move_base/cancel* topic, which stops the publishing of velocity commands. We can see that this topic is published when the red traffic light is detected. Figure 4.2.3(d) is the output of subscribing to the *cmd_vel* topic, which corresponds to the velocity commands. It is shown that this topic returns values equal to zero when the *move_base/cancel* topic is published, i.e., the navigation is stopped.



(a) Colour Image.

```
The traffic light is red!
The traffic light is red!
The traffic light is red!
```

(b) Output from the object detection algorithm.

```
piresmicael@piresmicael-GL502VM: ~ 86x20
piresmicael@piresmicael-GL502VM:~$ rostopic echo move_base/cancel
WARNING: no messages received and simulated time is active.
Is /clock being published?
stamp:
  secs: 1626961639
  nsecs: 106106182
id: ''
---
stamp:
  secs: 1626961639
  nsecs: 187679005
id: ''
---
stamp:
  secs: 1626961639
  nsecs: 420337202
id: ''
---
```

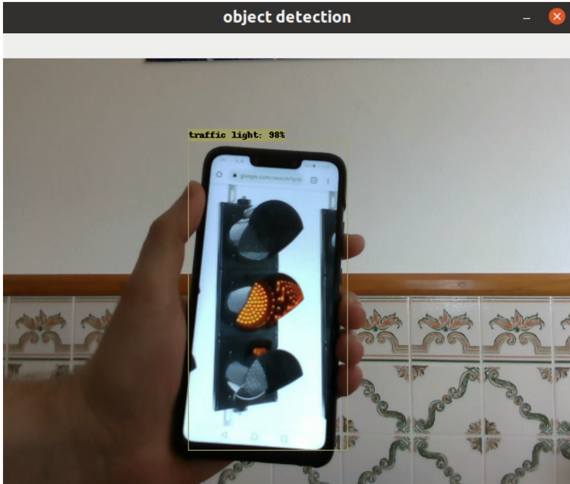
(c) *move_base/cancel* topic being published.

```
linear:
  x: 0.45
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.47368421052631565
---
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

(d) *cmd_vel* topic values turning to zero.

Figure 4.2.3: Red traffic light effect on navigation.

Figure 4.2.4 shows the successful detection of a yellow traffic light. The console outputs for the ROS topics are not shown because a yellow traffic light has no effect on navigation.



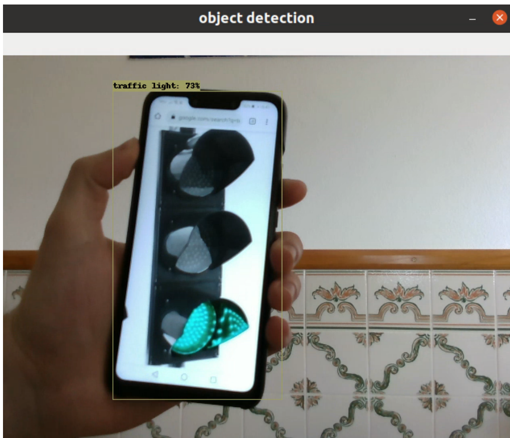
(a) Colour image.

```
The traffic light is yellow!
The traffic light is yellow!
The traffic light is yellow!
```

(b) Output from the object detection algorithm.

Figure 4.2.4: Yellow traffic light detection.

Figure 4.2.5 presents the successful detection of the green light and the restart to the navigation, since the *cmd_vel* topic has non-zero values.



(a) Colour image.

```
The traffic light is green!
The traffic light is green!
The traffic light is green!
```

(b) Output from the object detection algorithm.

```
linear:
  x: 0.4131578947368421
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.894736842105263
---
linear:
  x: 0.35789473684210527
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.894736842105263
```

(c) *cmd_vel* topic values.

Figure 4.2.5: Green traffic light effect on navigation.

Chapter V

Conclusion and Future Work

This work has been focused on the development of natural navigation methods for AGVs and AMRs with the use of the Intel RealSense D435i depth camera. This implies performing SLAM and allowing the robot to find the best path to a destination considering the robot's surroundings. Successful attempts to map a space have been carried out with Gmapping and RTAB-Map using RGB-D odometry. Pathfinding and traffic light obedience has also been implemented. Additionally, some attempts to improve the mapping results have been tested through the removal of people from the depth image using object detection algorithms.

It has been found that HectorSLAM is not a good choice for narrow field of view depth cameras like the Intel RealSense D435i since the amount of features in each frame is usually not enough for mapping, leading to system failure. This would not be the case in a 360º laser scanner, for instance. Furthermore, it has been determined that the ability to have some kind of delayed map/trajectory correction is very important, which HectorSLAM does not have. The other two SLAM algorithms tested have this capability and show much better results. In the case of Gmapping, this feature is provided by the particle filter, by keeping a number of particles that describe various possibilities for the state of the system. For RTAB-Map, this feature is done through pose graph optimisation in loop closures, which is very powerful.

All in all, RTAB-Map is the algorithm that provides the most accurate results in a more reliable way both in static and dynamic environments. Additionally, RTAB-Map also contains a very good implementation for performing RGB-D odometry, which we conclude that is a good way to generate an odometry source with a depth camera.

In dynamic environments, the Yolact object detection implementation is added to the system, in order to remove people from the camera's data to improve results. In the tests made as a part of this work, it was found that both Gmapping and RTAB-Map are robust against this type of moving obstacles and removing people from the data does not make much difference in the maps. That being said, the detection of people in the robot's surroundings is very important for the final iteration of the AMR-UVC project. For instance, to implement a solution that turns off the UV light used for disinfection when a person is detected so that the UVC radiation does not cause accidental injuries to anyone. Another feature that can be added through object detection is the ability for the robot to follow a person, for an AGV with industrial purposes.

Finally, for pathfinding, the results are very good, with the system having robust path planning capabilities in a variety of environments, either static or dynamic, with and without a previous map. The trajectories take the known map into account as well as current sensor data in order to plan feasible and collision-free navigation for the robot. The pathfinding system is also compatible with all SLAM algorithms tested. Furthermore, an object detection algorithm was added through the TensorFlow Object Detection API in order to detect traffic lights and respective colour, and the navigation is successfully stopped and resumed accordingly. We consider that this algorithm is ready for further iterations of the project with the integration with the AGV's control system.

References

[1] S. Yu, C. Fu, A. K. Gostar, and M. Hu, “A review on map-merging methods for typical map types in multiple-ground-robot slam solutions,” *Sensors*, vol. 20, no. 23, p. 6988, 2020.

[2] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, “A tutorial on graph-based slam,” *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[3] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf, “A flexible and scalable slam system with full 3d motion estimation,” in *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, IEEE, November 2011.

[4] G. Grisetti, C. Stachniss, and W. Burgard, “Improved techniques for grid mapping with rao-blackwellized particle filters,” *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

[5] “Depth cameras and rgb-d slam.” `https://www.kudan.io/archives/517`, Nov 2020.

[6] “Tracking camera t265 – intel realsense depth and tracking cameras.” `https://www.intelrealsense.com/tracking-camera-t265/`.

[7] S. Chen, “Kalman filter for robot vision: a survey,” *IEEE Transactions on Industrial Electronics*, vol. 59, no. 11, pp. 4409–4420, 2011.

[8] S. J. Julier and J. K. Uhlmann, “New extension of the kalman filter to nonlinear systems,” in *Signal processing, sensor fusion, and target recognition VI*, vol. 3068, pp. 182–193, International Society for Optics and Photonics, 1997.

[9] K. P. Murphy *et al.*, “Bayesian map learning in dynamic environments.,” in *NIPS*, pp. 1015–1021, 1999.

[10] G. Grisetti, C. Stachniss, and W. Burgard, “Improved techniques for grid mapping with rao-blackwellized particle filters,” *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

[11] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, “Monte carlo localization for mobile robots,” in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, vol. 2, pp. 1322–1328, IEEE, 1999.

[12] S. Thrun and M. Montemerlo, “The graph slam algorithm with applications to large-scale mapping of urban structures,” *The International Journal of Robotics Research*, vol. 25, no. 5-6, pp. 403–429, 2006.

[13] F. Lu and E. Milios, “Robot pose estimation in unknown environments by matching 2d range scans,” *Journal of Intelligent and Robotic systems*, vol. 18, no. 3, pp. 249–275, 1997.

[14] K. P. Murphy *et al.*, “Bayesian map learning in dynamic environments.,” in *NIPS*, pp. 1015–1021, 1999.

[15] M. Labbé and F. Michaud, “Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation,” *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[16] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, “The office marathon: Robust navigation in an indoor office environment,” in *International Conference on Robotics and Automation*, 2010.

[17] “Ros homepage.” `https://www.ros.org/`.

[18] “Rep-105: Coordinate frames for mobile platforms, aug 2021.” `https://www.ros.org/reps/rep-0105.html`.

[19] T. Moore and D. Stouch, “A generalized extended kalman filter implementation for the robot operating system,” in *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*, Springer, July 2014.

[20] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, “Yolact: Real-time instance segmentation,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 9157–9166, 2019.

[21] “Yolact ROS wrapper, aug 2021.” `https://github.com/Eruvae/yolact_ros`.

[22] “Gmapping basement mapping.” `https://youtu.be/YrDK4oitf4k`.

[23] “Rtab-map basement mapping.” `https://youtu.be/eI8bm1bgUds`.

[24] P. Leal, “Navigation solutions for autonomous mobile robots using lidar,” Universidade de Coimbra, 2021.

[25] “Pathfinding path updating with map.” `https://youtu.be/6R-H4BdsIQk`.

[26] “Pathfinding path deviating from obstacle.” `https://youtu.be/IVPmYvcyobo`.