Extended Kalman Filter SLAM Implementation for a Differential Robot with LiDAR

A thesis

submitted in fulfilment

of the requirements for the degree

of

Master of Engineering

at

The University of Waikato

by

Hisham Abdel Qader

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Year of submission

2018

Abstract

SLAM is an approach deployed in robotics to develop autonomous mobile robots. These robots have been deployed in numerous fields such as manufacturing, aerospace navigation, and other areas deemed dangerous to humans. SLAM techniques have made these robots to operate without necessarily having the prior maps, a shortcoming of the current robots which require prior maps. Several SLAM approaches exist, but EKF has been seen to possess all the useful features of convergence and consistency. Via SLAM, concurrency between localisation and mapping has been made possible.

An EKF SLAM algorithm has been presented in this thesis, which was implemented in a two-wheeled mobile robot. The robot autonomously navigated in a structured indoor environment, while simultaneously building a map and localising itself within that map. A 360 degrees LiDAR was used to measure the range and bearing of the surroundings and an ultrasound sensor was used to avoid the obstacles. Furthermore, the algorithm was implemented using Python 3.

Acknowledgements

I would like to relay my sincere gratitude to my supervisor, Dr. Chikit Au, for all the help he has given me. He gave me professional advice throughout this project and helped in purchasing needed hardware.

I would also thank Brett Nichol for his assistance in making holes in the robot's top plate for attaching the LiDAR and I thank Prof. Dr. Claus Brenner for his professional assistance in the algorithm. Furthermore, I thank Mary Dalbeth and Cheryl Ward for their administrative support and I thank my friend, Yousef AlShimmiri, for all the guidance and inspiration he gave me.

I would like to express my heartfelt gratitude to my wife, Abeer Suleiman, for her continuous support and encouragement and to my brothers Bashar, Husam, and Ghassan for their financial support. Finally, I thank my parents for raising me to be the man I am today.

Table of Contents

List of Figures

Chapter 1

Introduction

At the outset, it is worth noting that Simultaneous Localization and Mapping (SLAM) is a state-of-the-art approach where maps of given localities or surroundings are built while at the same time an estimation of a robot's location is generated (Khairuddin *et al.*, 2015). The key interest in this case, which inspired the need for further research, is the importance of having the two processes executed concurrently. It has been a norm that any mobile robot be programmed with prior maps, especially of the surrounding environment in which it operates. It is easier noting that such projects were of limited applications, especially with the ever-rising need to navigate foreign areas that are hardly reachable or even dangerous to human operations. Consequently, SLAM algorithms have been crucial in offering such mobile robots pivotal capability to pinpoint and constrain themselves as well as the environmental features with no need for prior maps. This indeed has proved vital for quite a number of navigation exercises.

Furthermore, one of the chief goals within the robotic discourse is developing autonomous robots. In order to achieve this pivotal goal, suitable algorithms have been sought, all aiming at minimising, if not eliminating, human involvement in their operations. The robots are to self-explore given unknown surroundings, successfully evading the obstacles as well as landmarks within the localities from which they are meant to operate. Such regions that have demanded the aforementioned robotic operations include aerial space, terrain and subsea among other regions which could be unreachable and/or deemed potentially risky to human beings. With increasing interest in the exploration of these, and more areas

motivated by further research aimed at say economic growth or medical explorations, SLAM is quite crucial.

1.1 Motivation

The need for autonomous mobile robots has increased recently but many of the fields requiring them usually cannot provide prior information. Hence the need for an algorithm that allows mobile robots to navigate and draw maps simultaneously.
The most important field urgently requiring SLAM is search and rescue. In cases of natural disasters such as earthquakes and buildings collapse with people being trapped under rubble, search and rescue teams using heavy machinery before having any information on how many people are trapped and where, could result in accidents and increases in the number of casualties. Also, sending search and rescue workers into areas of rubble to assist injured people could result in them also being trapped. To solve this issue, a small robot could navigate under the rubble and report back a map showing the exact locations of trapped people. Figure 1.1 shows an example where SLAM could have been used in search and rescue.
Another use can be in military operations, shown in Figure 1.2; where a small robot could navigate behind enemy lines and report back some intel on the enemy's base, or the locations of imprisoned soldiers. SLAM can also be used in scientific exploration, where an expedition needs to explore an inaccessible underground cave, as in Figure 1.3, or an undersea cave (Figure 1.4). Scientists might also want to explore dangerous locations where no human can enter, such as volcanoes. SLAM could be useful in aerospace navigation, or basically any location where humans cannot reach.

Figure 1.1: PGC building collapsed (Ministry for Culture and Heritage, 2011).

Figure 1.2: SLAM possible military usage (Radio New Zealand, 2016).

Figure 1.3: An underground cave that can be explored using SLAM (Swarbrick, 2015).

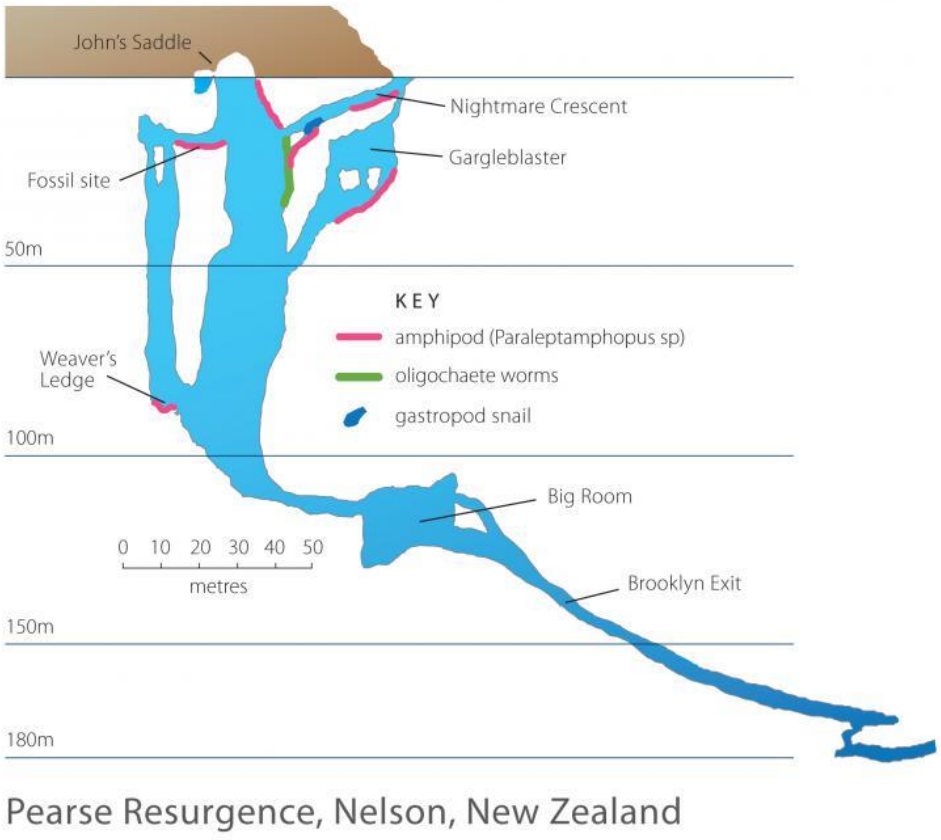

Figure 1.4: An underwater cave that can be explored by SLAM (NIWA, 2012).

1.2 Objectives

The main objective of this study is to assemble a mobile robot and use a SLAM algorithm to programme it to move around freely (unmanned) without a prior map, and receive a map of the surroundings after the robot finishes navigation. To achieve this, there are three specific objectives that need to be achieved:

1. To programme Raspberry Pi, the project's control unit, to help successfully operate a GoPiGo robot.
2. To successfully write and compile a Python code to receive inputs from the input devices and trigger the output devices.
3. To interface the Raspberry Pi, GoPiGo robot, ultra sound sensor, and LiDAR with an EKF SLAM algorithm.

1.3 Scope

As far as scope is concerned, this research project will only review SLAM algorithms as they are applied in autonomous mobile robots. Other technologies may only be mentioned but not explored. In particular, only EKF SLAM algorithm will be looked into in detail. It will suffice just mentioning the other SLAM techniques.

Chapter 2

Literature Review

The SLAM researchers have made amazing developments over the past 30 years, enabling large-scale real-world applications, and a steady transition of this technology to industry (Cadena *et al.*, 2016). The history of the SLAM problem can be divided into three main ages: the classical age, the algorithmic-analysis age, and the robust-perception age (Cadena *et al.*, 2016). Each section of this chapter will elaborate on each of these ages, showing the accomplishments achieved during these ages, in regards to algorithms, autonomous robots, and sensors; used to solve the SLAM problem.

2.1 The Classical Age

This age represents the first 20 years of SLAM, between the years 1986 and 2004 (Cadena *et al.*, 2016). In 1986, when probabilistic methods were only just beginning to enter both robotics and artificial intelligence (AI), the genesis of the probabilistic SLAM problem occurred at the IEEE Robotics and Automation Conference of the same year (Durrant-Whyte & Bailey, 2006). During the following few years, some key papers were introduced, such as (Durrant-Whyte, 1988; Smith & Cheeseman, 1986). They described statistically the relationships between landmarks and manipulating geometric uncertainty.

Meanwhile, early works in visual navigation were done by (Ayache & Faugeras, 1988). Also, (Chatila & Laumond, 1985) and (Crowley, 1989) were working on sonar-based navigation using Kalman filter based algorithms, which resulted soon after in (Smith *et al.*, 1990).

However, this work did not look at the convergence properties of the map or its steady-state behaviour. Instead, it was assumed that the predicted map errors would

never converge, and the researchers at the time believed in very small or even zero correlation between landmarks (Durrant-Whyte & Bailey, 2006) which reduced the full filter to a series of decoupled landmark to vehicle filters (Leonard & Durrant-Whyte, 1992) and (Rencken, 1993). Later in 1995, at the International Symposium on Robotics, (Durrant-Whyte *et al.*, 1996) explained the structure of the SLAM problem, the convergence result, and the coining of the acronym SLAM. This led to further research on convergence by (Csorba, 1998) and (Csorba *et al.*, 1996).

Later towards the end of the 20th century, researchers' work focused on improving the computational efficiency and solving problems related to data association and loop closure, while ensuring consistent and precise predictions for the map and robot pose (Durrant-Whyte & Bailey, 2006) and (Bailey & Durrant-Whyte, 2006). A degree of convergence between the Kalman filter based SLAM and the probabilistic localisation and mapping methods was achieved by (Thrun *et al.*, 1998). The 2000 IEEE International Conference on Robotics and Automation (ICRA) Workshop on SLAM discussed issues like algorithmic complexity, data association, and implementation challenges (Durrant-Whyte & Bailey, 2006). Since then, many SLAM workshops and summer schools were held, and many researchers and PhD. students have shown interest in SLAM; which foresaw a tremendous success in building the field (Durrant-Whyte & Bailey, 2006).

2.1.1 The Algorithms

The classical age uncovered the start of the main probabilistic formulations for SLAM, including solutions based on Extended Kalman Filters, Rao-Blackwellised Particle Filters, and maximum likelihood estimation (Cadena *et al.*, 2016).

These three algorithms are mathematical derivations of the recursive Bayes rules, and their popularity is due to the uncertainty and sensor noise of robot mapping

which are investigated by explicitly modelling different sources of noise and their effects on measurements (Thrun, 2002).

2.1.1.1 Extended Kalman Filter

The most common representation is in the form of a state-space model with additive Gaussian-noise, which led to the use of the Extended Kalman Filter (EKF) to solve the SLAM issue (Durrant-Whyte & Bailey, 2006). It is one example of a Bayes filter, which does not only "guess" the new state, but also calculates the probability of that state being correct (Thrun *et al.*, 2005). Many existing SLAM solutions are based on EKF SLAM, such as (Davison & Murray, 2002), (Leonard & Newman, 2003), (Jensfelt *et al.*, 2006), and (Se *et al.*, 2002).

Kalman Filter methods have been proven to be capable of mapping large-scale environments, including outdoor and underwater, while simultaneously predicting the robot's position relative to the map (Thrun *et al.*, 2005) and (Dissanayake *et al.*, 2000).

The basis for the EKF SLAM method is to describe the vehicle motion model as a function of its kinematics, adding a zero mean uncorrelated Gaussian motion covariance (Durrant-Whyte & Bailey, 2006). The observation or measurement model is described by a function of its geometry, adding a zero-mean uncorrelated Gaussian measurement error (Durrant-Whyte & Bailey, 2006). An example of a standard EKF method can be found in (Dissanayake *et al.*, 2001). This EKF SLAM solution inherits the same benefits as the standard EKF solutions to navigation, but at the same time it also inherits the same problems; such as convergence, computational effort, data association, and nonlinearity (Durrant-Whyte & Bailey, 2006). Many studies have been completed to produce efficient variants of EKF SLAM solutions, such as (Guivant & Nebot, 2001) and (Leonard & Feder, 2000) have introduced a computationally efficient variant. A Kalman filter based SLAM

can converge to the true solution after collecting enough information about the surrounding environment (Huang & Dissanayake, 2016).

2.1.1.2 Rao-Blackwellized Particle Filter

Written as Rao-Blackwellised in (Cadena *et al.*, 2016) or Rao-Blackwellized in (Durrant-Whyte & Bailey, 2006), it describes the robot motion model as a set of samples of a more general non-Gaussian probability distribution (Durrant-Whyte & Bailey, 2006). One example of this filter is the FastSLAM algorithm introduced by (Montemerlo *et al.*, 2002). All previous work tried to improve the performance of EKF SLAM algorithm, keeping its essential linear Gaussian assumptions, but FastSLAM with particle filtering was the first to directly represent the nonlinear process model and non-Gaussian pose distribution (Durrant-Whyte & Bailey, 2006). This approach was influenced by (Murphy, 2000) and (Thrun *et al.*, 2000b). In FastSLAM, the recursive prediction of the pose states is done by particle filtering and EKF SLAM is used for map states estimation (Durrant-Whyte & Bailey, 2006). The theory behind this algorithm is derived from Sequential Important Sampling (SIS) (Doucet, 1998).

Two versions of FastSLAM have been introduced in the literature, FastSLAM 1.0 (Montemerlo *et al.*, 2002) and FastSLAM 2.0 (Michael *et al.*, 2003). They differ only in the form of their proposed distribution and in their importance weight; FastSLAM 2.0 is more efficient than FastSLAM 1.0 (Durrant-Whyte & Bailey, 2006). For FastSLAM 1.0, the proposed distribution is the motion model; while in FastSLAM 2.0, it includes the current observation (Durrant-Whyte & Bailey, 2006). Both versions of FastSLAM suffer from degeneration due to their inability to forget the past (Durrant-Whyte & Bailey, 2006), nevertheless, (Michael *et al.*, 2003) proved that FastSLAM 2.0 algorithm is applicable in real outdoor environments and generates an accurate map.

2.1.1.3 Maximum Likelihood Estimation

Based on the Expectation Maximisation (EM) algorithm, which is used for state estimation with unknown data association (Lachlan & Krishnan, 1997), this algorithm searches iteratively by alternating a step that calculates expectations over the data association and related latent variables, followed by a step that computes a new mode under these fixed expectations. This search leads to a sequence of state estimates (such as maps) of increasing likelihood (Thrun *et al.*, 2005). An example of the usage of this algorithm is shown in (Thrun, 2001).

This algorithm is ideal for mapping but not localisation (Aulinas *et al.*, 2008), and is able to build a map when the robot pose is known (Burgard *et al.*, 1999). However, the need to process the same data numerous times to find the most likely map makes this algorithm inefficient and not appropriate for real-life implementations (Chen *et al.*, 2007). Hence, in real-life, only the mapping is accomplished using this algorithm and other algorithms are used for localisation (Thrun, 2002).

2.1.2 SLAM Implementations

Prior to actual SLAM implementations, there have been a few robotic systems working in ambiguous situations that paved the way for true SLAM applications. One of these was (Durrant-Whyte, 1996) which was a straddle carrier, developed at the University of Sydney, that helped in moving containers faster than human workers (Thrun *et al.*, 2005). Another example is an interactive museum tour-guide robot, shown in (Nourbakhsh *et al.*, 1999) and (Thrun *et al.*, 2000a) which safely leads visitors through heavily crowded museums (Thrun *et al.*, 2005).

Practical experiments of probabilistic SLAM have become increasingly impressive during the classical age, covering larger areas in more challenging environments (Durrant-Whyte & Bailey, 2006). The “explore and return” experiment by (Newman *et al.*, 2002), shown in Figure 2.1, was a medium-scale indoor experiment

that confirmed the non-divergence properties of EKF SLAM by returning, completely autonomously, to a marked starting position. The robot is manually controlled during exploration by an operator, who is completely dependent on the real-time rendering of the robot's map, then it plans its return trip without any human help (Durrant-Whyte & Bailey, 2006).

Another SLAM experiment was completed by (Guivant & Nebot, 2001), shown in Figure 2.2. It was the first SLAM application in very large outdoor environments. It dealt with computational concerns of real-time operation, while also addressing high-speed robot motion, non-flat terrains, and dynamic clutter. They closed several large loops and compared their results with accurate ground truth GPS, which showed the practical reliability of their algorithm. The data from their Victoria Park experiments are available online, and have become a benchmark within the SLAM research community (Durrant-Whyte & Bailey, 2006).

The classical age witnessed many SLAM applications in a wide range of fields, including indoor (Bosse *et al.*, 2004), (Davison *et al.*, 2004), (Castellanos *et al.*, 1998), and (Chong & Kleeman, 1999); outdoor (Folkesson & Christensen, 2004); airborne (Kim & Sukkarieh, 2003); and undersea (Newman & Leonard, 2003) and (Williams *et al.*, 2000). Different types of sensors were used such as bearing only (Deans & Hebert, 2001) and range only (Leonard *et al.*, 2002).

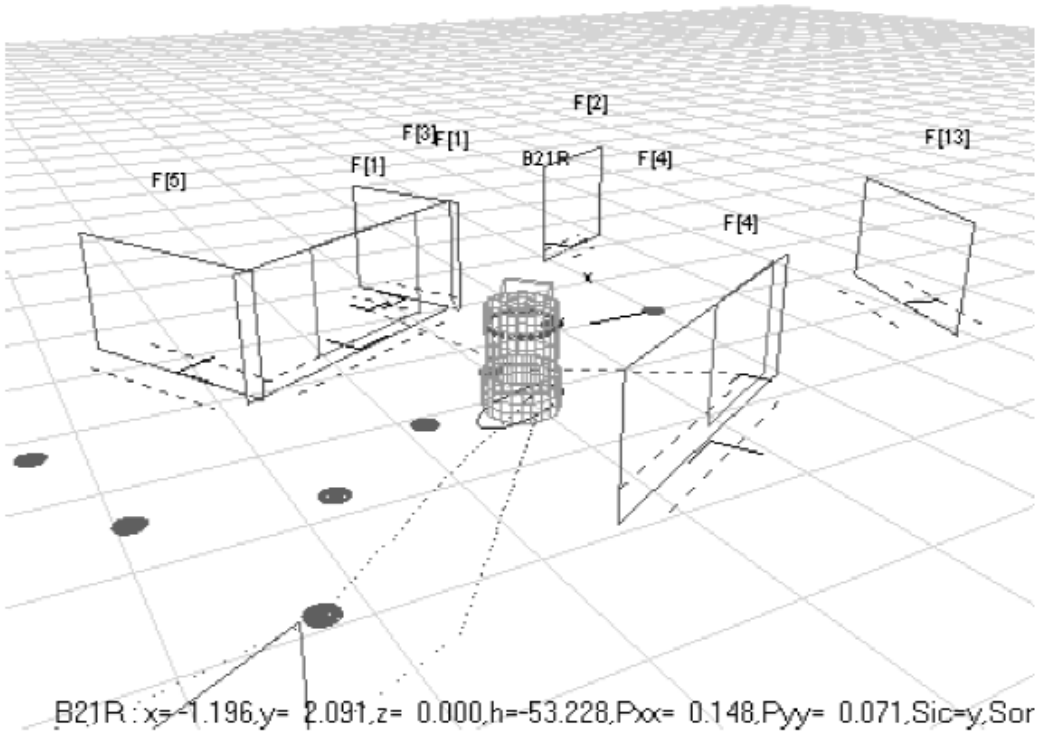

Figure 2.1: SLAM experiment by (Newman *et al.*, 2002).

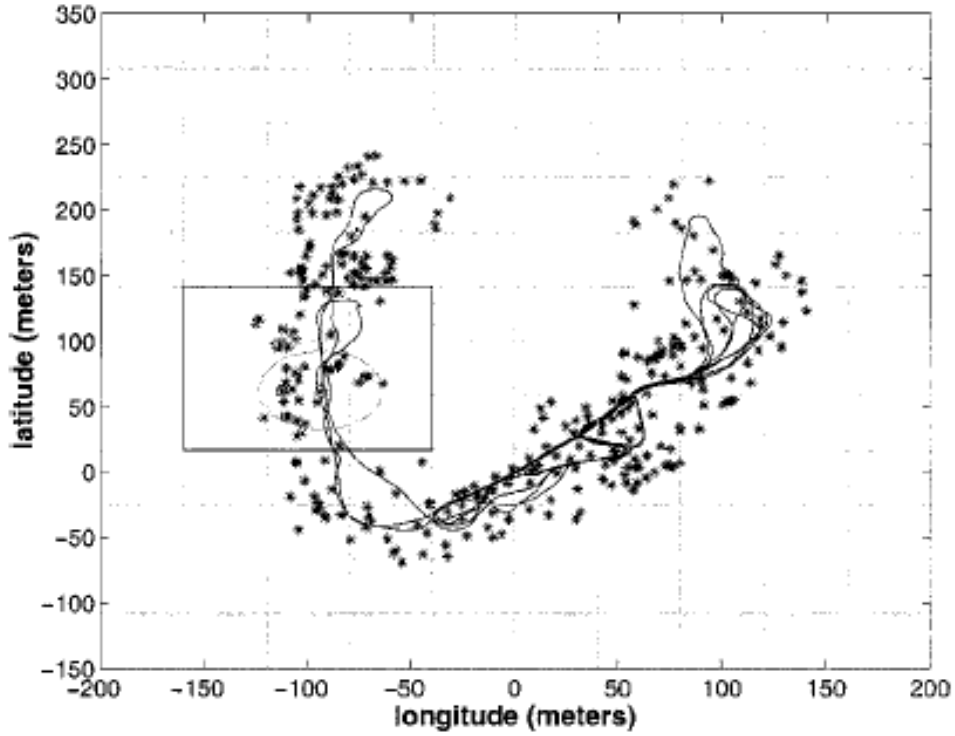

Figure 2.2: SLAM experiment by (Guivant & Nebot, 2001).

2.2 The Algorithmic-Analysis Age

This age occurred in the period between 2004 and 2015 (Cadena *et al.*, 2016). As can be seen from its name, the age represented the analysis of already developed SLAM algorithms. It aimed at tuning those algorithms to get better SLAM solutions, and at the same time, exploring new unaddressed fields where SLAM can be exploited.

2.2.1 SLAM Fundamental Properties

Researchers of the algorithmic-analysis age focused on the study of essential properties of SLAM, including observability, convergence, and consistency; and they developed the main open-source libraries of SLAM (Cadena *et al.*, 2016).

2.2.1.1 Observability

Observability is the ability to calculate the system state from a sequence of control actions and observations (Dissanayake *et al.*, 2011). It has been demonstrated that with bearing-only and range-only sensors, specific sensors are needed for target tracking (which represents the observation step of SLAM) to be observable (Song, 1996) and (Song, 1999). When using a range and bearing sensor, SLAM is observable even when the landmarks are moving, but the robot's initial pose needs to be known or assumed as known (which is the case in almost all of SLAM practical demonstrations) for the SLAM observability to hold (Dissanayake *et al.*, 2011).

2.2.1.2 Convergence

If the uncertainty of the estimated state converges to a finite value, given enough number of observations or amount of time, then convergence is achieved (Dissanayake *et al.*, 2011). When the data flow from the observations to the landmark states, which are stationary, landmark uncertainty will monotonically

decrease during SLAM (Dissanayake *et al.*, 2011). (Dissanayake *et al.*, 2001) and (Huang & Dissanayake, 2007) confirmed this fact for both linear and non-linear cases in their work on EKF based SLAM algorithm (Dissanayake *et al.*, 2011).

2.2.1.3 Consistency

A solution to the SLAM problem is consistent if the estimate is unbiased and the estimated covariance matrix equals the real mean square error (Bar-Shalom *et al.*, 2004). In EKF SLAM, inconsistency can occur when the relevant Jacobians are not evaluated at the true system state (Dissanayake *et al.*, 2011). Inconsistency became a clear problem only when researchers started using complex geometric landmarks (Dissanayake *et al.*, 2011), such as using line features in large-scale environments (Rodriguez-Losada *et al.*, 2006). Fortunately, inconsistency can be tolerated in some practical uses of SLAM; such as search and rescue, where the purpose of the map is purely for human interpretation and the topological structure of the map being correct is the only important information (Dissanayake *et al.*, 2011).

2.2.2 The Algorithms

One of the algorithms developed during the algorithmic-analysis age is called graph-based SLAM. It involves creating a graph whose nodes represent robot poses or landmarks, and in which an edge between two nodes encodes a sensor measurement that constrains the connected nodes (Grisetti *et al.*, 2010). Despite its proposal in 1997 by (Lu & Milios, 1997), it took many years for it to be popular due to its high complexity (Grisetti *et al.*, 2010). There are many graph-based SLAM approaches in the literature, and to name a few, (Frese *et al.*, 2005), (Dellaert & Kaess, 2006), (Kaess *et al.*, 2007), (Konolige *et al.*, 2010), and (Grisetti *et al.*, 2009).

Another approach is the Visual Odometry (VO), which is the process of predicting the robot pose using only the input image(s) from a single or multiple cameras (Scaramuzza & Fraundorfer, 2011). Since it works by detecting motion changes in the received images, the environment where the robot is moving needs to have enough illumination, and the image needs to have a static scene with enough texture to allow apparent motion to be detected. In addition, successive frames must have sufficient scene overlap (Scaramuzza & Fraundorfer, 2011). A SLAM solution using VO is called V-SLAM, and successful results of this algorithm can be found in (Clemente *et al.*, 2007), (Civera *et al.*, 2010), (Strasdat *et al.*, 2010), and (Mei *et al.*, 2011).

Polygon-based SLAM or 3D SLAM uses planar segments composed of infinite planes, and those segments are then connected with landmarks of the environment. Using polygon sets, landmarks are denoted with planar segments. The advantage of this method is that it produces high-detail small-size maps, well-suited for higher-level tasks like interacting with the environment; but, it needs very high computational power (Saeedi *et al.*, 2016). Stereo or RGBD cameras, such as in (Dryanovski *et al.*, 2013) and (Sturm *et al.*, 2013), and 3D laser scanners, such as (Segal *et al.*, 2009), are commonly used sensors in 3D SLAM applications (Saeedi *et al.*, 2016). For other examples please see (Weingarten & Siegwart, 2006) and (Pathak *et al.*, 2010).

Another approach to solving SLAM came to be during this age, which was based on artificial intelligence, called AI SLAM. In such solutions, the filtering or smoothing is realised using AI algorithms (Saeedi *et al.*, 2016). Many practical experiments with this approach were conducted, such as Chatterjee (2009), (Wyeth & Milford, 2009), (Saeedi *et al.*, 2011), and (Salas-Moreno *et al.*, 2013).

Single-robot SLAM is not an easy task, and adding more robots to the solution makes it even harder. This is because now, multiple robots must incorporate all available data to build a consistent global map, while simultaneously localising themselves within that map. On the other hand, multiple-robot SLAM has advantages such as performing missions faster and being robust to failure of any of the robots, but this comes with an extremely complex system that requires coordination between the robots (Saeedi *et al.*, 2016). The multiple-robot SLAM literature has much research, each using a different approach; for example, EKF (Chellali, 2013), Extended Information Filter (EIF) (Thrun & Liu, 2005), Rao-Blackwellized Particle Filter (Carlone *et al.*, 2010), Expectation Maximization (EM) (Indelman *et al.*, 2014), Cooperative Positioning System (CPS) (Tobata *et al.*, 2012), manifold representation (Howard *et al.*, 2006), map merging (Saeedi *et al.*, 2014), and 3D SLAM (Vidal-Calleja *et al.*, 2011).

2.3 The Robust-Perception Age

The Robust Perception Age represents the SLAM research work from 2015 to date, and triggers the question “is SLAM solved?”. This question is very hard to answer, because the answer is known only for a given combination of robot/environment/performance. For example, current SLAM algorithms can easily fail when either the robot motion or the environment is too challenging, or a strict performance is required (Cadena *et al.*, 2016).

This age is characterised by robust performance, high-level understanding, resource awareness, and task-driven perception. The SLAM system should have low failure rate while operating for an extended period of time in various environments, and include a fail-safe mechanism and self-tuning capability. It should adapt to the available resources and the given task (Cadena *et al.*, 2016).

Few works have been proposed recently attempting to accomplish the above characteristics and improve previous SLAM solutions. Newcombe et al. addressed the small-scale dynamic environment reconstruction issue (Newcombe *et al.*, 2015), but it is still greatly unaddressed for non-rigid large-scale maps (Cadena *et al.*, 2016). Mazuran et al. proposed a solution to the scalability issue of the graph-based SLAM needed nodes for long term SLAM operation, using the Nonlinear Graph Sparsification (NGS) method (Mazuran *et al.*, 2016). For a detailed review of the current and future developments of SLAM, the reader is encouraged to read the article of Cadena et al. (Cadena *et al.*, 2016).

2.4 Related Work

Much research in the current SLAM literature are related to the study investigated in this thesis, differing only in minor aspects. This section provides a comparison between such researches, and how their authors implemented SLAM using what types of robots, their purpose, used software, used sensors, and algorithms.

Different types of robots were used to suit different purposes and implementations. A humanoid robot was used as an autonomous tour guide in (Pujals, 2014). Omnidirectional robots were used by (Charabaruk, 2015) and (Berg, 2013), while (Wang *et al.*, 2013) used a human-driven car for testing only. A two-wheel robot, similar to the one used in this study, was used by (Santhanakrishnan *et al.*, 2017). However, four-wheel robots were the popular used robots (Teslić *et al.*, 2011), (Colon & Pécsi), (Klančar *et al.*, 2014), and (Darmstadt *et al.*, 2014).

Those SLAM systems were developed for different purposes, such as hazardous material handling (Charabaruk, 2015), soil sampling (Gan, 2015), maintenance (Berg, 2013), and robotic competition (Bradley *et al.*, 2013) and (Darmstadt *et al.*, 2014). Variant operating systems were installed, such as Ubuntu (Pujals, 2014),

(Colon & Pécsi, 2013), and (Bradley *et al.*, 2013); and Windows (Gan, 2015) and (Berg, 2013).

Robot Operating System (ROS) was used by (Pujals, 2014), (Charabaruk, 2015), (Colon & Pécsi, 2013), (Bradley *et al.*, 2013), and (Darmstadt *et al.*, 2014). Gazebo, a software for simulated testing, was used in (Pujals, 2014) and (Darmstadt *et al.*, 2014). Programming languages used to code the SLAM systems were C++ in (Pujals, 2014), (Colon & Pécsi, 2013), and only for analogue to digital conversion of remote control signal in (Gan, 2015); Matlab for mapping, autonomous and LiDAR control in (Gan, 2015); and Go, an open source programming language maintained by Google, in (Berg, 2013).

The laser range/bearing sensor (LiDAR) was found to be the most popular among SLAM researchers; it was used by (Pujals, 2014), (Wang *et al.*, 2013), (Teslić *et al.*, 2011), (Colon & Pécsi, 2013), (Klančar *et al.*, 2014), (Gan, 2015), (Berg, 2013), (Bradley *et al.*, 2013), (Darmstadt *et al.*, 2014), and (Santhanakrishnan *et al.*, 2017). However, Colon and Pécsi also used a 3D LiDAR for 3D mapping (Colon & Pécsi, 2013), and Charabaruk used the laser ranger in odometry prediction instead of the wheel encoders (Charabaruk, 2015). Wheel encoders were used by (Pujals, 2014), (Teslić *et al.*, 2011), (Wang *et al.*, 2013), (Colon & Pécsi, 2013), (Klančar *et al.*, 2014), and (Berg, 2013); however, (Bradley *et al.*, 2013) did not use these encoders and (Darmstadt *et al.*, 2014) used them only for robot speed control.

Other types of sensors were installed, for example (Colon & Pécsi, 2013) also used an Innertial Measurement Unit (IMU), for sensing the robot's orientation, which had a built-in GPS receiver. IMU was also installed by (Darmstadt *et al.*, 2014), who used RGB-D and thermal cameras for detecting injured humans and installed an ultrasound sensor facing backwards to detect obstacles while the robot is moving backward. Pujals added a second LiDAR for obstacle detection (Pujals, 2014). Gan

installed a low-accuracy GPS and an electronic compass for predicitng the robot's heading (Gan, 2015), and Bradley, Albu et al. also installed a GPS (Bradley *et al.*, 2013).

Some of these robots were designed for indoor use, such as (Pujals, 2014), (Teslić *et al.*, 2011), (Klančar *et al.*, 2014), and (Berg, 2013); while other robots were designed for outdoor use, such as (Gan, 2015) and (Bradley *et al.*, 2013); while (Colon & Pécsi, 2013) was designed to work in both.

EKF SLAM was implemented in much research such as (Teslić *et al.*, 2011), (Wang *et al.*, 2013), (Colon & Pécsi, 2013), (Klančar *et al.*, 2014), and (Santhanakrishnan *et al.*, 2017). Berg implemented Hector SLAM, an algorithm based on EKF SLAM (Berg, 2013), and Bradley, Albu et al. used it as a ready-coded ROS package (Bradley *et al.*, 2013). Colon and Pécsi used a ready-coded EKF SLAM and modified its pose covarinace matrix for better results (Colon & Pécsi, 2013), while Berg used EKF for pose prediction and correction and tinySLAM for mapping and feature matching (Berg, 2013). Pujals implemented a ready-coded graph-based SLAM called "openKarto" with slight modifications, and coded the exploration algorithm only (Pujals, 2014). Charabaruk and Bradley, Albu et al. used ready-coded SLAM ROS packages (Charabaruk, 2015) and (Bradley *et al.*, 2013). Line-segment mapping was used by (Teslić *et al.*, 2011), (Klančar *et al.*, 2014), and (Santhanakrishnan *et al.*, 2017); while Santhanakrishnan, Rayappan et al. accomplished it with a point-feature approach.

The robot of Gan was designed to work on a flat field (usualy a farm), which lacked the existance of landmarks; so instead of installing the LiDAR on top of the robot, which is the case of most SLAM systems, it was stationary and used the robot itself as a dynamic landmark. The controller of this robot had a user interface showing Google maps, which allowed the user to click anywhere on the map and the robot

would navigate autonomously to reach the destination and take a soil sample (Gan, 2015). (Darmstadt *et al.*, 2014) used grid mapping, since it is easier for human interpretation in search and rescue operations, and octomap package, with slight modifications, for 3D mapping.

Figure 2.3 shows the resulting map of (Pujals, 2014). Figure 2.4 shows the

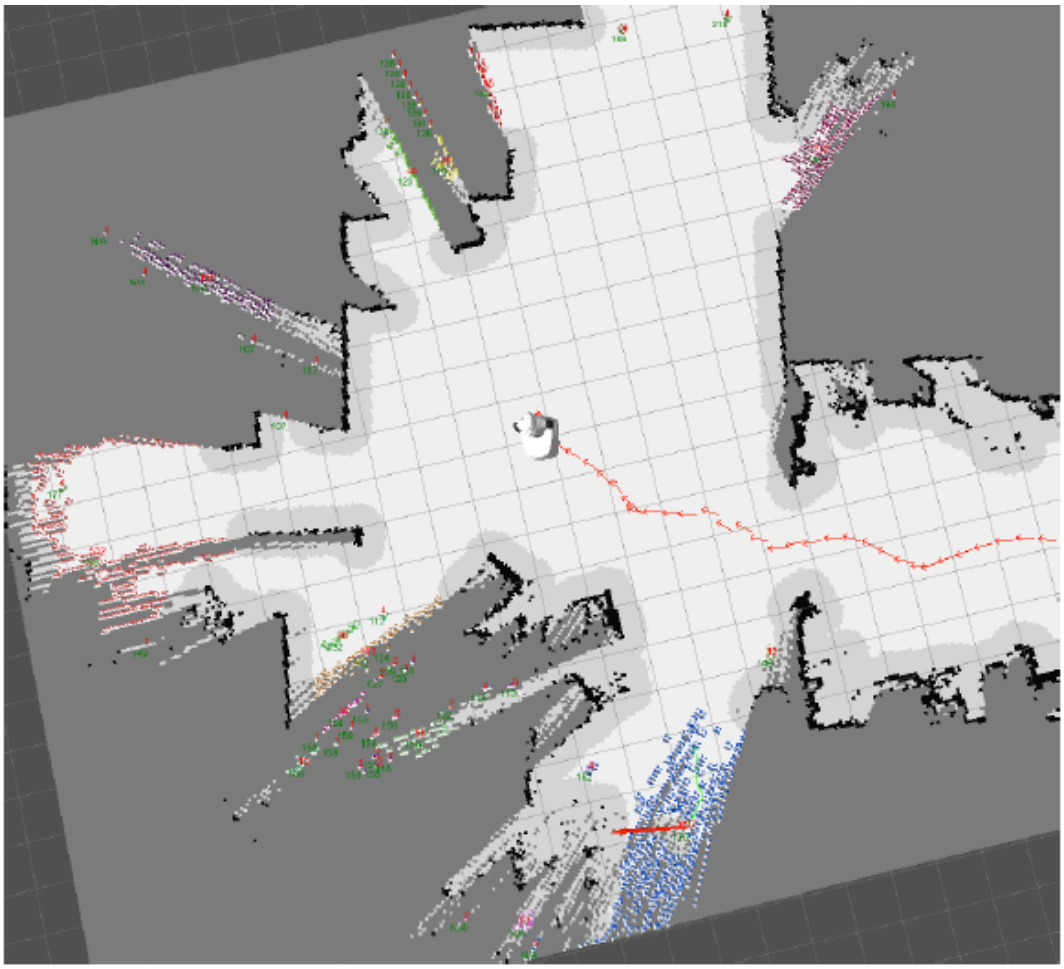

Figure 2.3: Screenshot of the map showing the robot exploring (Pujals, 2014).

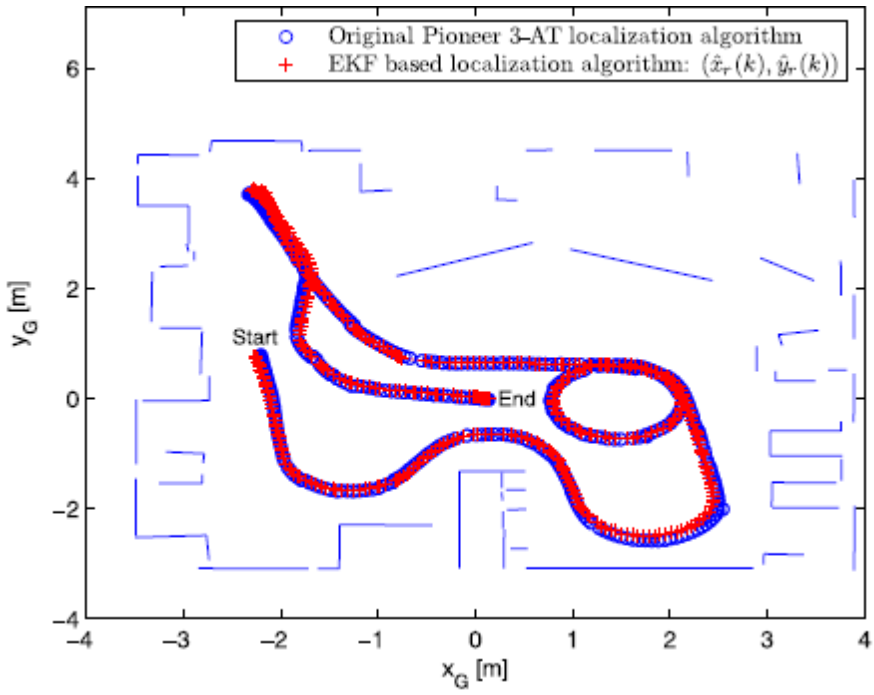

Figure 2.4: EKF based localisation with line segments representing the walls of a room (Teslić *et al.*, 2011).

successful localisation and mapping of (Teslić *et al.*, 2011). The trajectory of the robot resulted from the algorithm implemented by (Wang *et al.*, 2013), is shown in Figure 2.5. The maximum error in this experiment was found to be 3.28m, while the average error was around 1.5m.

Colon and Pécsi presented results for four different testing scenarios, Figure 2.6 shows the resulting map of scenario #4. The average error of the estimated final pose of (Klančar *et al.*, 2014) was found to be 4cm, with 0.7° average error in the robot's heading, Figure 2.7 shows the estimated trajectory and the environment map of (Klančar *et al.*, 2014).

The results of Gan's research shows an average error of 0.194m, but if the test with outliers is removed, it decreases to 0.127m. Figure 2.8 shows the robot's trajectory of one of the test runs (Gan, 2015). Berg executed many tests, each with different approach and criteria, one of the resulting maps is shown in Figure 2.9 (Berg, 2013).

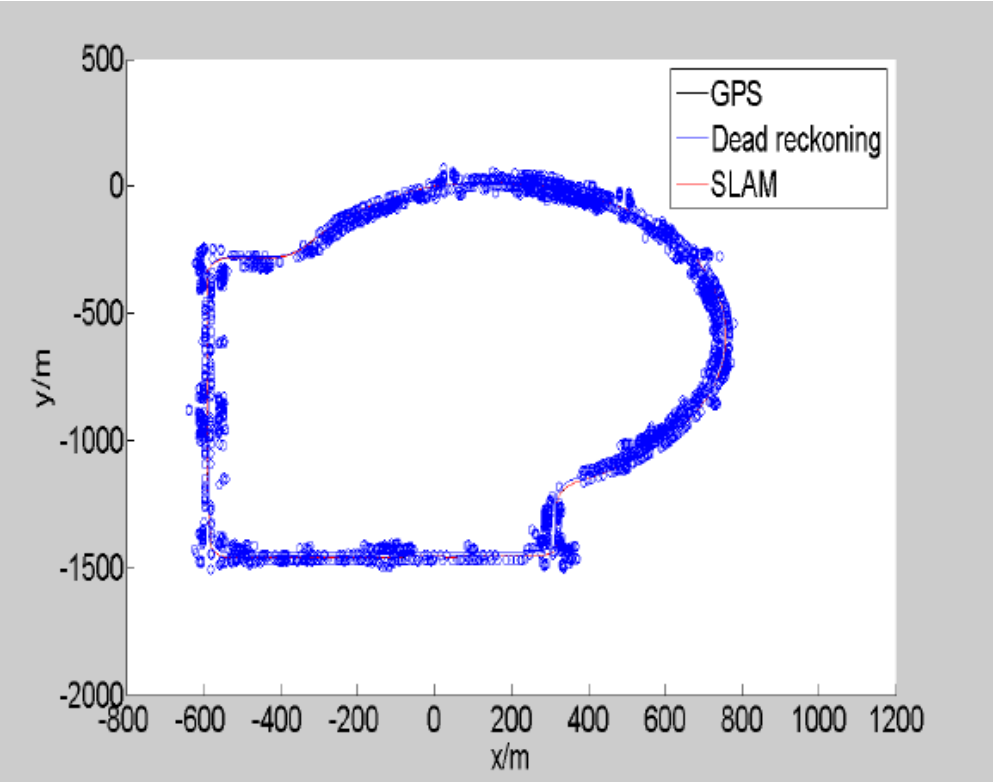

Figure 2.5: The trajectory of the robot (Wang *et al.*, 2013).

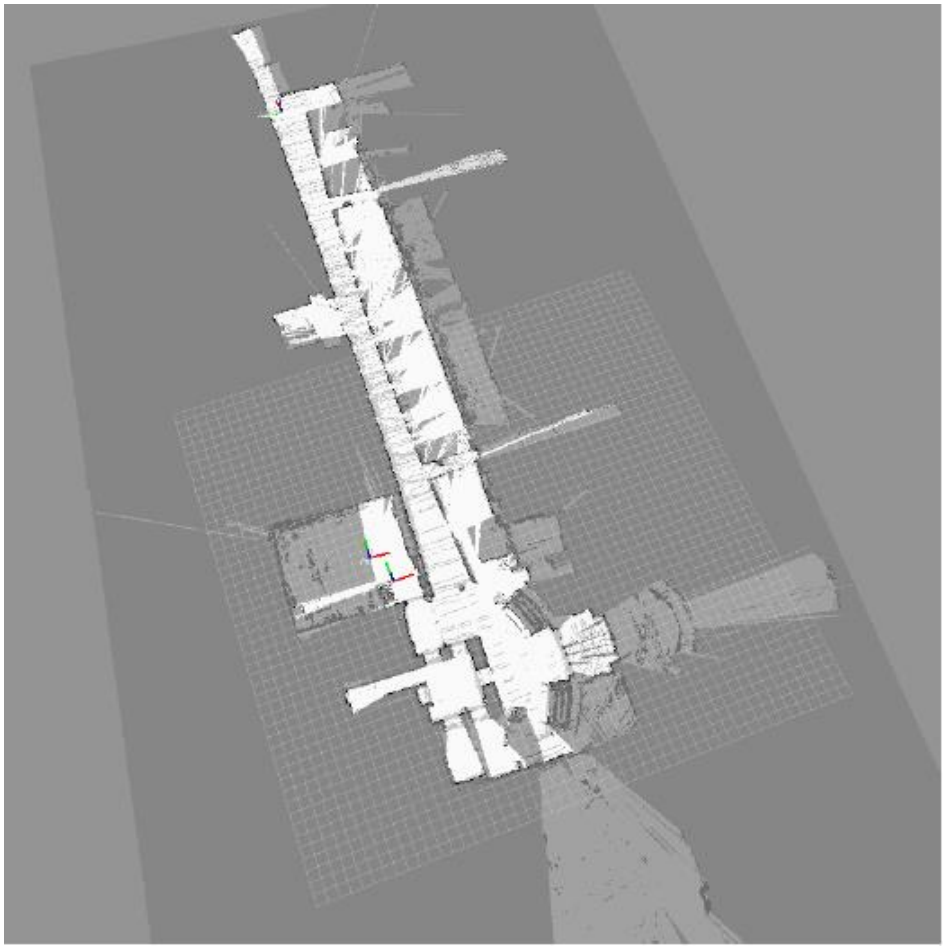

Figure 2.6: Octomap and Hector Mapping (Colon & Pécsi, 2013).

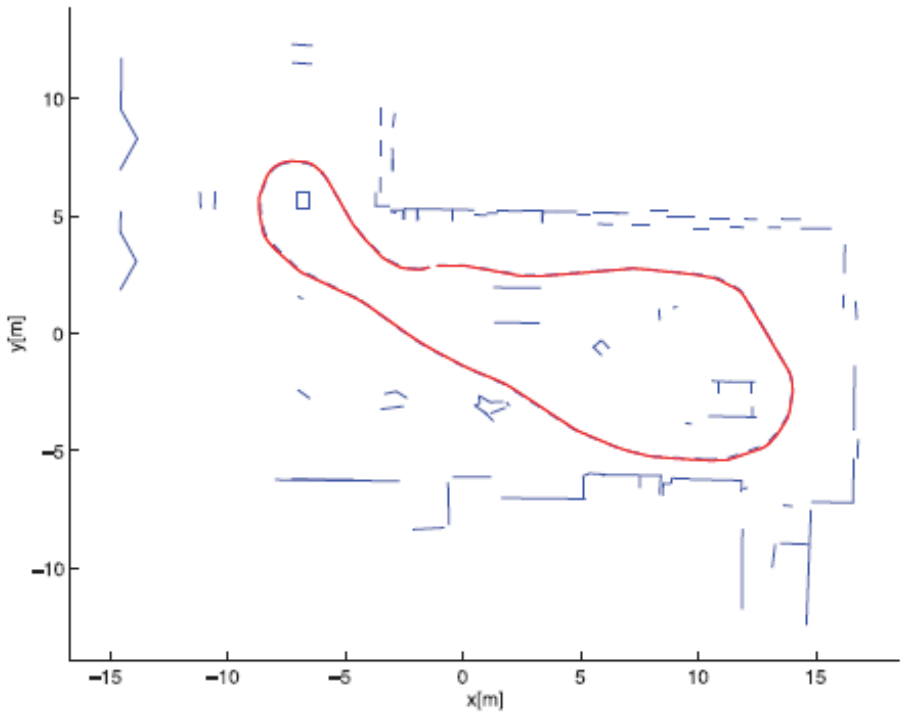

Figure 2.7: The estimated trajectory of the robot and the environment map (Klančar *et al.*, 2014).

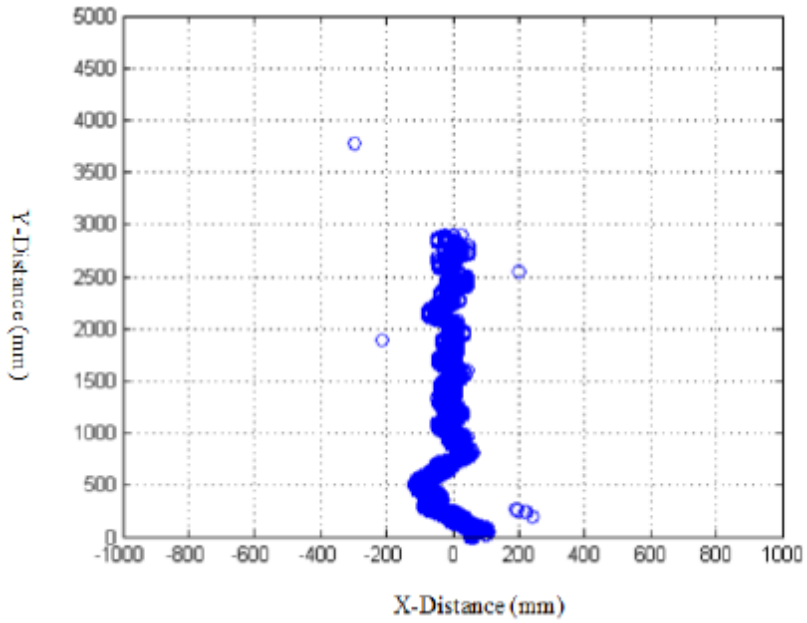

Figure 2.8: Robot's trajectory (Gan, 2015).

(a) With odometry (b) Without odometry

Figure 2.9: The resulting maps with and without odometry, both with identical LiDAR data (Berg, 2013).

Santhanakrishnan, Rayappan et al. proved that the implementation of EKF with point features approach leads to a highly precise SLAM solution with an average error of ±0.11m, see Figure 2.10 (Santhanakrishnan *et al.*, 2017).

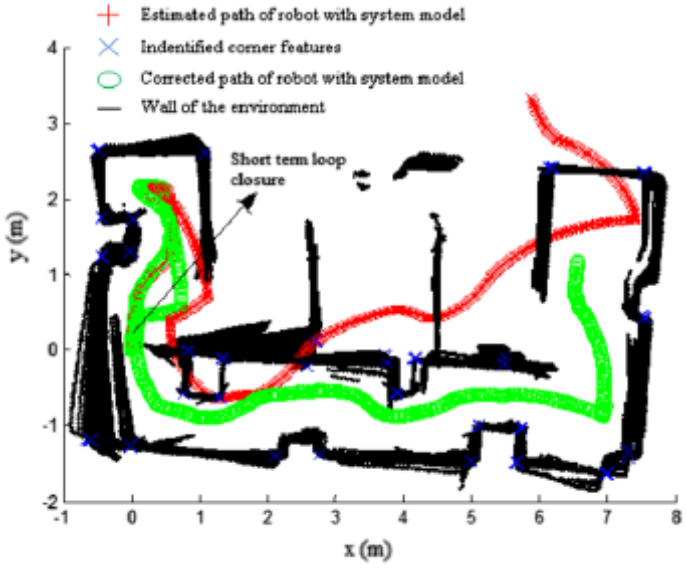

Figure 2.10: Generated map using EKF SLAM with the robot's trajectory (Santhanakrishnan *et al.*, 2017).

Chapter 3

Methodology

This chapter describes briefly the steps taken to accomplish the objectives mentioned in chapter 1.

To start, an autonomous car is needed to implement the SLAM algorithm. However, since we have a very limited budget, it needs to have the least possible specifications, so a GoPiGo 2 was chosen, which is shown in Figure 3.1.

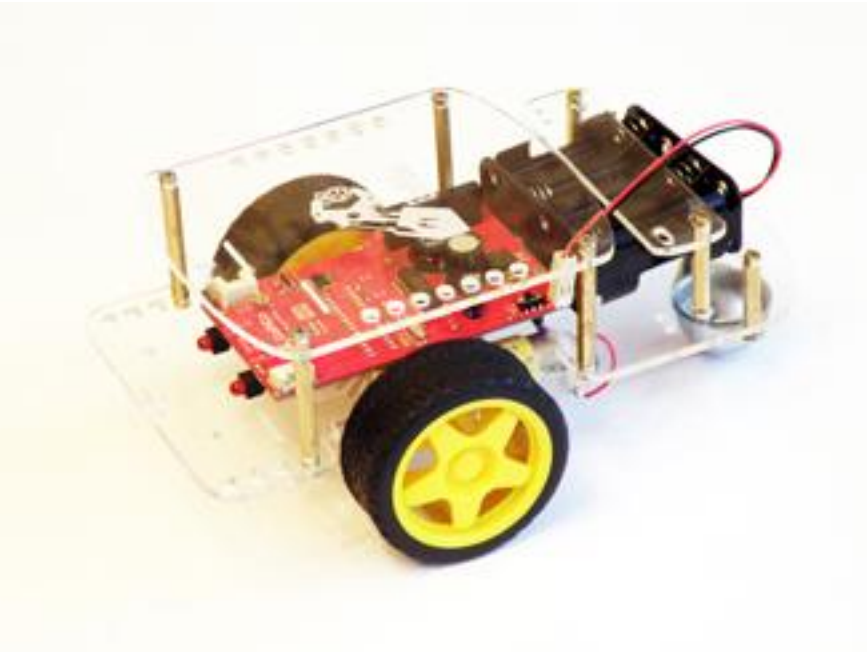

Figure 3.1: GoPiGo 2 autonomous car (Dexter Industries, 2018a)

Dexter Industries designed the GoPiGo to be compatible with the Raspberry Pi minicomputer, shown in Figure 3.2, so it was chosen to be the brain of the car.

To give the car the ability to avoid obstacles while autonomously navigating, an ultrasound sensor with a servo was installed, see Figure 3.3.

Finally, a laser sensor was installed to read the range and bearing of the car's surroundings (please see Figure 3.4).

As for the software that will control all of the previously mentioned hardware, Python 3 was selected to code the SLAM algorithm.

The upcoming chapters will elaborate more on how the hardware was used, and how the algorithm was implemented.

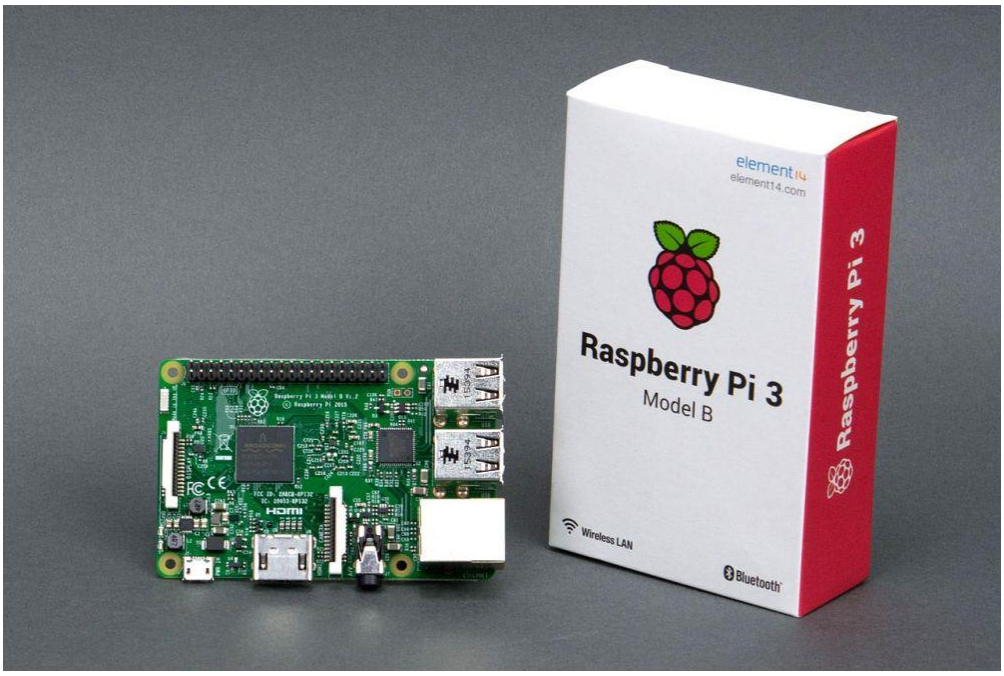

Figure 3.2: Raspberry Pi 3 minicomputer (Kiwi Electronics, 2018)

Figure 3.3: Ultrasound sensor installed on a servo (Dexter Industries, 2018b)

Figure 3.4: 360 degrees rpLiDAR (Robo Peak, 2018)

Chapter 4

Autonomous Car Setup

In this chapter, the autonomous car setup will be explained in more detail, showing step by step all the decisions made regarding the car itself, all added sensors, and the software installed. The chapter is divided into two sections. Section 4.1 explains the decisions made concerning the hardware, the assembly process, and the role of every sensor while section 4.2 discusses the software installed, and how it was utilised to operate the autonomous car.

4.1 The hardware

The hardware used in this study can be divided into three parts; the minicomputer, the sensors and the car that carries the aforesaid two parts. Starting from the last part, the upcoming subsections will explain all specifications, decisions, and alterations (if needed) made regarding each hardware part.

4.1.1 The Car

Since the budget for this study is very limited, the car used needed to have the following attributes:

1. Small in size, since bigger usually means more expensive.
2. Powerful enough to move around with ease, while carrying the minicomputer and the sensors.
3. Ability to provide the needed electrical power for itself, and all other parts to operate without power loss.
4. Low cost due to the limited budget.

Taking these attributes into account, GoPiGo 2 made by Dexter Industries was chosen.

This car is small in size with dimensions 14.5cm (W) x 21.5cm (L) x 9cm (H), yet big enough to carry the sensors and minicomputer. Its motors can handle the final weight of the car without any problems. Also, it can provide enough current (up to 2A) to power and operate all connected parts including the motors. Furthermore, in addition to all of that, being priced at around $100.00 US Dollars, it is relatively cheap.

Following the assembly instructions provided on the manufacturer's website, all the car's parts along with all sensors and the minicomputer were connected; to give the final assembled car shown in Figure 4.1.

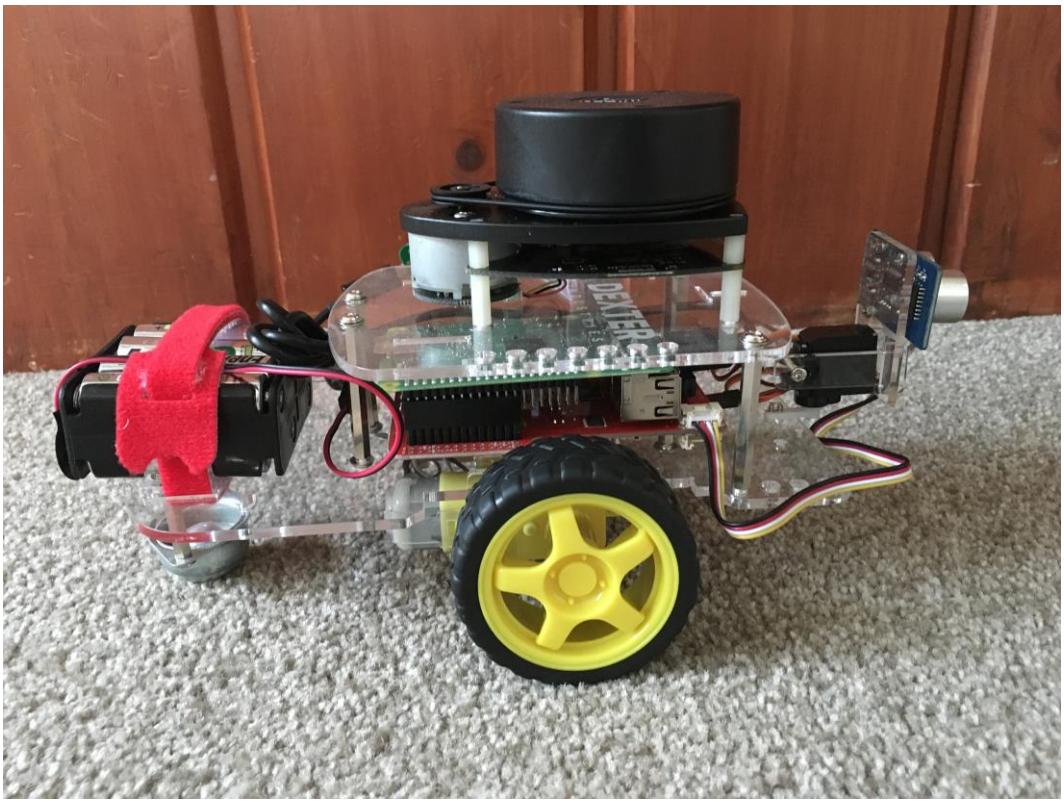

Figure 4.1: GoPiGo 2 manufactured by Dexter Industries, fully assembled with all needed sensors. The ultrasound sensor is installed on the servo facing forward (right of the figure), the laser reader is placed on top of the car, and the minicomputer is the green board below the laser reader.

The servo is attached to the front of the car, allowing the ultrasound sensor to face forward. The minicomputer (green board) is connected to the car's board (the red board next to the wheels, as shown in Figure 4.1). The laser reader placed on top of

the car is connected to the minicomputer via USB cable and a battery pack, consisting of eight AA rechargeable batteries, is placed in the back.

4.1.2 The Sensors

In order to implement the SLAM algorithm, and to allow the car to navigate while avoiding obstacles, the following sensors were needed:

1. Optical wheel encoders.
2. Ultrasound range sensor.
3. Laser range/bearing reader.

The following subsections will elaborate more on the specs and use of each of these sensors.

4.1.2.1 Optical Encoders

As shown in Figure 4.2, the car has two optical encoders located next to the wheels. These encoders are used to measure the distance travelled by each wheel separately, which is useful to check if the vehicle is moving forward when both distances are equal; or turning left/right if the distances are not equal.

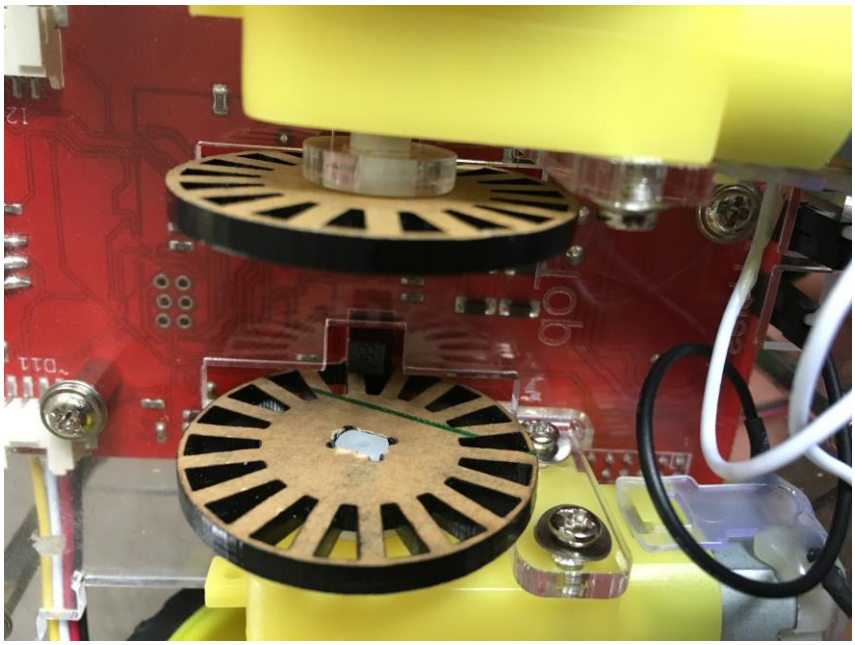

Figure 4.2: Two optical encoders installed next to the wheels. They are used to measure the distance travelled by each wheel separately.

With eighteen pulses per rotation, these encoders measure the distance travelled using equation **4.1**:

$$d = \frac{p * \phi_w}{18} \tag{4.1}$$

Where,

d is the distance travelled (m).

p is the count of pulses sensed by the optical encoder.

ϕ_w *is the diameter of the wheel (m).*

4.1.2.2 Ultrasound Range Sensor

This sensor as shown in Figure 4.3, can detect obstacles up to 350cm away from the vehicle. A programme that keeps reading the range feedback from this ultrasonic ranger, can detect a close obstacle up ahead and stops the car before crashing.

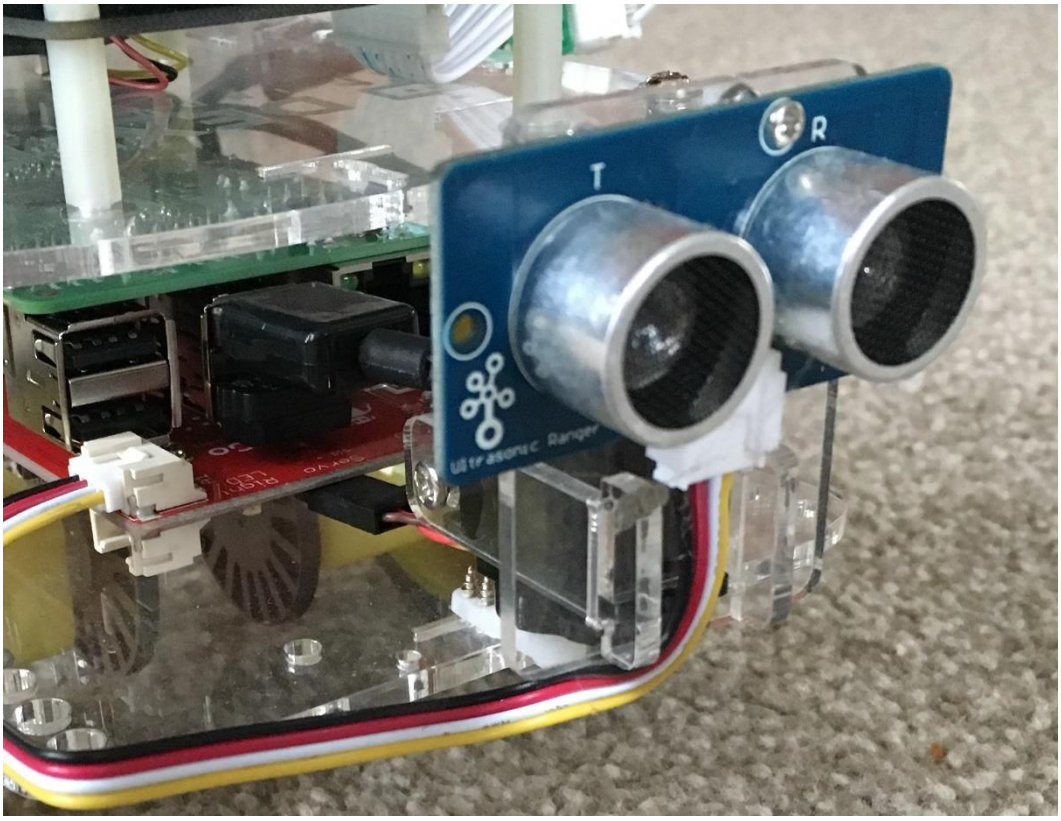

Figure 4.3: Ultrasound range sensor fixed to a servo

It is fixed to a servo to allow pointing it towards the left and the right. This helps the code to decide which side has more empty space and turn the car to avoid obstacles.

4.1.2.3 Laser Range/Bearing Reader

The sensor used in this study is an rpLiDAR A1 (Light Detection And Ranging) manufactured by Slamtec. It is a 360^{o} laser reader with a typical measurement range of 0.15m – 6m. It can scan the surroundings five and half times per second, reporting back 2000 or more measurements consisting of the distance and angle of all objects. See Figure 4.4.

Figure 4.4: rpLiDAR A1 from Slamtec

Using the range and bearing measurements received from this sensor, the code can detect all landmarks surrounding the vehicle; and draw a map of the environment.

4.1.3 The Minicomputer

This is the brain that will receive all input data, process all the information available, and give commands accordingly. It will hold the programme that controls the car, and execute the SLAM algorithm.

Therefore, it needs to have minimum specifications, such as:

1. Enough storage space for the algorithm code, and received readings.
2. Fast processing speed, to handle all the calculations while communicating with all sensors.
3. Wireless connection, to report back the result of the navigation.
4. Small size to fit onboard GoPiGo 2.
5. Compatibility with all other hardware components.

Raspberry Pi 2 was the best option available at the time, costing only 35 US Dollars. It has a built-in slot for micro-SD cards, which means the storage space available is flexible and can be chosen by the user. The micro-SD card used in this study has 8GB of data space, which was tested and found to be enough.

It has a Quad Cortex A7 processor with 900MHz processing speed, which was found to be fast enough to handle the code execution, data processing, and reporting back the results.

The lack of a wireless connection was solved by connecting a Wi-Fi dongle to one of the many USB ports available, and communication with the vehicle was achieved.

With dimensions of 8.5cm x 5.6cm, it clearly fits easily inside the GoPiGo 2. Also, Dexter Industries made this car to be completely compatible with Raspberry Pi, hence the ‘Pi’ in GoPiGo. Since all the sensors (except for the LiDAR) were purchased with the GoPiGo, they were also compatible with Raspberry Pi. However, the LiDAR needed some effort to integrate with the other parts, and this will be detailed in section 4.2.

4.2 The software

Any computer needs appropriate software to utilise the hardware and fulfil the purpose of its existence. In this study, two types of software were needed for the car to navigate, localise itself and report back a map of the surroundings (or execute the SLAM algorithm); the operating system or OS, and the programming language platform.

4.2.1 The Operating System (OS)

GoPiGo 2 comes with a micro-SD card pre-installed with Raspian Jessie for Robots. It is an operating system designed by Raspberry Foundation and modified by Dexter Industries to fully exploit the features of GoPiGo. It comes with all needed software to programme and operate GoPiGo, including communication protocol with the ultrasound and optical decoder sensors.

However, what this OS lacks is the ability to read data from the LiDAR, which is a disadvantage that forced the search for another OS. After a lot of research and installation/testing trials, the perfect OS was found.

Ubuntu Mate 16.04 is a Linux based OS compatible with Robot Operating System (ROS). The latter is a group of libraries that help to build robot applications (Open Source Robotics Foundation, 2018) such as this study. ROS Kinetic was chosen to be installed with Ubuntu Mate 16.04, which allowed the installation of a LiDAR driver called rplidar_ros. This driver permits the GoPiGo-LiDAR communication, and the LiDAR readings can now be retrieved through a command in a Linux terminal.

The final state of this OS has all needed software for controlling the GoPiGo, and at the same time, communicating with all attached sensors. The next step now would be programming the SLAM algorithm.

4.2.2 The Programming Language

There are a lot of different programming languages available, such as C, C++, C#, Java, Python …etc., each with its own strengths and characteristics. Choosing one of these languages depends on its simplicity, since a complex approach means more storage space for the code itself and more processing time, as well as its ability to control all of the hardware parts, such as GoPiGo, all sensors, and the Raspberry Pi. First, Dexter Industries provided a Python library called *Gopigo* which gives the user the ability to control the GoPiGo and all attached hardware (except for the LiDAR). Second, the Raspberry Pi has Ubuntu Mate installed, which is compatible with Python. Finally, Python has the ability to send commands to the Linux terminal within its syntax, which means communicating with the LiDAR and receiving the laser readings.

As a result, Python 3.5 was selected to be the programming language used in this study; for the strengths mentioned in the previous paragraph, and because it was the latest version of Python available when this study commenced.

4.3 Conclusion

This chapter explained in detail the steps taken to set up the autonomous car with all attached sensors, and all software installed. Also, it discussed the specifications of all hardware parts, and how they were chosen, as well as how all the software were selected and how they operated the connected hardware.

At this stage the car is ready to be programmed, so the next chapter will discuss the Extended Kalman Filter or EKF. This filter is one of many filters used to implement the SLAM algorithm, and it will be used in this study.

Chapter 5

Extended Kalman Filter SLAM

Using an Extended Kalman Filter (EKF) for estimating the state vector containing the robot pose (both location and orientation) and all landmark locations is still one of the most popular strategies for solving SLAM (Huang & Dissanayake, 2007). EKF is derived through a linearisation procedure, in which the nonlinear model describing the measurements is continuously linearised about each current state estimate (Jiang *et al.*, 2017), and the linear Kalman Filter is then applied to predict the next state (Einicke & White, 1999).

EKF was chosen for this study because of its simple and straight-forward implementation, and due to its good performance in small areas with a limited number of features (Darmanin & Bugeja, 2016).

This chapter is split into three sections. Section 5.1 explains the prediction step of EKF SLAM, and how the current state and landmark locations are predicted through the movement and measurement models. Section 5.2 elaborates on how the environment features or landmarks are selected. Then section 5.3 explains how the robot state (pose) and features locations are corrected using correction formulas.

5.1 Prediction

SLAM state is defined as the vehicle pose (location and direction) and the locations of observed stationary features (Bailey *et al.*, 2006). The prediction step of EKF SLAM predicts the state, and it consists of the movement or kinematic and measurement models (Santhanakrishnan *et al.*, 2017), (Klančar *et al.*, 2014), and (Wang *et al.*, 2013). The following subsections explain each model and give more detail on how they should be implemented in this study.

5.1.1 Kinematic Model

In order to obtain an estimate of the vehicle pose change between two consecutive time steps, the robot's odometry measurements received from the wheels encoders must be processed (Huang *et al.*, 2008). In addition, the horizontal distance between the wheels (car width) needs to be included in the pose prediction calculations (Teslić *et al.*, 2011).

The motion model that will predict the new vehicle's pose consists of two cases, when the robot is turning and when it is moving forward. Using the two given information, the current pose (x, y, θ) and the travelled distance of each wheel (l and r) as inputs from the wheels encoders, the kinematic model equations can be derived (Brenner, 2012). Figure 5.1 shows the case when the car is turning left around a rotation centre. It changed its position from point a to point b.

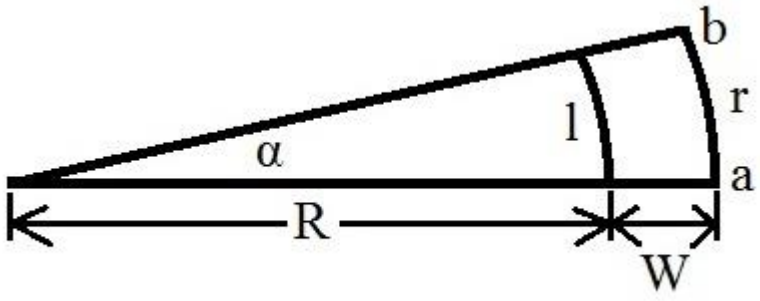

Figure 5.1: The car is turning left around a centre of rotation. Moving from point a to point b.

To solve the case in Figure 5.1, we need to find the unknowns R and α

$$r = \alpha (R + W) \tag{5.1}$$

Where,

r is the distance travelled by the right-hand wheel (m).

α is the rotation angle (rad).

R is the distance between the centre of rotation and the left-hand wheel (m).

W is the width of the vehicle (m).

$$l = \alpha R \tag{5.2}$$

Where,

l is the distance travelled by the left-hand wheel (m).

Subscribing equation (5.2) from equation (5.1)

$$r - l = \alpha W$$

$$\Rightarrow \alpha = \frac{r - l}{W} \tag{5.3}$$

Equation (5.2) can be rearranged to

$$R = \frac{l}{\alpha} \tag{5.4}$$

Clearly, α cannot be zero in equation (5.4), which is expected, since it will be zero only when the vehicle is moving forward (l = r).

Using the now known values of R and α, the new position can be predicted as shown in Figure 5.2

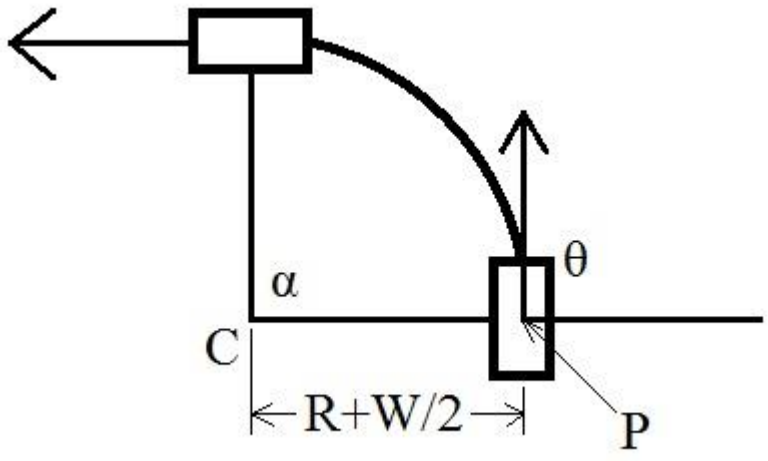

Figure 5.2: New pose prediction. C is the centre of rotation, P is the current robot's pose, and θ is the robot's current heading angle.

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \end{bmatrix} - (R + \frac{W}{2}) \begin{bmatrix} \sin\theta \\ -\cos\theta \end{bmatrix} \tag{5.5}$$

Where,

C_x and C_y are the x-y coordinates of the centre of rotation (m).

P_x and P_y are the x-y coordinates of the car's current pose (m).

θ is the car's current heading angle (rad).

After using equation (5.5) to calculate the x-y coordinates of the centre of rotation, the new heading angle can be predicted

$$\theta' = (\theta + \alpha) \bmod 2\pi \tag{5.6}$$

Where,

θ' is the new heading angle (rad).

Then the x-y coordinates of the new pose can also be predicted

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} C_x \\ C_y \end{bmatrix} + (R + \frac{W}{2}) \begin{bmatrix} \sin\theta' \\ -\cos\theta' \end{bmatrix} \tag{5.7}$$

Where,

P'_x and P'_y are the predicted x-y coordinates of the new pose (m).

Now for the case of moving forward, where l = r, the motion model in equations (5.6) and (5.7) become

$$\theta' = \theta \tag{5.8}$$

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \end{bmatrix} + l \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \tag{5.9}$$

Equation (5.8) is self-explanatory since the heading angle is not changing when the car is moving forward. In equation (5.9), l and r are exchangeable and either can be used since they are equal.

The algorithm coded in this study should read the distances travelled by left- and right-hand wheels, and compare their values. If they are not equal, equations (5.3) to (5.7) should be used to predict the new pose of the robot; otherwise, equations (5.8) and (5.9) should be executed (Brenner, 2012).

5.1.2 Measurement Model

The measurement model is responsible for locating all landmarks surrounding the vehicle, and giving their x-y coordinates with respect to the vehicle's location. It achieves this by processing the received measurements from the LiDAR (Brenner, 2012). Figure 5.3 shows a single landmark being observed by the LiDAR's laser rays.

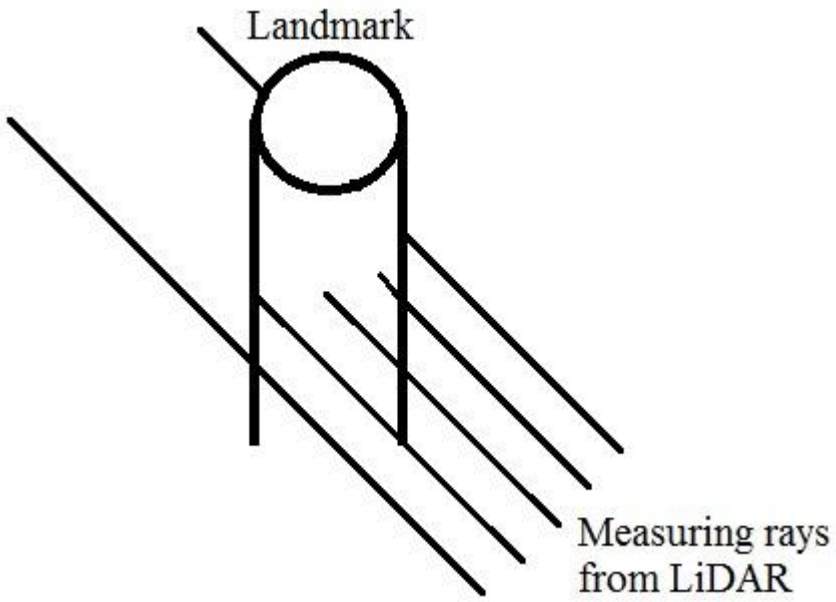

Figure 5.3: The LiDAR is detecting a landmark.

Features can be detected by calculating the derivatives of all received measurements

$$f'(i) = \frac{f(i+1) - f(i-1)}{2} \tag{5.10}$$

Where,

f'(i) is the derivative of the current measurement.

f(i+1) is the next measurement.

f(i-1) is the previous measurement.

Then the list of derivatives computed by equation (5.10) is searched for a negative value less than a threshold, starting from the first derivative. The found negative derivative represents the left edge of a landmark. As shown in Figure 5.3, the

measured depth of the second ray from the left is much less than the first ray, which makes the derivative negative. Now to locate the found feature, the average of rays' depths and angles are calculated by dividing the sum of each by the number of rays till we reach the right edge of that feature; which can be detected by a positive derivative larger than the threshold. Then, the Cartesian coordinates can be computed using equations (5.11) and (5.12).

$$x = d\cos\theta \tag{5.11}$$

$$y = d\sin\theta \tag{5.12}$$

Where,

x and y are the Cartesian coordinates of the found landmark (m).

d is the average depth of the found landmark (m).

θ is the average angle of the found landmark (rad).

The measurement model can be summarised in the algorithm shown in Figure 5.4.

```
for i in measurements
    if measurements(i-1) & measurements(i+1) ≥ min dist
        compute derivative
    else
        derivative = 0

for i in derivatives
    if derivatives(i) ≤ -depth threshold
        on landmark
    if on landmark
        if |derivatives(i)| ≤ depth threshold
            compute sum of depths and angles
        else if deriviatives(i) ≥ depth threshold
            off landmark
            compute average of depths and angles

for i in landmarks
    compute cartesian coordinates
```

Figure 5.4: Algorithm for the measurement model (Brenner, 2012).

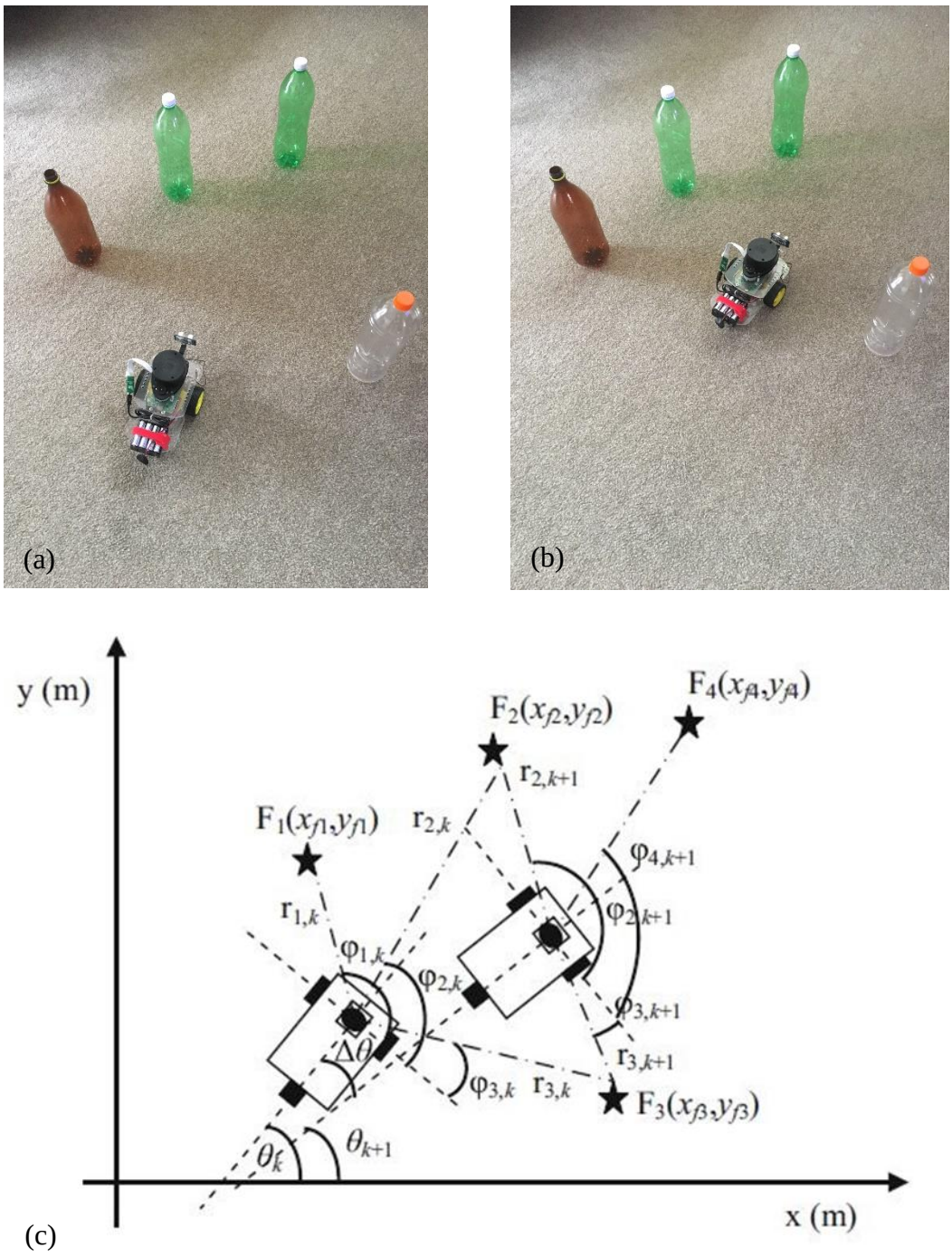

Figure 5.5: Illustration of movement model while the robot is moving. Figure (a) shows the robot's pose at time instant k and figure (b) shows the pose at time instant k+1. Figure (c) illustrates the movement model at poses (a) and (b) (Santhanakrishnan *et al.*, 2017).

Figure 5.5 illustrates the measurement model while the robot is moving. Where,

$F_i(x_{fi},y_{fi})$ is the global location of the i^{th} feature (m).

$r_{i,k}$ is the measured distance of the i^{th} feature from the robot at time instant k (m).

$\varphi_{i,k}$ is the measured bearing angle of the i^{th} feature with respect to the robot's local plane at time instant k (degrees).

x_k, y_k, and θ_k are the Cartesian coordinates and heading of the robot at time k.

5.2 Feature Selection

Feature selection step is responsible for deciding which found landmarks from the measurement model to be added to the state vector as unique and acceptable landmarks. It is an essential step before moving on to the correction step, since the accuracy of the pose correction depends on the accuracy of the detected landmarks not being duplicated or wrongfully located. One of the techniques used to achieve this is the Maximum Likelihood Landmark Assignment (Brenner, 2012) using the Mahalanobis distance (Santhanakrishnan *et al.*, 2017) and (Klančar *et al.*, 2014).

The Mahalanobis distance considers the correlation in the data, since it is calculated using the inverse of the variance-covariance matrix of the data set of interest (De Maesschalck *et al.*, 2000); In this case, the detected features. It is shown in equations (5.13) and (5.14) (Brenner, 2012).

$$\left(z - h(x)\right)^T \Psi^{-1} \left(z - h(x)\right) \leq \varepsilon \tag{5.13}$$

$$\Psi = H \bar{\Sigma} H^T + Q \tag{5.14}$$

Where,

z is the measured landmark depth and bearing (m, rad).

h(x) is the depth and bearing of the x^{th} added landmark in the state vector (m, rad).

Ψ^{-1} is the inverse of the covariance matrix of the added landmark.

ε is a threshold to compare the Mahalanobis distance to (m).

H and H^T are the Jacobian and its transverse of the added landmark.

$\bar{\Sigma}$ is the covariance of the state.

Q is the measurement error.

The left-hand side of equation (5.13) represents the Mahalanobis distance, and the right-hand side threshold can be found by a set of trials.

The usage of the Mahalanobis distance can be explained in Figure 5.6. Cases (a) and (b) show two found landmarks near to two added landmarks in the state vector. If the distance between the found and added features was solely taken into consideration in feature selection, then both features would be added to the state vector. But the error (covariance) ellipse of landmark (b) is big enough to include the found nearby landmark, which indicates that the found and added landmarks might be the same landmark and should not be added to the state vector. So the feature selection algorithm based on the Mahalanobis distance would add feature (a) to the state vector, but not feature (b). Less added features to the state vector will reduce the computational overhead of the entire algorithm, and the Mahalanobis distance validation process will eliminate the probability of wrong feature selection (Santhanakrishnan *et al.*, 2017).

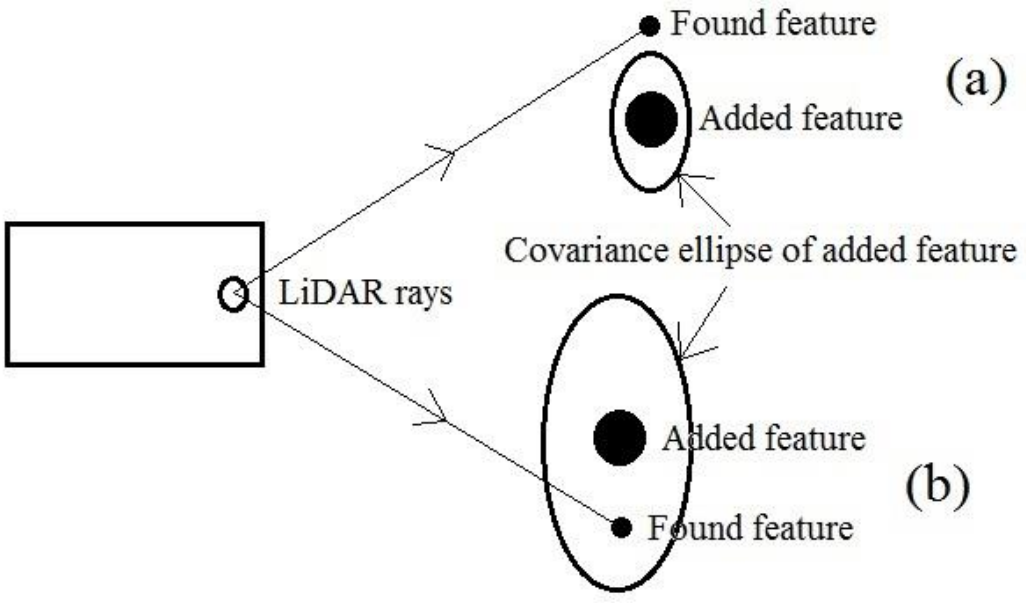

Figure 5.6: The Mahalanobis Distance (Brenner, 2012).

5.3 Correction

The implementation of EKF-SLAM starts in the correction step. A slight modification to equations (5.5) to (5.7) is needed, to ignore the centre of rotation and make the calculations simpler (Brenner, 2012).

$$g(x,y,\theta,l,r)=\begin{bmatrix}x\\y\\\theta\end{bmatrix}+\begin{bmatrix}(R+\frac{W}{2})(\sin(\theta+\alpha)-\sin\theta)\\(R+\frac{W}{2})(-\cos(\theta+\alpha)+\cos\theta)\\\alpha\end{bmatrix}=\begin{bmatrix}g_1\\g_2\\g_3\end{bmatrix} \quad (5.15)$$

Where,

g_1 and g_2 are the predicted xy-coordinates of the new pose (m).

g_3 is the predicted bearing of the new pose (rad).

Equation (5.15) predicts the new pose when the car is turning, i.e. l is not equal to r. When the vehicle is moving forward, equations (5.8) and (5.9) can be used to calculate g_1, g_2, and g_3.

Now, the Jacobian matrix of the state (G) can be calculated by the partial derivative of g in respect to the state (Brenner, 2012). In case of l not equal to r:

$$G=\begin{bmatrix}\frac{\partial g_1}{\partial x} & \frac{\partial g_1}{\partial y} & \frac{\partial g_1}{\partial \theta}\\\frac{\partial g_2}{\partial x} & \frac{\partial g_2}{\partial y} & \frac{\partial g_2}{\partial \theta}\\\frac{\partial g_3}{\partial x} & \frac{\partial g_3}{\partial y} & \frac{\partial g_3}{\partial \theta}\end{bmatrix}=\begin{bmatrix}1 & 0 & (R+\frac{W}{2})(\cos(\theta+\alpha)-\cos\theta)\\0 & 1 & (R+\frac{W}{2})(\sin(\theta+\alpha)-\sin\theta)\\0 & 0 & 1\end{bmatrix} \quad (5.16)$$

When the robot is moving forward:

$$G=\begin{bmatrix}1 & 0 & -l\sin\theta\\0 & 1 & l\cos\theta\\0 & 0 & 1\end{bmatrix} \quad (5.17)$$

In equation (5.17), r can be used instead of l since both are equal. This applies to all equations related to the case of l equals r.

Next, V will be calculated, which is the Jacobian matrix of the state g with respect to the control (l and r) (Brenner, 2012). In the case of l not equal to r:

$$V = \begin{bmatrix} \frac{\partial g_1}{\partial l} & \frac{\partial g_1}{\partial r} \\ \frac{\partial g_2}{\partial l} & \frac{\partial g_2}{\partial r} \\ \frac{\partial g_3}{\partial l} & \frac{\partial g_3}{\partial r} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{Wr}{(r-l)^2}(\sin\theta' - \sin\theta) - \frac{r+l}{2(r-l)}\cos\theta' & \frac{-Wl}{(r-l)^2}(\sin\theta' - \sin\theta) + \frac{r+l}{2(r-l)}\cos\theta' \\ \frac{Wr}{(r-l)^2}(\cos\theta - \cos\theta') - \frac{r+l}{2(r-l)}\sin\theta' & \frac{-Wl}{(r-l)^2}(\cos\theta - \cos\theta') + \frac{r+l}{2(r-l)}\sin\theta' \\ -\frac{1}{W} & \frac{1}{W} \end{bmatrix}$$

(5.18)

Where,

$\theta' = \theta + \alpha$

For the case of l equals r:

$$V = \begin{bmatrix} \frac{1}{2}(\cos\theta + \frac{l}{W}\sin\theta) & \frac{1}{2}(\cos\theta - \frac{l}{W}\sin\theta) \\ \frac{1}{2}(\sin\theta - \frac{l}{W}\cos\theta) & \frac{1}{2}(\sin\theta + \frac{l}{W}\cos\theta) \\ -\frac{1}{W} & \frac{1}{W} \end{bmatrix} \quad (5.19)$$

Assuming that the left and right motors' controls are uncorrelated, the covariance of the control ($\Sigma_{control}$) is calculated using equation (5.20) (Brenner, 2012).

$$\Sigma_{control} = \begin{bmatrix} \sigma_l^2 & 0 \\ 0 & \sigma_r^2 \end{bmatrix} = \begin{bmatrix} (\alpha_1 l)^2 + (\alpha_2 (l-r))^2 & 0 \\ 0 & (\alpha_1 r)^2 + (\alpha_2 (l-r))^2 \end{bmatrix} \quad (5.20)$$

And when l equals r

$$\sigma_l^2 = \sigma_r^2 = (\alpha_1 l)^2 \quad (5.21)$$

Where,

σ_l and σ_r are the errors corresponding to the left and right motors respectively.

α_1 is the control's motion error factor.

α_2 is the control's turn error factor.

The predicted covariance of the new pose can now be calculated using equation (5.22) (Brenner, 2012).

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + V_t \Sigma_{control} V_t^T \tag{5.22}$$

Where,

t and t-1 are the time instants at the current and previous pose respectively.

T is the transverse of the matrix.

The added landmarks in the state vector are represented in the Cartesian coordinates, and they need to be translated to the polar coordinates before they can be used to correct the predicted pose. This can be achieved using equations (5.23) and (5.24) (Brenner, 2012).

$$r = \sqrt{(x_m - x_l)^2 + (y_m - y_l)^2} \tag{5.23}$$

$$\alpha = \arctan\left(\frac{y_m - y_l}{x_m - x_l}\right) - \theta \tag{5.24}$$

Where,

$x_l = x + d\cos\theta$, $y_l = y + d\sin\theta$

r is the distance between the landmark and the LiDAR (m).

x_m and y_m are the xy-coordinates of the landmark's location (m).

x_l and y_l are the xy-coordinates of the robot's current location (m).

d is the displacement of the LiDAR from the centre of the robot (m).

α is the bearing of the landmark (rad).

θ is the robot's current heading (rad).

Now the Jacobian matrix of the measurement (equations (5.23) and (5.24)) is calculated with respect to the state (Brenner, 2012)

$$H = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial \alpha}{\partial x} \\ \frac{\partial r}{\partial y} & \frac{\partial \alpha}{\partial y} \\ \frac{\partial r}{\partial \theta} & \frac{\partial \alpha}{\partial \theta} \end{bmatrix} = \begin{bmatrix} \frac{-\Delta x}{\sqrt{q}} & \frac{\Delta y}{q} \\ \frac{-\Delta y}{\sqrt{q}} & \frac{-\Delta x}{q} \\ \frac{d}{\sqrt{q}}(\Delta x \sin\theta - \Delta y \cos\theta) & -\frac{d}{q}(\Delta x \cos\theta + \Delta y \sin\theta) - 1 \end{bmatrix} \quad (5.25)$$

Where,

$\Delta x = x_m - x_l$, $\Delta y = y_m - y_l$, $q = (x_m - x_l)^2 + (y_m - y_l)^2$

Next, the Kalman gain (K) can be computed using equation (5.26) (Brenner, 2012).

$$K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q)^{-1} \quad (5.26)$$

Where,

$$Q = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\alpha^2 \end{bmatrix} \quad (5.27)$$

$\bar{\Sigma}_t$ *is the predicted covariance matrix of the state at time instant t.*

Q is the measurement covariance matrix.

σ_r^2 *is the measurement distance standard deviation.*

σ_α^2 *is the measurement angle standard deviation.*

The predicted state and its predicted covariance can now be corrected using equations (5.28) and (5.29) (Brenner, 2012).

$$\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t)) \quad (5.28)$$

$$\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t \quad (5.29)$$

Where,

μ_t *and* Σ_t *are the corrected state and covariance at time instant t.*

$\bar{\mu}_t$ *is the predicted state at time instant t.*

z_t *is the measured landmark in polar coordinates at time instant t.*

h( ) is the translator from Cartesian to polar coordinates.

I is the identity matrix.

Up until now, no found landmarks have been considered in all the matrices in equations (5.15) onward. But since the state vector will get larger with added features while the robot is navigating, these landmarks need to be considered starting from equation (5.15) (Brenner, 2012). Assuming only two landmarks were added:

$$g(x, y, \theta, x_1, y_1, x_2, y_2, l, r) = \begin{bmatrix} x \\ y \\ \theta \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} * \\ * \\ * \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.30}$$

Where,

** is to be replaced with the previous formulas from previous equations.*

Similarly, equations (5.16) and (5.17) become

$$G = \begin{bmatrix} 1 & 0 & * & 0 & 0 & 0 & 0 \\ 0 & 1 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.31}$$

Noting that adding landmarks to the state only makes g and G matrices bigger without changing the used formulas. Also, the added items are either a zero or a one. The total number of rows and columns for the G matrix can be calculated using equation (5.32) (Brenner, 2012).

$$\#\,of\ \ rows\ \ or\ \ columns\ = 2 * number\ \ of\ \ added\ \ landmarks\ + 3 \tag{5.32}$$

V matrix in equations (5.18) and (5.19) is modified exactly like the G matrix in equation (5.31), except that the matrix is filled with zeros only (Brenner, 2012).

When adding a landmark, its xy-coordinates need to be added to the state vector and its measurement covariance needs to be added to the state covariance; filling any empty items in the state covariance matrix with zeros (Brenner, 2012). The size of the state covariance matrix can be calculated in a similar manner to the G and V matrices using equation (5.32).

The H matrix is modified in a different manner. For the first added landmark, the negative of the first two rows and columns of the original H matrix will be appended to the end of the matrix to give the H matrix used with the correction equations (5.28) and (5.29). When dealing with the second landmark, zeros need to be appended first to represent the first landmark, then repeat the procedure done for the first landmark. And so on for the next landmarks (Brenner, 2012). Figure 5.7 illustrates the procedure for modifying the H matrix depending on the index of the particular landmark.

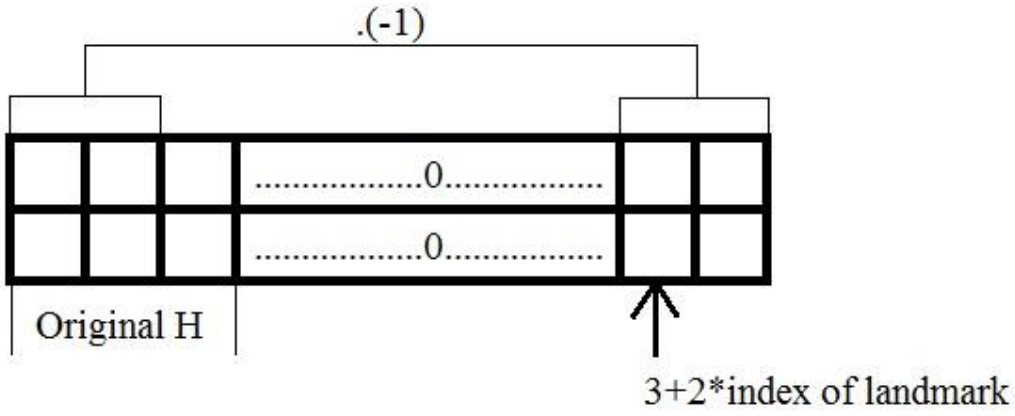

Figure 5.7: Modification of H matrix, depending on the index of the particular landmark (Brenner, 2012).

Chapter 6

Results and Discussion

This chapter elaborates on the implementation of the theory explained in Chapter 5, then shows the experimental results obtained after several trials and discusses these results. It is divided into two sections: Section 6.1 explains the algorithm used to programme the robot, and Section 6.2 discusses the experimental results after several trials.

6.1 The Algorithm

Chapter 5 explained two different cases to be considered when predicting and correcting the car's pose. The equations used in the second case when the vehicle is moving forward are much simpler than the equations for the first case when the vehicle is turning, hence smaller code and less needed computational power. This simplicity leaned this study towards making the robot move forward only and turn only after stopping the robot, which will only change the heading.

This section will explain the main parts of the algorithm separately, and how to use the equations of Chapter 5 in each of them. Then, it gathers them in the main code algorithm, sorted in the order of execution. First, Figure 6.1 shows the algorithm for the prediction step.

Second, Figure 5.4 shows the algorithm for the measurement model. Third, the algorithm for the correction step is shown in Figure 6.2. Finally, the EKF SLAM algorithm is shown in Figure 6.3. The final and complete python code can be found in Appendix A.

Prediction Algorithm (predict)

Compute G using equation (5.17)

Compute control covariance using equation (5.21)

Compute V using equation (5.19)

Predict the covariance of the new pose using equation (5.22)

Predict the new pose using equations (5.8) and (5.9)

Figure 6.1: Prediction Algorithm (predict).

Correction Algorithm (correct)

Compute H using equation (5.25)

Compute Q using equation (5.27)

Compute K using equation (5.26)

Correct the predicted pose using equation (5.28)

Correct the predicted pose covariance using equation (5.29)

Figure 6.2: The Correction Algorithm (correct).

```
move forward
predict
if ultrasound ≤ dist_to_stop
    stop
    if right is clear
        turn right
        θ' = θ + 90°
    else
        turn left
        θ' = θ − 90°
    measuremen t
    for all found landmarks
        if not in added_landmarks and mahalanobi s_dist ≤ threshold
            add found landmark
        else
            get landmark index
        correct
repeat
```

Figure 6.3: The EKF SLAM Algorithm.

6.2 Experimental Results

After programming the robot with the python code shown in Appendix A, and making the robot navigate for fifty one-minute-navigation trials, the predicted final pose had a ± 300mm deviation from the actual pose; in both x and y coordinates.

In each of these trials, the code was executed using Linux terminal. A prompt menu appears asking the user to input a character, as shown in Figure 6.4. Inputting ‘r’ will start the navigation and the EKF SLAM algorithm, typing ‘t’ will terminate the programme and show the map and all the car’s poses, while any other character will pause the navigation and put the robot on hold waiting for the next command.

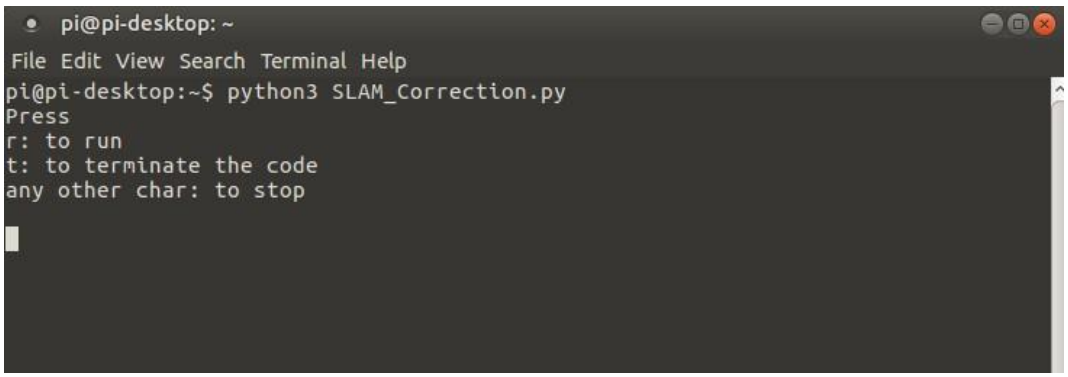

Figure 6.4: The Linux terminal window showing the prompt menu.

After navigating for about a minute then typing ‘t’, a map appears showing the path travelled by the robot, the walls of the room where the navigation occurred, and the added landmarks. The first navigation trial is divided into three segments, starting from the position shown in Figure 6.5 with the initial map shown in Figure 6.6, through to the position in Figure 6.7 with the map shown in Figure 6.8, and moving to the second pose in Figure 6.9 and map in Figure 6.10. Figure 6.11 shows the final actual robot’s position and Figure 6.12 is the map resulted from the final navigation segment. The maps show the room’s walls as green dots, the added landmarks as red circles, and the robot’s travelled path as blue dots with the last pose shown as a black dot.

The tilted walls in the map shown in Figure 6.12 are due to the correction applied only to the car’s heading in the state, not to the actual physical heading. Also, when the LiDAR’s scan data is drawn, the next set of wall dots will be tilted from the previous set, which explains the tilted walls in the second and final maps.

The drawn map’s coordinates are in millimetres, and the map represents the world from the robot’s point of view. The robot’s initial pose is always at the origin (0, 0) Cartesian coordinates, while heading towards the positive x axes (zero degrees). The robot moves forward until it faces an obstacle and stops, then it looks left and right to find the clearest path and turns towards it. The LiDAR measurements are

processed to draw the walls and find landmarks while the car is stationary; after the processing is done, the car moves forward again and so on. The map and poses are only shown after the termination of the programme.

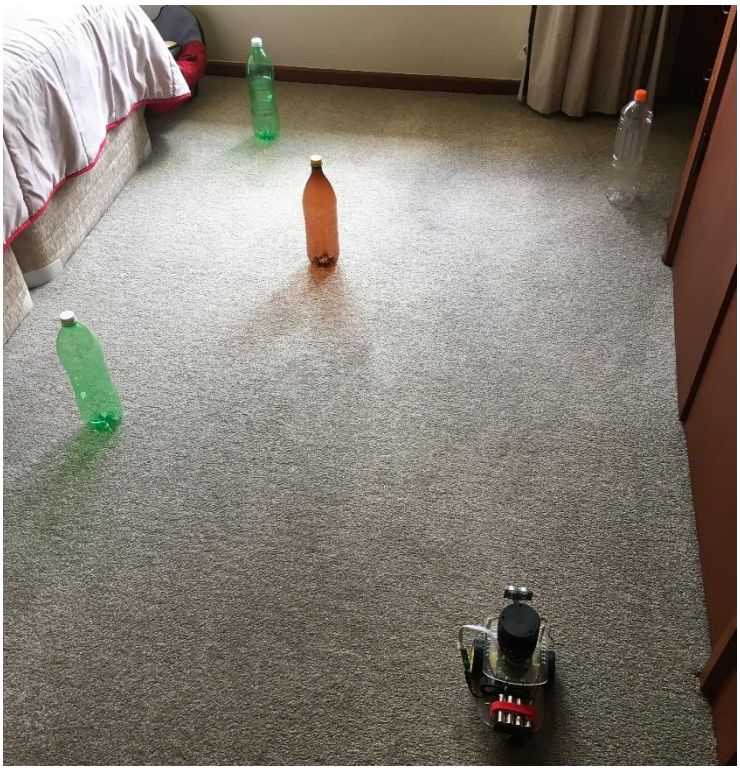

Figure 6.5: The robot's initial position. The room consists of walls and landmarks (bottles).

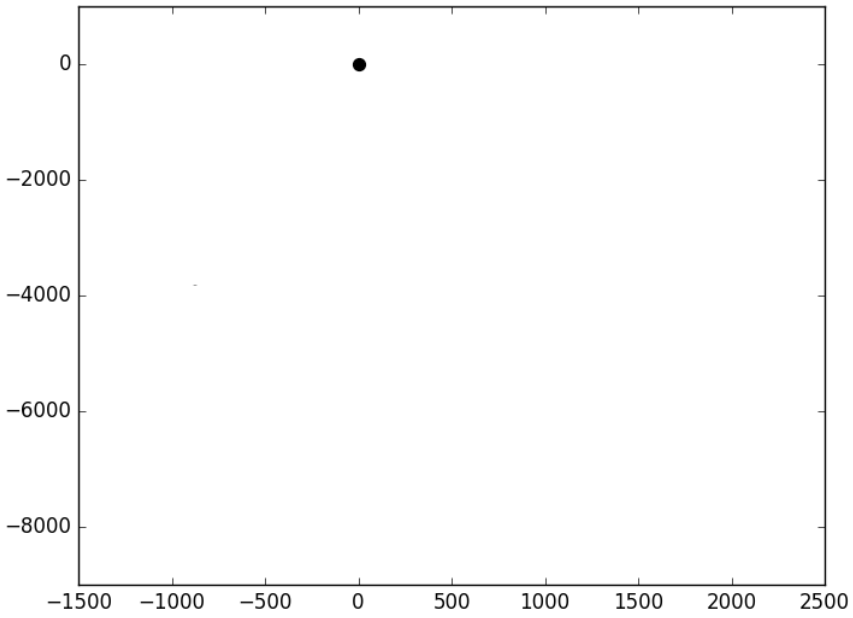

Figure 6.6: The initial map showing only the start position at (0, 0).

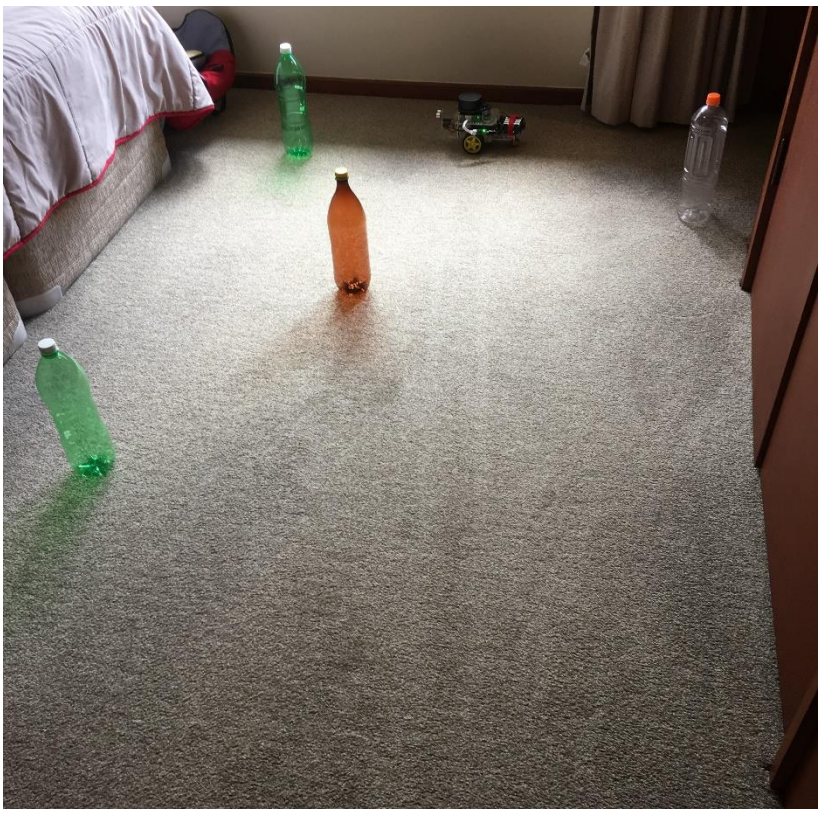

Figure 6.7: The robot's position after the first navigation segment.

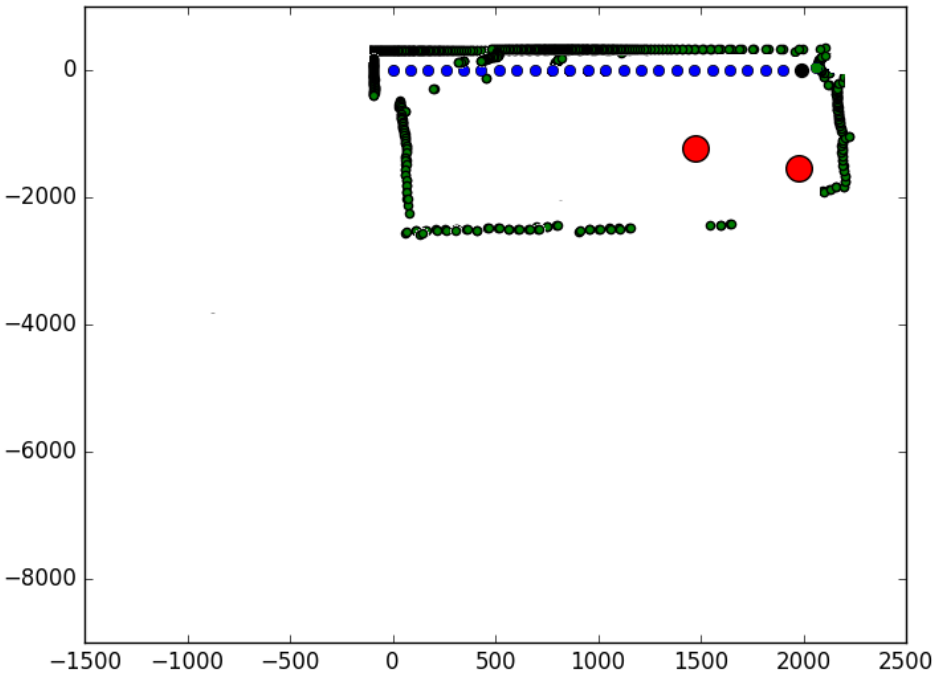

Figure 6.8: The resulting map after the first navigation segment. The green dots represent the walls, the red circles are the added landmarks, and the blue dots are the travelled path with the last pose shown as a black dot.

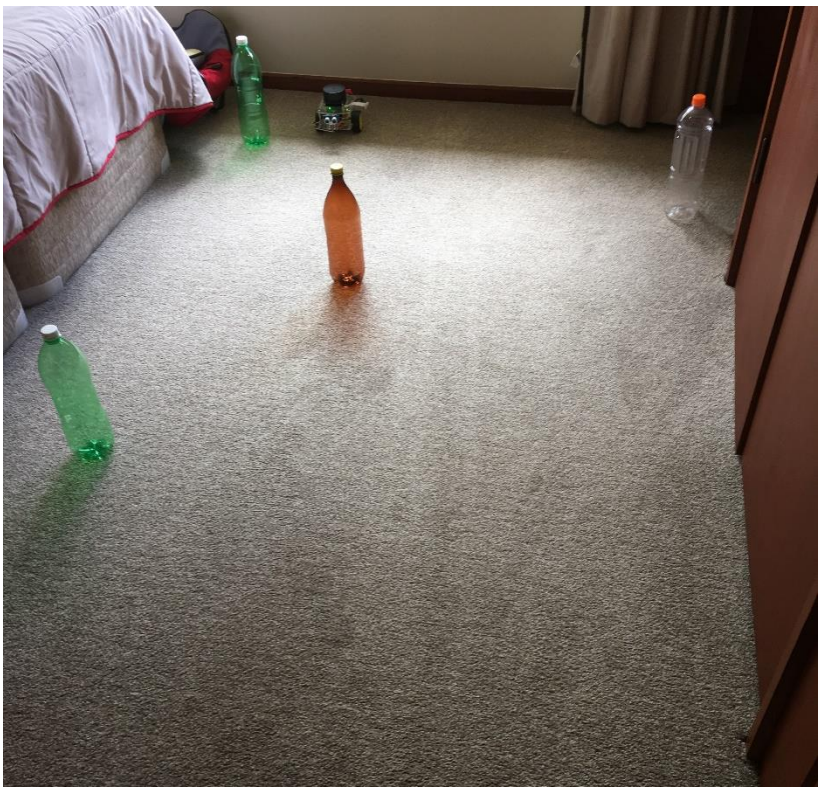

Figure 6.9: The robot's position after the second navigation segment.

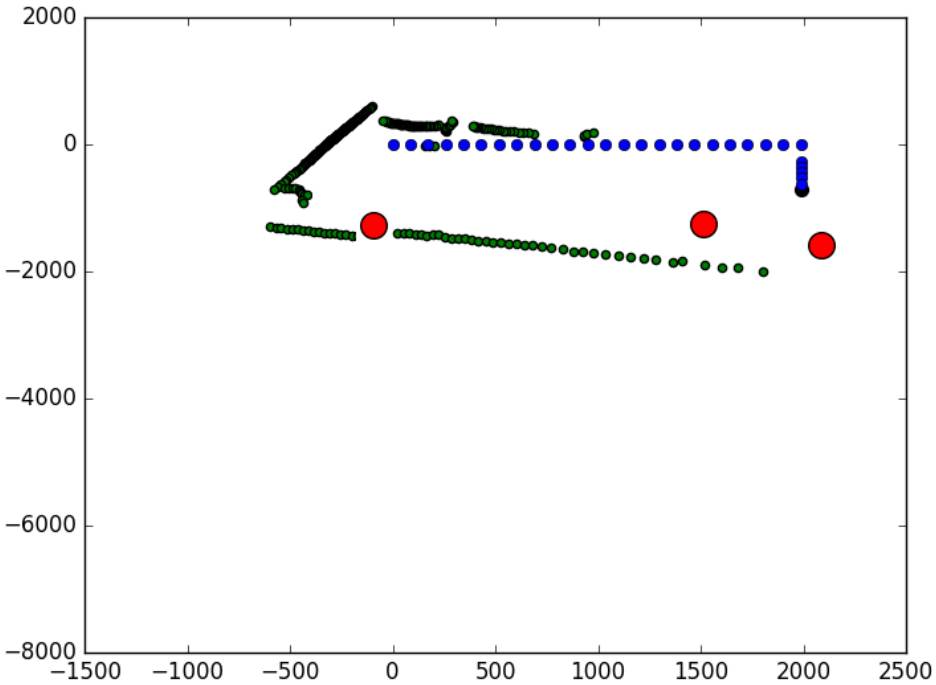

Figure 6.10: The resulting map after the second navigation segment.

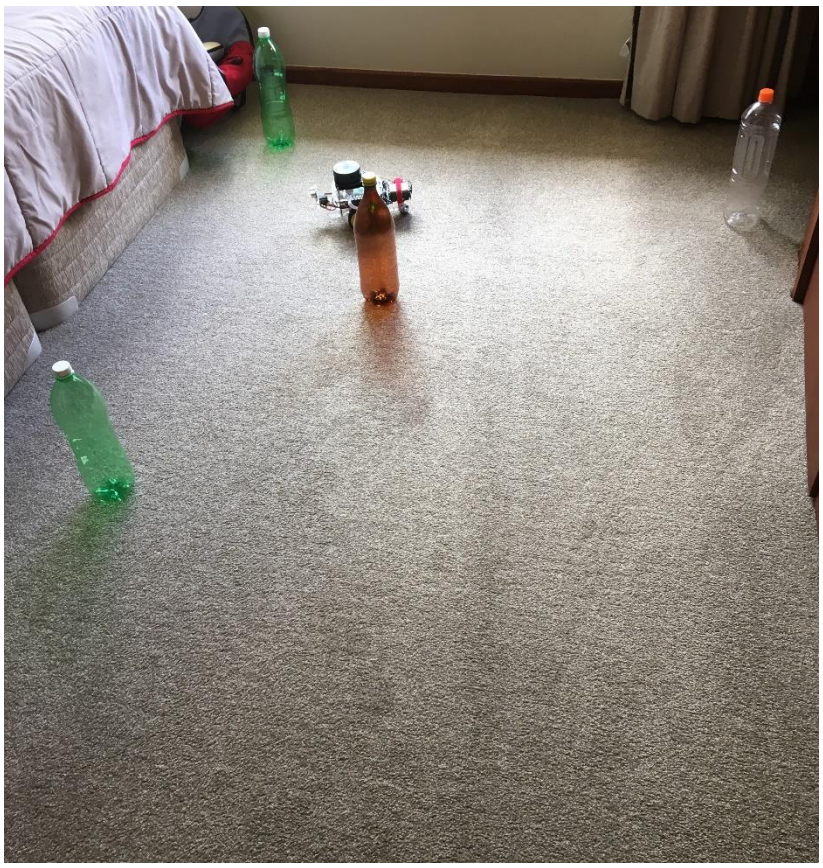

Figure 6.11: The robot's position after the final navigation segment.

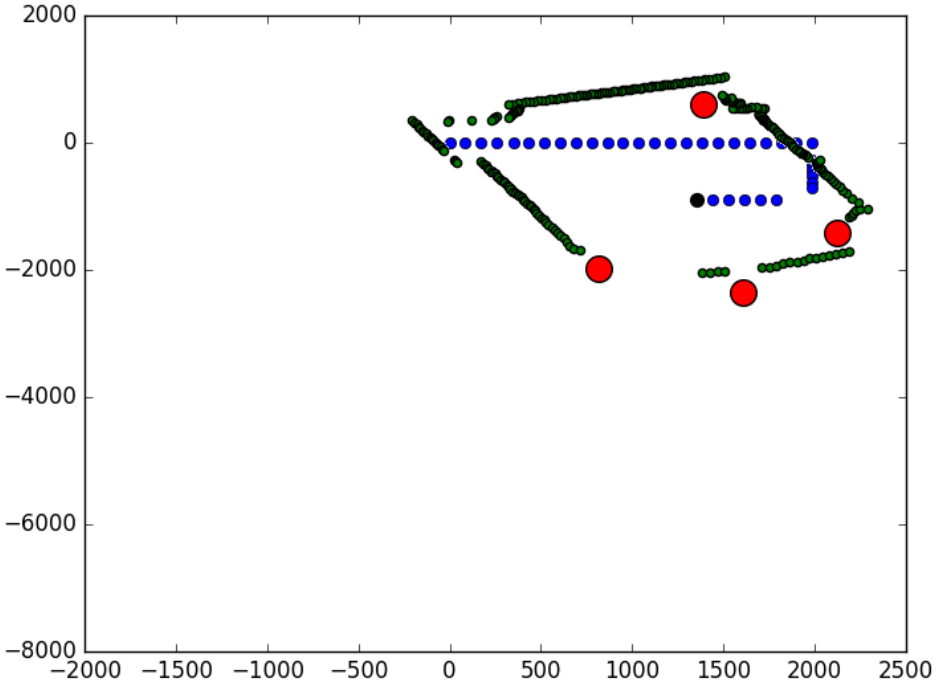

Figure 6.12: The resulting map after the final navigation segment.

Chapter 7

Conclusion and Recommendations

To enable a mobile robot to navigate in an unknown environment, while avoiding obstacles and reporting back its current location and a map of that environment, this study presented an algorithm based on EKF SLAM. This algorithm was programmed in a mobile robot to test it. According to the tests done per the autonomous mobile car, the main objective has been achieved. The vehicle moves around freely (unmanned) without a prior map, and reports back its location and a map of the surroundings after it finishes navigation. This could not be achieved without achieving the specific objectives. Therefore, the Raspberry Pi was programmed to help successfully operate a GoPiGo robot, a Python code was successfully written and compiled to receive inputs from the input devices and trigger the output devices, and the Raspberry Pi, GoPiGo robot, ultra sound sensor, and LiDAR were interfaced with an EKF SLAM algorithm.

In the future, this algorithm can be tuned to produce a more accurate current location and environment map. Another type of measuring device can be added to or replace the currently used LiDAR, to allow receiving measurement data simultaneously while navigating and avoiding obstacles. Also, implementing this algorithm in a real-life application to test its performance under real-life pressures and obstacles.

References

Aulinas, J., Petillot, Y. R., Salvi, J., & Lladó, X. (2008). The SLAM problem: a survey. *CCIA*, *184*(1), 363-371.

Ayache, N., & Faugeras, O. D. (1988). Building, registrating, and fusing noisy visual maps. *The International Journal of Robotics Research*, *7*(6), 45-65.

Bailey, T., & Durrant-Whyte, H. (2006). Simultaneous localization and mapping (SLAM): Part II. *IEEE Robotics & Automation Magazine*, *13*(3), 108-117.

Bailey, T., Nieto, J., Guivant, J., Stevens, M., & Nebot, E. (2006) Consistency of the EKF-SLAM algorithm. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* (pp. 3562-3568): IEEE.

Bar-Shalom, Y., Li, X. R., & Kirubarajan, T. (2004). *Estimation with applications to tracking and navigation: theory algorithms and software*. John Wiley & Sons.

Berg, M. (2013). *Navigation with Simultaneous Localization and Mapping: For Indoor Mobile Robot.* thesis, Institutt for teknisk kybernetikk.

Bosse, M., Newman, P., Leonard, J., & Teller, S. (2004). Simultaneous localization and map building in large-scale cyclic environments using the Atlas framework. *The International Journal of Robotics Research*, *23*(12), 1113-1139.

Bradley, C., Albu, A. B., Arai, K., Erasmus, R., Johnson, J., & Smith, C. (2013). Object Detection and Mapping of an Outdoor Field for UVic IGVT.

Brenner, C. (2012). *SLAM Lectures*. Retrieved 14 Jan, 2018, from https://www.youtube.com/watch?v=HPeBhArNpzY&list=PLpUPoM7Rgzi_7YWn14Va2FODh7LzADBSm&index=4.

Burgard, W., Fox, D., Jans, H., Matenar, C., & Thrun, S. (1999) Sonar-based mapping with mobile robots using EM. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-* (pp. 67-76): MORGAN KAUFMANN PUBLISHERS, INC.

Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., & Leonard, J. J. (2016). Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, *32*(6), 1309-1332.

Carlone, L., Ng, M. K., Du, J., Bona, B., & Indri, M. (2010) Rao-Blackwellized particle filters multi robot SLAM with unknown initial correspondences and limited communication. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on* (pp. 243-249): IEEE.

Castellanos, J. A., Martínez, J. M., Neira, J., & Tardós, J. D. (1998). Experiments in multisensor mobile robot localization and map building. *IFAC Proceedings Volumes*, *31*(3), 369-374.

Charabaruk, N. (2015). *Development of an autonomous omnidirectional hazardous material handling robot.* thesis, University of Ontario Institute of Technology (Canada).

Chatila, R., & Laumond, J.-P. (1985) Position referencing and consistent world modeling for mobile robots. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on* (Vol. 2, pp. 138-145): IEEE.

Chatterjee, A. (2009). Differential evolution tuned fuzzy supervisor adapted extended Kalman filtering for SLAM problems in mobile robots. *Robotica, 27*(3), 411-423.

Chellali, R. (2013) A distributed multi robot SLAM system for environment learning. In *Robotic Intelligence In Informationally Structured Space (RiiSS), 2013 IEEE Workshop on* (pp. 82-88): IEEE.

Chen, Z., Samarabandu, J., & Rodrigo, R. (2007). Recent advances in simultaneous localization and map-building using computer vision. *Advanced Robotics, 21*(3-4), 233-265.

Chong, K. S., & Kleeman, L. (1999). Feature-based mapping in real, large scale environments using an ultrasonic array. *The International Journal of Robotics Research, 18*(1), 3-19.

Civera, J., Grasa, O. G., Davison, A. J., & Montiel, J. (2010). 1-Point RANSAC for EKF Filtering: Application to Real-Time Structure from Motion and Visual dometry. *Journal of Field Robotics*.

Clemente, L. A., Davison, A. J., Reid, I. D., Neira, J., & Tardós, J. D. (2007) Mapping Large Loops with a Single Hand-Held Camera. In *Robotics: Science and Systems* (Vol. 2).

Colon, E., & Pécsi, R. (2013). Mobile Robots: Localization and 3D Mapping.

Crowley, J. L. (1989) World modeling and position estimation for a mobile robot using ultrasonic ranging. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on* (pp. 674-680): IEEE.

Csorba, M. (1998). *Simultaneous localisation and map building.* thesis, University of Oxford.

Csorba, M., Uhlmann, J. K., & Durrant-Whyte, H. F. (1996) New approach to simultaneous localization and dynamic map building. In *Navigation and Control Technologies for Unmanned Systems* (Vol. 2738, pp. 26-37): International Society for Optics and Photonics.

Darmanin, R. N., & Bugeja, M. K. (2016) Autonomous Exploration and Mapping using a Mobile Robot Running ROS. In *ICINCO (2)* (pp. 208-215).

Darmstadt, H., Kohlbrecher, S., Meyer, J., Graber, T., Kurowski, K., von Stryk, O., & Klingauf, U. (Compiler) (2014). *Robocuprescue 2014-robot league team*: RoboCupRescue.

Davison, A. J., Cid, Y. G., & Kita, N. (2004). Real-time 3D SLAM with wide-angle vision. *IFAC Proceedings Volumes, 37*(8), 868-873.

Davison, A. J., & Murray, D. W. (2002). Simultaneous localization and map-building using active vision. *IEEE transactions on pattern analysis and machine intelligence, 24*(7), 865-880.

De Maesschalck, R., Jouan-Rimbaud, D., & Massart, D. L. (2000). The mahalanobis distance. *Chemometrics and intelligent laboratory systems, 50*(1), 1-18.

Deans, M., & Hebert, M. (2001). Experimental comparison of techniques for localization and mapping using a bearing-only sensor. In *Experimental Robotics VII* (pp. 395-404). Springer.

Dellaert, F., & Kaess, M. (2006). Square Root SAM: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research, 25*(12), 1181-1203.

Dexter Industries. (2018a). *GoPiGo 2*. Retrieved 03 Jan, 2018, from https://raw.githubusercontent.com/DexterInd/GoPiGo/master/GoPiGo_Chassis-300.jpg.

Dexter Industries. (2018b). *Ultrasound Sensor*. Retrieved 03 Jan, 2018, from https://www.dexterindustries.com/GoPiGo/projects/python-examples-for-the-raspberry-pi/gopigo-robot-raspberry-pi-ultrasonic-sensor-example/.

Dissanayake, G., Huang, S., Wang, Z., & Ranasinghe, R. (2011) A review of recent developments in simultaneous localization and mapping. In *Industrial and Information Systems (ICIIS), 2011 6th IEEE International Conference on* (pp. 477-482): IEEE.

Dissanayake, M. G., Newman, P., Clark, S., Durrant-Whyte, H. F., & Csorba, M. (2001). A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Transactions on robotics and automation, 17*(3), 229-241.

Dissanayake, M. G., Newman, P., Durrant-Whyte, H. F., Clark, S., & Csorba, M. (2000). An experimental and theoretical investigation into simultaneous localisation and map building. In *Experimental robotics VI* (pp. 265-274). Springer.

Doucet, A. (1998). On sequential simulation-based methods for Bayesian filtering.

Dryanovski, I., Valenti, R. G., & Xiao, J. (2013) Fast visual odometry and mapping from RGB-D data. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on* (pp. 2305-2310): IEEE.

Durrant-Whyte, H., & Bailey, T. (2006). Simultaneous localization and mapping: part I. *IEEE robotics & automation magazine, 13*(2), 99-110.

Durrant-Whyte, H., Rye, D., & Nebot, E. (1996). Localization of autonomous guided vehicles. In *Robotics Research* (pp. 613-625). Springer.

Durrant-Whyte, H. F. (1988). Uncertain geometry in robotics. *IEEE Journal on Robotics and Automation, 4*(1), 23-31.

Durrant-Whyte, H. F. (1996). An autonomous guided vehicle for cargo handling applications. *The International Journal of Robotics Research, 15*(5), 407-440.

Einicke, G. A., & White, L. B. (1999). Robust extended Kalman filtering. *IEEE Transactions on Signal Processing, 47*(9), 2596-2599.

Folkesson, J., & Christensen, H. (2004) Graphical SLAM-a self-correcting map. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on* (Vol. 1, pp. 383-390): IEEE.

Frese, U., Larsson, P., & Duckett, T. (2005). A multilevel relaxation algorithm for simultaneous localization and mapping. *IEEE Transactions on Robotics, 21*(2), 196-207.

Gan, H. (2015). *Multi-sensor guidance of an agricultural robot.* thesis.

Grisetti, G., Kummerle, R., Stachniss, C., & Burgard, W. (2010). A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine, 2*(4), 31-43.

Grisetti, G., Stachniss, C., & Burgard, W. (2009). Nonlinear constraint network optimization for efficient map learning. *IEEE Transactions on Intelligent Transportation Systems, 10*(3), 428-439.

Guivant, J. E., & Nebot, E. M. (2001). Optimization of the simultaneous localization and map-building algorithm for real-time implementation. *IEEE transactions on robotics and automation, 17*(3), 242-257.

Howard, A., Sukhatme, G. S., & Mataric, M. J. (2006). Multirobot simultaneous localization and mapping using manifold representations. *Proceedings of the IEEE, 94*(7), 1360-1369.

Huang, G. P., Mourikis, A. I., & Roumeliotis, S. I. (2008) Analysis and improvement of the consistency of extended Kalman filter based SLAM. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on* (pp. 473-479): IEEE.

Huang, S., & Dissanayake, G. (2007). Convergence and consistency analysis for extended Kalman filter based SLAM. *IEEE Transactions on robotics, 23*(5), 1036-1049.

Huang, S., & Dissanayake, G. (2016). A critique of current developments in simultaneous localization and mapping. *International Journal of Advanced Robotic Systems, 13*(5), 1729881416669482.

Indelman, V., Nelson, E., Michael, N., & Dellaert, F. (2014) Multi-robot pose graph localization and data association from unknown initial relative poses via expectation maximization. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (pp. 593-600): IEEE.

Jensfelt, P., Kragic, D., Folkesson, J., & Bjorkman, M. (2006) A framework for vision based bearing only 3D SLAM. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on* (pp. 1944-1950): IEEE.

Jiang, Y., Huang, Y., Xue, W., & Fang, H. (2017). On designing consistent extended Kalman filter. *Journal of Systems Science and Complexity, 30*(4), 751-764.

Kaess, M., Ranganathan, A., & Dellaert, F. (2007) iSAM: Fast incremental smoothing and mapping with efficient data association. In *Robotics and Automation, 2007 IEEE International Conference on* (pp. 1670-1677): IEEE.

Khairuddin, A. R., Talib, M. S., & Haron, H. (2015) Review on simultaneous localization and mapping (SLAM). In *Control System, Computing and Engineering (ICCSCE), 2015 IEEE International Conference on* (pp. 85-90): IEEE.

Kim, J.-H., & Sukkarieh, S. (2003) Airborne simultaneous localisation and map building. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on* (Vol. 1, pp. 406-411): IEEE.

Kiwi Electronics. (2018). *Raspberry Pi 3*. Retrieved 03 Jan, 2018, from https://www.kiwi-electronics.nl/raspberry-pi-3-model-b.

Klančar, G., Teslić, L., & Škrjanc, I. (2014). Mobile-robot pose estimation and environment mapping using an extended Kalman filter. *International Journal of Systems Science, 45*(12), 2603-2618.

Konolige, K., Grisetti, G., Kümmerle, R., Burgard, W., Limketkai, B., & Vincent, R. (2010) Efficient sparse pose adjustment for 2D mapping. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on* (pp. 22-29): IEEE.

Lachlan, G., & Krishnan, T. (1997). The EM algorithm and extensions. *Probability and Statistics*.

Leonard, J., & Durrant-Whyte, H. (Compiler) (1992). *Direct sonar navigation*: Kluwer Academic Press, London.

Leonard, J., & Newman, P. (2003) Consistent, convergent, and constant-time SLAM. In *IJCAI* (pp. 1143-1150).

Leonard, J. J., & Feder, H. J. S. (2000). A computationally efficient method for large-scale concurrent mapping and localization. In *Robotics Research* (pp. 169-176). Springer.

Leonard, J. J., Rikoski, R. J., Newman, P. M., & Bosse, M. (2002). Mapping partially observable features from multiple uncertain vantage points. *The International Journal of Robotics Research, 21*(10-11), 943-975.

Lu, F., & Milios, E. (1997). Globally consistent range scan alignment for environment mapping. *Autonomous robots, 4*(4), 333-349.

Mazuran, M., Burgard, W., & Tipaldi, G. D. (2016). Nonlinear factor recovery for long-term SLAM. *The International Journal of Robotics Research, 35*(1-3), 50-72.

Mei, C., Sibley, G., Cummins, M., Newman, P., & Reid, I. (2011). RSLAM: A system for large-scale mapping in constant-time using stereo. *International journal of computer vision, 94*(2), 198-214.

Michael, M., Sebastian, T., Daphne, K., & Ben, W. (2003) Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)* (Vol. 2).

Ministry for Culture and Heritage. (2011). *PGC Building Collapse*. Retrieved 27 Jan, 2018, from https://nzhistory.govt.nz/media/photo/pgc-building-collapse.

Montemerlo, M., Thrun, S., Koller, D., & Wegbreit, B. (2002). FastSLAM: A factored solution to the simultaneous localization and mapping problem. *Aaai/iaai, 593598*.

Murphy, K. P. (2000) Bayesian map learning in dynamic environments. In *Advances in Neural Information Processing Systems* (pp. 1015-1021).

Newcombe, R. A., Fox, D., & Seitz, S. M. (2015) Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 343-352).

Newman, P., & Leonard, J. (2003) Pure range-only sub-sea SLAM. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on* (Vol. 2, pp. 1921-1926): Ieee.

Newman, P., Leonard, J., Tardós, J. D., & Neira, J. (2002) Explore and return: Experimental validation of real-time concurrent mapping and localization. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on* (Vol. 2, pp. 1802-1809): IEEE.

NIWA. (2012). *Divers discover new to science species down in one of the deepest flooded caves in the world*. Retrieved 27 January, 2018, from https://www.niwa.co.nz/news/divers-discover-new-to-science-species-down-in-one-of-the-deepest-flooded-caves-in-the-world.

Nourbakhsh, I. R., Bobenage, J., Grange, S., Lutz, R., Meyer, R., & Soto, A. (1999). An affective mobile robot educator with a full-time job. *Artificial Intelligence, 114*(1-2), 95-124.

Open Source Robotics Foundation. (2018). Retrieved 08 Jan, 2018, from www.ros.org.

Pathak, K., Birk, A., Vaskevicius, N., Pfingsthorn, M., Schwertfeger, S., & Poppinga, J. (2010). Online three- dimensional SLAM by registration of large planar surface segments and closed- form pose- graph relaxation. *Journal of Field Robotics, 27*(1), 52-84.

Pujals, S.-A. G. (2014). Autonomous Exploration with a Humanoid Robot.

Radio New Zealand. (2016). *NZ Defence Force to get $20bn upgrade*. Retrieved 27 Jan, 2018, from https://www.radionz.co.nz/news/political/305878/nz-defence-force-to-get-$20bn-upgrade.

Rencken, W. D. (1993) Concurrent localisation and map building for mobile robots using ultrasonic sensors. In *Intelligent Robots and Systems' 93, IROS'93. Proceedings of the 1993 IEEE/RSJ International Conference on* (Vol. 3, pp. 2192-2197): IEEE.

Robo Peak. (2018). *LiDAR*. Retrieved 03 Jan, 2018, from http://www.ransen.com/pointor/How-to-convert-2D-LIDAR-files-into-DXF-files.htm.

Rodriguez-Losada, D., Matia, F., Jimenez, A., & Galán, R. (2006) Consistency improvement for SLAM-EKF for indoor environments. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on* (pp. 418-423): IEEE.

Saeedi, S., Paull, L., Trentini, M., & Li, H. (2011). Neural network-based multiple robot simultaneous localization and mapping. *IEEE Transactions on Neural Networks, 22*(12), 2376-2387.

Saeedi, S., Paull, L., Trentini, M., Seto, M., & Li, H. (2014). Map merging for multiple robots using hough peak matching. *Robotics and Autonomous Systems, 62*(10), 1408-1424.

Saeedi, S., Trentini, M., Seto, M., & Li, H. (2016). Multiple- Robot Simultaneous Localization and Mapping: A Review. *Journal of Field Robotics, 33*(1), 3-46.

Salas-Moreno, R. F., Newcombe, R. A., Strasdat, H., Kelly, P. H., & Davison, A. J. (2013) Slam++: Simultaneous localisation and mapping at the level of objects. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on* (pp. 1352-1359): IEEE.

Santhanakrishnan, M. N., Rayappan, J. B. B., & Kannan, R. (2017). Implementation of extended Kalman filter-based simultaneous localization and mapping: a point feature approach. *Sādhanā, 42*(9), 1495-1504.

Scaramuzza, D., & Fraundorfer, F. (2011). Visual odometry [tutorial]. *IEEE robotics & automation magazine, 18*(4), 80-92.

Se, S., Lowe, D., & Little, J. (2002). Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *The international Journal of robotics Research, 21*(8), 735-758.

Segal, A., Haehnel, D., & Thrun, S. (2009) Generalized-icp. In *Robotics: science and systems* (Vol. 2, pp. 435).

Smith, R., Self, M., & Cheeseman, P. (1990). Estimating uncertain spatial relationships in robotics. In *Autonomous robot vehicles* (pp. 167-193). Springer.

Smith, R. C., & Cheeseman, P. (1986). On the representation and estimation of spatial uncertainty. *The international journal of Robotics Research, 5*(4), 56-68.

Song, T. L. (1996). Observability of target tracking with bearings-only measurements. *IEEE Transactions on Aerospace and Electronic Systems, 32*(4), 1468-1472.

Song, T. L. (1999). Observability of target tracking with range-only measurements. *IEEE Journal of Oceanic Engineering, 24*(3), 383-387.

Strasdat, H., Montiel, J., & Davison, A. J. (2010). Scale drift-aware large scale monocular SLAM. *Robotics: Science and Systems VI, 2*.

Sturm, J., Bylow, E., Kerl, C., Kahl, F., & Cremer, D. (2013). Dense tracking and mapping with a quadrocopter. *Unmanned Aerial Vehicle in Geomatics (UAV-g), Rostock, Germany*.

Swarbrick, N. (2015). *National parks - Upper South Island parks*. Retrieved 27 January, 2018, from http://www.TeAra.govt.nz/en/photograph/14445/entrance-to-bulmer-cavern.

Teslić, L., Škrjanc, I., & Klančar, G. (2011). EKF-based localization of a wheeled mobile robot in structured environments. *Journal of Intelligent & Robotic Systems, 62*(2), 187-203.

Thrun, S. (Compiler) (2001). *Using EM to Learn 3D Models with Mobile Robots*.

Thrun, S. (2002). Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium, 1*, 1-35.

Thrun, S., Beetz, M., Bennewitz, M., Burgard, W., Cremers, A. B., Dellaert, F., Fox, D., Haehnel, D., Rosenberg, C., & Roy, N. (2000a). Probabilistic algorithms and the interactive museum tour-guide robot minerva. *The International Journal of Robotics Research, 19*(11), 972-999.

Thrun, S., Burgard, W., & Fox, D. (1998). A probabilistic approach to concurrent mapping and localization for mobile robots. *Autonomous Robots, 5*(3-4), 253-271.

Thrun, S., Burgard, W., & Fox, D. (2000b) A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on* (Vol. 1, pp. 321-328): IEEE.

Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic robotics*. MIT press.

Thrun, S., & Liu, Y. (2005) Multi-robot SLAM with sparse extended information filers. In *Robotics Research. The Eleventh International Symposium* (pp. 254-266): Springer.

Tobata, Y., Kurazume, R., Noda, Y., Lingemann, K., Iwashita, Y., & Hasegawa, T. (2012). Laser-based geometrical modeling of large-scale architectural

structures using co-operative multiple robots. *Autonomous Robots, 32*(1), 49-62.

Vidal-Calleja, T. A., Berger, C., Solà, J., & Lacroix, S. (2011). Large scale multiple robot visual mapping with heterogeneous landmarks in semi-structured terrain. *Robotics and Autonomous Systems, 59*(9), 654-674.

Wang, D., Liang, H., Mei, T., Zhu, H., Fu, J., & Tao, X. (2013) Lidar Scan matching EKF-SLAM using the differential model of vehicle motion. In *Intelligent Vehicles Symposium (IV), 2013 IEEE* (pp. 908-912): IEEE.

Weingarten, J., & Siegwart, R. (2006) 3D SLAM using planar segments. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* (pp. 3062-3067): IEEE.

Williams, S. B., Newman, P., Dissanayake, G., & Durrant-Whyte, H. (2000) Autonomous underwater simultaneous localisation and map building. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on* (Vol. 2, pp. 1793-1798): IEEE.

Wyeth, G., & Milford, M. (2009). Spatial cognition for robots. *IEEE Robotics & Automation Magazine, 16*(3).

Appendix A

EKF SLAM Python 3 Code

Figure A.1 to Figure A.12 show the complete Python 3 code implemented in the robot.

```
#!/usr/bin/python2

#########################################################################
# This code tells GoPiGo to navigate using the Ultrasonic sensor, and report back its current position and a map of the environment after it finishes navigation
#
# In this code, the GoPiGo moves forward starting from origin (0,0) and heading towards positive x direction (zero degrees) and keeps reading from the ultrasonic
# sensor while predicting its current location on the xy plain, and when it is close to an obstacle, the servo turns the Ultrasonic sensor right and left to decide
# which side to turn, then it turns to the side with more empty space. Before moving forward again, the measurement data received from the LiDAR is processed and used
# to find and add landmarks to the state vector. Those added landmarks will then be used to correct the predicted pose. Then it moves forward and the previous code
# is repeated until it is terminated by the user. Then a map pops up on the screen showing the travelled path, the added landmarks, and the room's walls
#
# For simplicity, GoPiGo moves only forward and turns left/right by 90 degrees (no arc movement)
#
#
# Author                Date
# Hisham Abdel Qader    01 Jan 18
#
#########################################################################
#
# ! Attach Ultrasonic sensor to A1 Port.
#
#########################################################################

import sys
import select as uselect
import tty
import termios
from gopigo import *
import time
from math import sin, cos, pi, atan2, sqrt
from matplotlib.pyplot import *
from numpy import *
import os

class ExtendedKalmanFilter:     # This class contains all parts of the EKF SLAM Algorithm
    def __init__(self, state, covariance,
                             robot_width,
                             control_motion_factor, ticks_to_mm, measurement_distance_stddev, measurement_angle_stddev):
            # The state. This is the core data of the Kalman filter.
            self.state = state
            self.covariance = covariance

            # Some constants.
            self.robot_width = robot_width
```

Figure A.1: EKF SLAM Python 3 code (part 1).

```
        self.control_motion_factor = control_motion_factor
        self.ticks_to_mm = ticks_to_mm
        self.measurement_distance_stddev = measurement_distance_stddev
        self.measurement_angle_stddev = measurement_angle_stddev

        # Currently, the number of landmarks is zero.
        self.number_of_landmarks = 0

    @staticmethod
    def g(state, motor_ticks, ticks_to_mm, number_of_landmarks):        # This updates the state vector
        land_marks = []
        x, y, theta = state[0:3]
        if len(state) > 3:          # If there are added landmarks already in the state vector, then save them
                land_marks.extend(state[3:2*number_of_landmarks+3])
        l = motor_ticks
        if sin(theta*pi/180) < 0.5 and sin(theta*pi/180) > -0.5:             # If moving in x axis direction (both positive and negative)
                        g1 = x + (l * ticks_to_mm * cos(theta*pi/180))       # Update x location
                        g2 = y
        else:
                        g2 = y + (l * ticks_to_mm * sin(theta*pi/180))       # Else, update y location
                        g1 = x
        g3 = theta              #no change since GoPiGo is moving forward

        s = list([g1, g2, g3])
        s.extend(land_marks)
        return s

    @staticmethod
    def dg_dstate(state, motor_ticks, ticks_to_mm):     # The partial derivative of g with respect to state
        theta = state[2]
        l = motor_ticks
        dg1_dtheta = -l * ticks_to_mm * sin(theta*pi/180)
        dg2_dtheta = l * ticks_to_mm * cos(theta*pi/180)
        m = array([[1, 0, dg1_dtheta], [0, 1, dg2_dtheta], [0, 0, 1]])

        return m

    @staticmethod
    def dg_dcontrol(state, motor_ticks, w):     # The partial derivative of g with respect to control
        theta = state[2]
        l = motor_ticks
        dg1_dl = (cos(theta*pi/180) + ((l / w) * sin(theta*pi/180))) / 2
        dg2_dl = (sin(theta*pi/180) - ((l / w) * cos(theta*pi/180))) / 2
```

Figure A.2: EKF SLAM Python 3 code (part 2).

```
        dg3_dl = -1 / w

        dg1_dr = (cos(theta*pi/180) - ((l / w) * sin(theta*pi/180))) / 2
        dg2_dr = (sin(theta*pi/180) + ((l / w) * cos(theta*pi/180))) / 2
        dg3_dr = 1 / w
        m = array([[dg1_dl, dg1_dr], [dg2_dl, dg2_dr], [dg3_dl, dg3_dr]])

        return m

    def predict(self, motor_ticks):
        """The prediction step of the Extended Kalman Filter."""
        # covariance' = G * covariance * GT + R
        # where R = V * (covariance in control space) * VT.
        # Covariance in control space depends on move distance.
        # Compute G using dg_dstate
        G3 = self.dg_dstate(self.state, motor_ticks, self.ticks_to_mm)

        # Compute the covariance in control space
        left = motor_ticks
        left_var = (self.control_motion_factor * left * self.ticks_to_mm)**2
        control_covariance = diag([left_var, left_var])

        # Then, compute V using dg_dcontrol
        V = self.dg_dcontrol(self.state, motor_ticks, self.robot_width)
        R3 = dot(V, dot(control_covariance, V.T))

        # Modify G and R to accomodate any added landmarks
        x = (self.number_of_landmarks * 2) + 3
        i = []
        for y in range(x):
            i.append(1)
        G = diag(i)
        G[0:3,0:3] = G3
        R = zeros((3+(2*self.number_of_landmarks),3+(2*self.number_of_landmarks)))
        R[0:3,0:3] = R3

        # Then, compute the new self.covariance
        self.covariance = dot(G, dot(self.covariance, G.T)) + R

        # And predict the new self.state
        self.state = self.g(self.state, motor_ticks, self.ticks_to_mm, self.number_of_landmarks)
```

Figure A.3: EKF SLAM Python 3 code (part 3).

```
def add_landmark_to_state(self, initial_coords):
        """Enlarges the current state and covariance matrix to include one more
           landmark, which is given by its initial_coords (an (x, y) tuple).
           Returns the index of the newly added landmark."""

        self.number_of_landmarks+=1     # Incrementing the number of landmarks

        # Initialize the state with the given initial_coords and the covariance with 1e10 (as an approximation for "infinity")
        s = []
        s.extend(self.state)
        s.extend(initial_coords)
        self.state = s

        c = []
        c = zeros((2*self.number_of_landmarks+3, 2*self.number_of_landmarks+3))
        c[0:2*(self.number_of_landmarks-1)+3,0:2*(self.number_of_landmarks-1)+3] = self.covariance[0:2*(self.number_of_landmarks-1)+3,0:2*(self.number_of_landmark
        c[2*(self.number_of_landmarks-1)+3, 2*(self.number_of_landmarks-1)+3] = 1e10
        c[2*(self.number_of_landmarks-1)+4, 2*(self.number_of_landmarks-1)+4] = 1e10
        self.covariance = c

        return self.number_of_landmarks-1  # Return the index of the added landmark

@staticmethod
def h(state, landmark):
        """Takes an (x, y, theta) state and an (x, y) landmark, and returns the
           measurement (range, bearing)."""
        dx = landmark[0] - state[0]
        dy = landmark[1] - state[1]
        r = sqrt(dx * dx + dy * dy)
        alpha = (atan2(dy, dx) - state[2] + pi) % (2*pi) - pi

        return array([r, alpha])

@staticmethod
def dh_dstate(state, landmark):     # The partial derivative of h with respect to state

        x, y, theta = state[0:3]
        x_m, y_m = landmark
        x_l = x
        y_l = y
        q = ((x_m - x_l) * (x_m - x_l)) + ((y_m - y_l) * (y_m - y_l))
```

Figure A.4: EKF SLAM Python 3 code (part 4).

```
        dr_dx = -(x_m - x_l) / sqrt(q)
        dr_dy = -(y_m - y_l) / sqrt(q)
        dr_dtheta = 0

        dalpha_dx = (y_m - y_l) / q
        dalpha_dy = -(x_m - x_l) / q
        dalpha_dtheta = -1

        return array([[dr_dx, dr_dy, dr_dtheta], [dalpha_dx, dalpha_dy, dalpha_dtheta]])

    def correct(self, measurement, landmark_index):
        """The correction step of the Extended Kalman Filter."""
        # Get (x_m, y_m) of the landmark from the state vector
        landmark = self.state[3+2*landmark_index : 3+2*landmark_index+2]

        # Compute H using dh_dstate
        H3 = self.dh_dstate(self.state, landmark)

        # Modify H to accomodate all added landmarks
        H = zeros((2, 2*self.number_of_landmarks+3))
        H[0:2, 0:3] = H3
        H[0:2, 3+2*landmark_index:5+2*landmark_index] = dot(H3[0:2, 0:2], -1)

        # Q, a diagonal matrix, from self.measurement_distance_stddev and self.measurement_angle_stddev
        Q = diag([self.measurement_distance_stddev**2, self.measurement_angle_stddev**2])

        # K from self.covariance, H, and Q
        x = dot(H, self.covariance)
        x2 = dot(x, H.T)
        x3 = x2 + Q
        x3 = linalg.inv(x3)
        K = dot(self.covariance, H.T)
        K = dot(K, x3)

        # The innovation, bearing in mind that the predicted measurement and the actual measurement of theta may have an offset of +/- 2 pi
        innovation = array(measurement) -\
                                 self.h(self.state, landmark)
        innovation[1] = (innovation[1] + pi) % (2*pi) - pi

        # Compute the new self.state
        self.state = self.state + dot(K, innovation)
```

Figure A.5: EKF SLAM Python 3 code (part 5).

```
            # Finally, compute the new self.covariance
            x = eye(len(self.state)) - dot(K, H)
            self.covariance = dot(x, self.covariance)

    def mahalanobis(self, measurement, landmark_index):     # The Mahalanobis Distance
            # Get (x_m, y_m) of the landmark from the state vector.
            landmark = self.state[3+2*landmark_index : 3+2*landmark_index+2]

            # Compute H using dh_dstate
            H3 = self.dh_dstate(self.state, landmark)

            # Modify H to accomodate all added landmarks
            H = zeros((2, 2*self.number_of_landmarks+3))
            H[0:2, 0:3] = H3  # Replace this.
            H[0:2, 3+2*landmark_index:5+2*landmark_index] = dot(H3[0:2, 0:2], -1)

            # Q, a diagonal matrix, from self.measurement_distance_stddev and self.measurement_angle_stddev
            Q = diag([self.measurement_distance_stddev**2, self.measurement_angle_stddev**2])

            # Inverse of measurement covariance
            x = dot(H, self.covariance)
            x2 = dot(x, H.T)
            x3 = x2 + Q
            x3 = linalg.inv(x3)

            # The innovation, bearing in mind that the predicted measurement and the actual measurement of theta may have an offset of +/- 2 pi
            innovation = array(measurement) -\
                                    self.h(self.state, landmark)
            innovation[1] = (innovation[1] + pi) % (2*pi) - pi

            return dot(innovation.T, dot(x3, innovation))

    def get_landmarks(self):
            """Returns a list of (x, y) tuples of all landmark positions."""
            l = []
            for j in range(self.number_of_landmarks):
                    l.append([self.state[3+2*j], self.state[4+2*j]])
            return l

def compute_derivative(scan, min_dist):     # The derivative of measurement data
    jumps = [ 0 ]
    for i in range(1, len(scan) - 1):
        left = scan[i-1]
```

Figure A.6: EKF SLAM Python 3 code (part 6).

```
        l = left[0]
        right = scan[i+1]
        r = right[0]
        if l > min_dist and r > min_dist:
            derivative = (r - l) / 2
            jumps.append(derivative)
        else:
            jumps.append(0)
    jumps.append(0)
    return jumps

def find_cylinders(scan, scan_derivative, jump):    # Finding landmarks
    cylinder_list = []
    on_cylinder = False
    sum_angle, sum_depth, rays = 0.0, 0.0, 0

    for i in range(len(scan_derivative)):
        df = scan_derivative[i]
        if df <= -jump:         # New landmark, detected due to noticeable decrease in depth
            on_cylinder = True
            sum_angle, sum_depth, rays = 0.0, 0.0, 0

        if on_cylinder:
            if abs(df) <= jump: # Practically constant
                depth = scan[i]
                sum_depth += depth[0]
                sum_angle += depth[1]
                rays += 1
            elif df >= jump:    # Not on landmark anymore, detected due to noticeable increase in depth
                on_cylinder = False
                if rays !=0:
                    cylinder_list.append( (sum_depth/rays, sum_angle/rays) )    # Add the average of depths and angles of the detected landmark

    return cylinder_list

def compute_cartesian_coordinates(cylinders):   # Convert from Polar to Cartesian coordinates
    result = []
    for c in cylinders:
        depth, angle = c
        x = depth*cos(angle*pi/180)
        y = depth*sin(angle*pi/180)
        result.append([x, y])
    return result
```

Figure A.7: EKF SLAM Python 3 code (part 7).

```
def get_observations(minimum_valid_distance, depth_jump):   # Process the received measurement data to find landmarks
    scan = []
    f = open("data.txt")
    finished = 0
    for line in f:
        if line.startswith('S'):
            data = line.split()
            finished +=1
            if finished == 2:
                break
            if int(data[6]) >= 1:
                theta = float(data[2])
                r = float(data[4])
                scan.append([r, theta])
        elif line.startswith(' '):
            data = line.split()
            if int(data[5]) >= 1:
                theta = float(data[1])
                r = float(data[3])
                scan.append([r, theta])

    f.close()
    drawg(scan)     # Draw the room's walls

    # Find landmarks
    der = compute_derivative(scan, minimum_valid_distance)
    cylinders = find_cylinders(scan, der, depth_jump)
    cartesian_cylinders = compute_cartesian_coordinates(cylinders)

    return cartesian_cylinders, cylinders

def drawb(g):       # Draw the travelled path
        for i in range(0, len(g)):
            p = g[i]
            if i == (len(g) - 1):
                scatter(p[0], p[1], c = "black", s=50)  # Draw final position as a black dot
            else:
                plot(p[0], p[1], 'bo')  # Draw other positions as blue dots

def drawr(g):       # Draw landmarks
        scatter([c[0] for c in g], [c[1] for c in g], c='r', s=200)
```

Figure A.8: EKF SLAM Python 3 code (part 8).

```
def drawg(g):       # Draw the room's walls
    points = []
    for p in g:
        r, theta = p
        x = r * cos(theta*pi/180)
        y = r * sin(theta*pi/180)
        points.append([x, y])
    xc, yc = zip(*points)
    scatter(xc, yc, c='g', s=20)

if __name__ == '__main__':

        fd = 0
        old_settings = termios.tcgetattr(fd)
        tty.setraw(fd)

        distance_to_stop=40         # Distance from obstacle where the GoPiGo should stop and turn

        # Number of consecutive right and left turns
        rightTurns = 0
        leftTurns = 0

        ch = 0      # Default user input character

        ticks_to_mm = 10.8          # The ratio to convert motor ticks reading to distance in mm
        robot_width = 125.0         # The robot width

        minimum_valid_distance = 20.0   # Min measured depth accepted
        depth_jump = 15.0               # Depth difference threshold
        max_cylinder_distance = 5.0

        control_motion_factor = 0.35    # Error in motor control

        measurement_distance_stddev = 600.0             # Distance measurement error of landmarks
        measurement_angle_stddev = 45. / 180.0 * pi     # Angle measurement error

        initial_state = array([0.0, 0.0, 0.0])          # Start pose at origin heading towards positive x axis
        initial_covariance = zeros((3,3))               # Covariance at start position
```

Figure A.9: EKF SLAM Python 3 code (part 9).

```
# Setup filter
kf = ExtendedKalmanFilter(initial_state, initial_covariance, robot_width,
                    control_motion_factor, ticks_to_mm, measurement_distance_stddev,
                    measurement_angle_stddev)

states = []                             # Empty list of poses
states.append(initial_state)            # Add the start pose to poses list

print("Press\n\rr: to run\n\rt: to terminate the code\n\rany other char: to stop\n\r")

while True:
        while ch == 'r':    # 'r' received from user
            fwd()           # Start moving forward
            time.sleep(0.2)
            ticks = 8       # Moving forward for 200ms gives 8 motor ticks
            kf.predict(ticks)    # Predict the new pose
            states.append(kf.state[0:3])        # Add the new pose to poses list
            dist=us_dist(15)                    # Find the distance of the object in front
            if dist<distance_to_stop:       # If the object is closer than the "distance_to_stop" distance, stop the GoPiGo
                    dist = us_dist(15)      # Confirm that the distance is indeed less than the "distance_to_stop"
                    if dist < distance_to_stop:
                            stop()                      # Stop the GoPiGo
                            time.sleep(0.1)
                            servo(0)                    # Looking right
                            time.sleep(0.2)
                            right_dist = us_dist(15)    # Distance of nearest object to the right
                            time.sleep(0.1)
                            servo(140)                  # Looking left
                            time.sleep(0.4)
                            left_dist = us_dist(15)     # Distance of nearest object to the left
                            time.sleep(0.1)
                            servo(70)                   # Look forward
                            #If (right is empty and we didn't turn right 3 consecutive times) or (we turned left 3 consecutive times and the nearest object
                            # to the right is at least 100cm away)
                            if (left_dist < right_dist and rightTurns < 3) or (leftTurns >= 3 and right_dist >= 100):
                                    right_rot()             # Then turn right
                                    rightTurns += 1         # Increment right consecutive turns
                                    leftTurns = 0           # Reset left consecutive turns
                                    p = list(kf.state[0:3])
                                    p[2] += 90              # Increment heading by 90 degrees
                                    kf.state[2] = p[2]
```

Figure A.10: EKF SLAM Python 3 code (part 10).

```
                else:
                        left_rot()              # If above condition failed, then turn left
                        leftTurns += 1          # Increment left consecutive turns
                        rightTurns = 0          # Reset right consecutive turns
                        p = list(kf.state[0:3])
                        p[2] -= 90              # Decrement heading by 90 degrees
                        kf.state[2] = p[2]
                time.sleep(0.435)
                stop()
                # Read the measurement data
                os.system('./startLidar.bash > data.txt')
                while os.stat('data.txt').st_size == 0:
                        os.system('./startLidar.bash > data.txt')
                # Detect landmarks
                observations, measurement = get_observations(minimum_valid_distance, depth_jump)
                landmarks = kf.get_landmarks()
                # Add new landmarks to the state vector and use added landmarks to correct the predicted pose
                for j in range(len(observations)):
                    not_added = True
                    if observations[j] not in landmarks:
                        for i in range(kf.number_of_landmarks):
                            if kf.mahalanobis(measurement[j], i) < max_cylinder_distance:
                                not_added = False
                                cylinder_index = i
                                break
                        if not_added:
                            cylinder_index = kf.add_landmark_to_state(observations[j])
                    else:
                        cylinder_index = 0
                        for i in range(len(landmarks)):
                            if observations[j] == landmarks[i]:
                                break
                            else:
                                cylinder_index += 1
                    kf.correct(measurement[j], cylinder_index)
    # Check for received character from user
    if sys.stdin in uselect.select([sys.stdin], [], [], 0)[0]:
        ch = sys.stdin.read(1)
        print("\n\r"+ch)
stop()
```

Figure A.11: EKF SLAM Python 3 code (part 11).

```
            if sys.stdin in uselect.select([sys.stdin], [], [], 0)[0]:
                ch = sys.stdin.read(1)
                print("\n\r"+ch)
            if ch == 't':   # Terminate the program
                stop()

                for pose in states:     # Print all poses
                    print(pose)

                drawb(states)   # Draw the travelled path
                drawr(kf.get_landmarks())   # Draw the landmarks
                show()      # Popup the map
                break

    termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
```


Figure A.12: EKF SLAM Python 3 code (part 12).