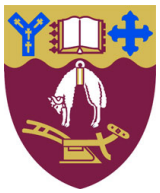
A Semi-autonomous Wheelchair Navigation System

Robert Tang

A thesis submitted in partial

fulfilment of the requirements of

Master of Engineering



Department of Mechanical Engineering

University of Canterbury

February 2012

Abstract

Many mobility impaired users are unable to operate a powered wheelchair safely, without causing harm to themselves, others, and the environment. Smart wheelchairs that assist or replace user control have been developed to cater for these users, utilising systems and algorithms from autonomous robots. Despite a sustained period of research and development of robotic wheelchairs, there are very few available commercially.

This thesis describes work towards developing a navigation system that is aimed at being retro-fitted to powered wheelchairs. The navigation system developed takes a systems engineering approach, integrating many existing open-source software projects to deliver a system that would otherwise not be possible in the time frame of a master's thesis.

The navigation system introduced in this thesis is aimed at operating in an unstructured indoor environment, and requires no *a priori* information about the environment. The key components in the system are: obstacle avoidance, map building, localisation, path planning, and autonomously travelling towards a goal. The test electric wheelchair was instrumented with the following: a laptop, a laser scanner, wheel encoders, camera, and a variety of user input methods. The user interfaces that have been implemented and tested include a touch screen friendly graphical user interface, keyboard and joystick.

Acknowledgements

I would like to begin by thanking my supervisors: Dr. XiaoQi Chen (Mechanical Engineering, University of Canterbury) and Dr. Michael Hayes (Electrical and Computer Engineering, University of Canterbury), as well as my industry mentor Ian Palmer (Dynamic Controls, Christchurch), for their guidance and support throughout the past 18 months. Ian provided me with insight into the wheelchair industry, along with help with sourcing of components; XiaoQi for his always enthusiastic attitude and suggestions towards the 'bigger picture'; Michael for his invaluable help with software development and debugging, his character, and for many interesting discussions on embedded systems.

Funding of my research has also played a crucial role. I greatly appreciated the base funding that this project received from the Foundation for Research, Science and Technology (FRST). I am also in debt to Lincoln Sell who helped me to put forward my research proposal to Dynamic Controls, and XiaoQi and Ian for transforming this proposal into a feasible project and attaining FRST funding.

Throughout the course of this project, NZi3 graciously provided a workspace and the use of computer equipment. I appreciated this, especially NZi3's comfortable and modern work environment plus the opportunity to join their community of like-minded research students and staff.

I thank the university technicians from both the electrical and mechanical departments for their support and practical knowledge. I would also like to thank the friendly and helpful staff at Dynamic Controls, in particular Warren Pettigrew for his help in instrumenting the wheelchair, and Reuben Posthuma for his work on developing a device that allowed a simple interface to the Dynamic Control's DX2 wheelchair controller.

I am also grateful to all those who helped me proof-read and edit my thesis, especially Michael for his grammatical prowess, minute attention to detail, and extensive knowledge on how to format a thesis using LaTeX. Finally a thank you goes to my supportive friends, family (including mother Jocelyn Syme and father Ngie Uong Tang), and fellow postgraduates, in particular, Malcolm Snowdon, who assisted with odometry experiments, and John Stowers, who provided help on numerous software related issues.

Contents

Glossary

AMCL	Adaptive Monte Carlo localisation
ANN	Artificial neural network
C-space	Configuration space, the legal configurations of the robot in an environment
DAQ	Data acquisition systems
EKF	Extended Kalman filter
FOV	Field of view
FPS	Frames per second
Gazebo	A 3-D simulator for Player
GCBB	Geometric compatibility branch and bound algorithm
GPSB	General Purpose SLIO Board (SLIO is serial linked input-output)
HAL	Hardware abstraction layer
IR	Infra-red
JCBB	Joint compatibility branch and bound
KLD	Kullback-Leibler distance
LIDAR	Light detection and ranging
LQG	Linear-quadratic Gaussian
MCL	Monte Carlo localisation
ND	Nearness diagram
NN	Nearest neighbour
PDF	Probability density function
PF	Particle filter
Player	The Player Project, an open-source robotics framework

POMDP	Partially observable Markov decision process
Pose	The combination of an object's position and orientation
PSG	A configuration with Player, Stage and Gazebo
RANSAC	Random sample and consensus
RBPF	Rao-Blackwellised particle filter
SLAM	Simultaneous localisation and mapping
SND	Smoothed nearness diagram
Stage	A 2-D simulator for Player
Tele-robot	Area of robotics controlled from a distance, usually through a wireless connection
VFH	Vector field histogram

Chapter 1

Introduction

Power assisted wheelchairs give a level of independence to mobility impaired users. Control of the wheelchair is usually through the use of a joystick. However, there are many users who have various cogitative impairments that prevent them controlling an electric wheelchair safely [1, 2], without causing harm to themselves, others and collisions with the surrounding environment. Despite the need for higher levels of user-assisted control modes, few smart wheelchairs are available. Due to the lack of commercial availability, this has resulted in a limited clinical impact [1] and thus lack of acceptance.

This project aims to cater for this group of users, by creating a wheelchair navigation system capable of providing autonomous operation, obstacle avoidance, and simplified command routines. Currently there are no commercial systems available with this capability. The purpose of this chapter is to provide an overview of the objectives and scope of the work (Section 1.1), followed by an summary of the other chapters in this thesis (Section 1.2). Section 1.3 provides a summary of the contributions resulting from the outcomes of this work.

1.1 Objectives

The primary objective of this work is to develop a navigation system that is able to drive an electric wheelchair from point to point without human intervention. The focus is on cost-effective solutions able to operate in unstructured indoor office/residential environments which can be easily retro-fitted to a range of electric wheelchair types while being minimally intrusive to the user. The task of developing the navigation system is split into the following areas:

- Map building: create a map of the environment, without prior knowledge or structuring the area in any way.
- Localisation: estimate the pose of the wheelchair within the map.
- Obstacle avoidance: routines responsible for preventing the wheelchair from colliding with dynamic obstacles and the environment.

- Path planning: compute the most efficient path to reach a goal from the wheelchair's current location.
- Semi-autonomous navigation: travel to a the user-specified goal without intervention.
- User input: provide a range of methods by which the user can control the system.
- Instrumentation: install sensors on a wheelchair, and evaluate the navigation system.

1.2 Thesis overview

This thesis centres on developing a navigation system for wheeled vehicles, namely electric wheelchairs. The thesis is structured as follows:

Chapter 2 — presents a review of the development of smart wheelchairs, and their core components, i.e., sensors and electronics, user input methods, and navigation assistance. This chapter also reviews both commercially available and existing research-based smart wheelchairs.

Chapter 3 — provides an overview of simulation and robotic frameworks, and reasoning why integrating them into this project is highly desirable. After a review of several open-source robotic software packages, a more thorough investigation into the selected framework is discussed (Player, see Section 3.3).

Chapter 4 — this chapter provides an introduction into the simultaneous localisation and mapping (SLAM) problem. The intricacy of mapping while carrying out localisation is discussed, as well as the key challenges in SLAM, including: handling measurement errors, data association, loop closure, and managing computational complexity. Modern SLAM filtering techniques are also discussed, followed by a review of prominent open-source software SLAM implementations.

Chapter 5 — provides an overview into the local (obstacle avoidance) and global (path planning) navigation tasks. A review of existing Player navigation drivers is also provided.

Chapter 6 — this chapter outlines the physical robots used to evaluate the navigation system that was developed in this project. This includes a small robot that was used as an intermediate step between a virtual robot and the instrumented wheelchair. High level block diagrams are used to show the connection between the main hardware components.

Chapter 7 — this chapter presents an overview of the software side of the navigation system. This includes embedded controller software (interfacing to the wheelchair's motors and joystick, performing dead reckoning, its serial interface), custom Player plugins, Player setup and the client program, and the user interface.

Chapter 8 — contains results from running the navigation system on both the small robot and the electric wheelchair. The results are also compared to that obtained within a simulation environment.

Chapter 9 — conclusion of the work with a summary and an outlook. Areas for improvement / future development are also discussed in this chapter.

1.3 Contributions of this thesis

The primary contribution of this thesis is the development of a navigation system for semi-autonomous operation of wheelchairs. The navigation system is demonstrably capable of performing all of the objectives as listed in Section 1.1. It has been built from a selection of open-source software libraries[1]. This is useful for the client, Dynamic Controls, as it has forced the developed navigation system to adhere to standards used by other robotics researchers.

Another key contribution that this thesis provides is literature reviews into relevant areas towards smart wheelchairs navigation systems. Together, the reviews provide a broad, but compact, overview into smart wheelchair systems. Chapter 2 reviews typical smart wheelchair components, and evaluates research and commercially available systems. Meanwhile, Chapter 4 provides an overview of the SLAM problem and highlights modern implementations. Also, Chapter 5 provides research into suitable obstacle avoidance and path planning techniques. The literature reviews also helped to ascertain current trends into smart wheelchairs and autonomous robots, thus providing direction for the research and development carried out in this thesis.

Finally, another contribution arising from the work from this thesis is an instrumented wheelchair. This was created to validate the navigation system in a real environment, and allows Dynamic Controls to conduct further research towards smart wheelchair navigation systems. The wheelchair instrumentation details are provided in Section 6, and experimentation (including sensor calibration) on this platform is discussed in Section 8.

[1]The main open-source frameworks/libraries used in this navigation system are: Player (see Section 3.3), Stage (see Subsection 3.3.5), GMapping (see Subsection 4.4.4), and OpenCV.

Chapter 2

Smart wheelchairs

Studies have shown that many individuals with disabilities would substantially benefit from a means of independent mobility [2–4]. Most of these individuals' mobility needs can be satisfied with traditional or powered wheelchairs. However, there are still others in the disabled community who find it difficult or impossible to operate these wheelchairs independently [1, 2]. According to Simpson [4], in 2008 there are between 1.4 to 2.1 million people in the USA alone that would benefit from a smart wheelchair. Smart wheelchairs have been developed since the 1980's specifically for these severely disabled people [1].

This chapter presents research into the development of smart wheelchairs and the core components that make up a smart wheelchair (Section 2.1). The benefits they provide to the user over manual or powered wheelchairs is also covered. A review of both commercially available and research-based smart wheelchairs are presented in sections 2.2 and 2.3, followed by challenges and shortcomings of current smart wheelchairs in Section 2.4.

2.1 Core components of smart wheelchairs

Smart wheelchairs are typically powered wheelchairs that have been instrumented with sensors and have an on-board computer [5]. They have been designed to provide navigation assistance in a number of different ways, including: collision avoidance [6–8], aiding specific tasks (e.g., passing through doorways) [6, 9], and autonomously transporting the user between locations [8, 10, 11]. The following subsections provide a breakdown of the main components that are usually in a smart wheelchair system.

2.1.1 Sensors and electronics

A distinguishing feature between smart and manual/powered wheelchairs is the ability of the machine to intervene. The sensors installed on a smart wheelchair play a key role, as they provide a means for the wheelchair to perceive its surroundings. A significant challenge in smart wheelchairs is finding the correct sensors, as the requirements are difficult to satisfy. They need to be accurate, inexpensive, small, lightweight, consume little power, and be robust to stand up to environmental con-

ditions [1]. Since researchers have yet to find a single sensor that satisfies all these needs, many smart wheelchair systems fuse information from multiple sensors.

By themselves, each sensor type has its own set of problems: ultrasonic range-finders fail on sound absorbent surfaces and IR range-finders can be fooled by light absorbent surfaces. Cameras require sophisticated algorithms to extract the desired information as well as filter out the unwanted details and noise. Fusing sensor data is a well known practice, whereby limitations of one sensor are compensated with another sensor(s) and vice versa [12, 13].

In the past, smart wheelchairs have been mostly instrumented with ultrasonic and infra-red (IR) range-finders to perceive their surroundings [1, 6, 10]. However, laser scanners are difficult to mount inconspicuously, consume relatively high amounts of power, and are often prohibitively expensive [1]. Recently, smart wheelchairs have been instrumented with cameras and by using computer vision, a wealth of information can be extracted (and often used to improve other sensor readings) [1, 14, 15].

Another key component of any smart wheelchair system is its on-board computer. Its role is to process sensory and user inputs and control the wheelchair accordingly [1]. Generally, the sensors contain their own specific data acquisition systems (DAQ) (sensor electronics, analog to digital converters, signal conditioning circuitry, etc) and interface to the computer via a serial bus, e.g., RS232/RS422/RS485, CAN, I2C, USB. This reduces the need for an expensive and specialised embedded controller board. Thus research-based smart wheelchair projects often use a PC or a laptop for the main processing power [5–8, 16].

2.1.2 User input methods

Traditionally, powered wheelchairs have been controlled through a joystick [1]. Although there are other options, a lack of configurability limits the list of input devices that can interface to the wheelchair. Therefore, smart wheelchairs provide excellent test beds for novel user input methods. The following subsections provide examples of input methods used in powered and/or smart wheelchairs.

Force-feedback, sip-and-puff devices, and other forms of joysticks

A study performed by Fehr et al. [2] found that more than 95 % of power wheelchair users manoeuvre their chairs by with a joystick, sip-and-puff, head, or chin control. This research was presented in 2000 and since then, more sophisticated input methods have become available. However it is likely that the joystick class of devices will remain as the most common method in controlling a powered wheelchair.

Many users struggle to manoeuvre their powered wheelchair in confined spaces [2]. Common tasks such as passing through doorways, turning around in halls or even travelling on a straight path are difficult for users with certain disabilities [2], such as Demyelinating disease. An extension to a joystick controlled wheelchair is to provide feedback via an active joystick. Force-feedback joysticks have been demonstrated [9, 17] to significantly improve the piloting performances over a traditional joystick

control.

A typical approach to force-feedback assisted control joysticks is to instrument the wheelchair with range-finding sensors. A control algorithm is then developed which adjusts the level of force feedback given to the user based on the proximity of obstacles [9, 17].

Alternatively, sip-and-puff devices[1] provide a solution for users who are not able to use any part of their body to operate a control device on a wheelchair. Activated by the user's breath, sip and puff devices are programmed to emulate joystick movements. This is accomplished through recognising if the user is blowing or sucking, and the strength or duration of a sip or puff. There are other forms of joysticks as well, including: chin control and finger touch pads as described by Felzer and Nordman [18].

Voice recognition

Another form of user input is through voice recognition. This has been successfully implemented on smart wheelchairs, e.g., NavChair [6], SENARIO [19] and "The MIT Intelligent Wheelchair Project" (referred as MIT wheelchair hereafter) [8]. This type of control is beneficial to users who suffer from severe motor impairments [6].

Voice recognition control is generally only applicable to wheelchairs equipped with obstacle detection and avoidance systems. This is because low bandwidth devices such as voice control do not provide adequate control without the safety net provided by obstacle avoidance [1]. Thus systems using voice recognition such as the NavChair use its obstacle avoidance system to fill in small, but appropriate, navigation commands.

In the past, voice recognition control has required the user to say specific, predefined key-words. For instance, the NavChair speech vocabulary includes: 'stop', 'go forward', 'go backward', 'soft left', 'hard left', 'soft right', etc [20]. On the other hand, the MIT wheelchair can be controlled by natural speech. This is achieved by using Partially observable Markov decision process (POMDP), as these models are capable of determining the user's intention in the presence of ambient noise, linguistic ambiguity, etc [21]. Note that although this system still requires the user to speak key words, they can be spoken in natural sentences (and thus making it more user friendly to operate).

User expressions

Wheelchairs have also been controlled using user expressions, by detection of the user's sight path (where the user is looking). One approach used electro-oculographic activity (recording and interpreting eye movements). This method is used in Wheelesely [7]. Another method uses computer vision to find the pose of the user's head, e.g., Osaka University [22] and Watson [16]. As mentioned by Kuno et al. [22], since control inputs derived from user expressions are generally noisy, systems using user

[1] An example of a commercially available sip-and-puff device: `http://www.therafin.com/sipnpuff.htm`

expression methods require integration from information obtained from environment sensing as well.

Brain-machine interface

The first documented discovery of electro-encephalogram (EEG) signals dates back to 1875 [23]. Since then, research to implement control of a machine via 'brain waves' has been pursued. However, only recently have developments yielded one of the first practical uses of EEG. A collaborative effort between RIKEN, Toyota, Genesis Research Institute and other groups developed a system which uses EEG to control a wheelchair [24]. The technology is reported to be capable of detecting a user's driving directives at 8 Hz with a 95 % accuracy rate [24], making it suitable for real-time control of a wheelchair. However, this technology is still in development. It also involves several hours of training the user how to think of a certain command before the system can be used. It is also likely that the commands that can be decoded from the EEG signals are primitive, and unlike speech control, limited to simple directives such as left, right, forward, reverse and other trained thought commands. A somewhat critical view on the brain-machine interface is provided by Felzer and Nordman [18]. Since the EEG signal is sensitive to all human actions (e.g., blinking, swallowing, laughing, talking), it is likely that in practice the resulting wheelchair command would be highly contaminated [18].

2.1.3 Navigation assistance

In the past, different smart wheelchairs have offered several different forms of navigation assistance to their users. Some smart wheelchairs simply provide collision avoidance and leave the planning and navigation tasks to the user, such as the NavChair [6]. These systems often have different user-selectable modes. For instance, the NavChair has the ability to assist in various tasks, including: obstacle avoidance, passing through doorways, and wall tracking. Other systems have the ability to follow targets, e.g., the MIT wheelchair [8, 25]. The benefit of this approach is that the system requires no prior knowledge of the area and that the environment does not need to be altered.

Another approach to navigation assistance is to teach the wheelchair to follow prescribed routes. Neural networks have been used to reproduce pre-taught routes. Examples are from The University of Plymouth [26] and The Chinese University of Hong Kong [27]. This approach is particularly suitable to situations where initial training is available and environments which do not change much, such as in elderly homes and hospital wards. The MIT wheelchair can also be trained on an environment, via its speech recognition interface by a human 'tour guide' [25]. By splitting an existing map of the premises into sections, the wheelchair is able to tag a keyword/phrase to these areas.

At the extreme end, some smart wheelchairs operate in a similar way to autonomous robots. Here the user only needs to specify a final destination, and carries out a path to the target location without user intervention. Such systems often require a complete map of the area to be traversed. The process of creating a map

of the environment is discussed in Chapter 4, while Chapter 5 covers path planning. A simple alternative to this is to structure the environment with unique features, e.g., tape tracks on the floor or markers placed on the walls [1]. Examples of such wheelchairs include TetraNauta [10] and Kanazawa University [11]. These systems are best suited to users who lack the ability to plan or execute a path to a destination and spend most of their time within the same environment [1].

2.2 Research-based smart wheelchairs

The majority of smart wheelchairs that have been developed to date have been either based on installing seats on mobile robots or modified power wheelchairs [1]. This section reviews some research-based smart wheelchairs.

2.2.1 NavChair

The NavChair [6] was developed at the University of Michigan from 1993 to 2002. Essentially, NavChair is a retro-fitted power wheelchair that uses an array of ultrasonic range-finders for detecting obstacles and wheel encoders for odometry. The user can control the machine through either a joystick or through the use of specific voice commands. NavChair shares vehicle control with the user and was not intended to provide autonomous operation.

Using data from both its ultrasonic sensors and wheel encoders, an grid-based obstacle map is generated, where the wheelchair is at the centre of this map. Obstacle avoidance is then carried out using a modified version of the Vector Field Histogram (VFH) method (see Subsection 5.1.2). The NavChair adjusts the level of driving assistance provided by VFH through a weighting function.

There are different user assistance modes on the NavChair system. These are: obstacle avoidance, passing through doorways, and wall following. Although the user could manually select the mode, this could make operation cumbersome. Instead, NavChair automatically switches between its modes by combining information about its immediate surroundings using Bayesian networks [6].

2.2.2 TetraNauta

Developed at the University of Seville from 1998 to 2004, TetraNauta [10] was designed as a system that could be retro-fitted to several makes/models of wheelchairs. The TetraNauta project was targeted towards operation in a known and controlled environment, such as elderly homes, hospitals, schools, homes, etc. Automatic navigation was possible and was achieved simply by following lines and marks painted on the floor of the environment. A camera is used to pick up the presence and location of the lines and marks. The lines are one of four colours, and when the wheelchair is in assisted mode, give the following behaviours:

Repulsive line — used to create 'virtual' corridors, with the intention of keeping the wheelchair within some bounds of an environment.

Impassable line — restricts the user from breaching over a boundary.

Attractive line — the wheelchair aligns itself with the line, providing navigation assistance for challenging tasks such as passing through a doorway.

Rail line — the wheelchair user can only go forwards or backwards once on a rail line but cannot deviate from it.

To allow multiple units operating within the same environment, there was a wireless central unit which received positions of each TetraNauta unit. The central unit relayed back information to manage 'traffic' and avoid collisions between wheelchairs. TetraNauta reduces the navigation task by travelling on pre-defined paths that are preferably obstacle free. Since this may not always be the case, it also is equipped with an infra-red based obstacle detection system to prevent collisions with dynamic obstacles [10].

2.2.3 The MIT Intelligent Wheelchair Project

The MIT Intelligent Wheelchair Project [8] was first started in 2005 and is currently being developed. It is controlled primarily through speech recognition. Unlike previous voice controlled smart wheelchairs (such as NavChair [6] and SENARIO [19]), the user is able to use natural speech. This makes the wheelchair suitable for patients who may have suffered a brain injury or the loss of limbs but who are still capable of speaking.

Absolute positioning within a complex is based on spatial variations in "wifi signatures" [28]. In their nine story test environment, there were 200 wireless access points. When the wheelchair came across an unfamiliar access point's signature, the system prompted the user to enter their location. Gradually, the system learns the positions of the access points, and can thus localise itself to within 10 m 92 % of the time [28]. The MIT wheelchair has also been equipped with forward- and rear-facing laser scanners, used for obstacle avoidance and to generate a map of the environment (see the Section 4 on Simultaneous Localisation and Mapping (SLAM)).

The natural speech interface is also used to train the wheelchair. When it is being trained, it follows a 'tour guide' around the environment [25]. The keywords/phrases the tour guide says are spatially tagged to locations within the environment. This permits the user to ask the wheelchair to take him or her to a destination autonomously.

2.3 Commercialisation of smart wheelchairs

Despite sustained research, few smart wheelchairs are currently on the market [1]. Companies which sell smart wheelchairs often sell them to researchers. However, the intended use of such smart wheelchairs is within the confines of a laboratory environment [1]. This lack of commercial availability has meant that smart wheelchairs have yet to have widespread clinical use (and thus acceptance, and vice versa) [1]. Moreover, extravagant smart wheelchairs are difficult to make commercially

viable as they are often too expensive and complicated [10]. Popular sensors in research wheelchairs such as laser scanners are prohibitively expensive [1]. Such smart wheelchairs also tend to require modifying a powered wheelchair, which by doing so voids its warranty.

One example of a commercially available smart wheelchair is that from Smile Rehab Limited (SRL)[1]. Originally, this wheelchair was developed at The University of Edinburgh in 1987. Compared to the research-based smart wheelchairs discussed in Section 2.2, the SRL wheelchair is somewhat basic: its sensors consist of a line follower and contact switches. However, it has an serial port with a specified protocol allowing an external controller to interface to it. This opens up the possibility for the end user to add more sophisticated functionality to it. It is interesting that SRL chose this approach opposed to providing an obstacle avoidance and path planning capabilities.

Another example is the Robotic Chariot by Boo-Ki Scientific[2]. Despite being listed as a research vehicle, the Robotic Chariot builds on a Pride Mobility Jazzy 1120[3], a popular powered wheelchair. It is retro-fitted with a laser scanner and other sensors and is controlled by ActivMedia Robotics Interface for Applications software (ARIA)[4]. Although the ARIA framework provides sophisticated obstacle avoidance and path planning, it appears that it is up to the end user to customise the Robotic Chariot's navigation functionality.

2.4 Current challenges in smart wheelchairs

An important aspect of any product is its cost. In most robotics applications, accurate and reliable sensors are vital and form a large portion of the system cost. Unfortunately, LIDAR based laser range-finders used in several research-based smart wheelchairs are prohibitively expensive [1]. Therefore, a current challenge is balancing the cost versus accuracy trade-off with sensors. Laser range-finders are also difficult to mount discreetly on a wheelchair [1].

Another key challenge is running smart wheelchair algorithms on mobile computing platforms. Computer vision algorithms are often demanding, as are map generation algorithms (particularly with large scale, high fidelity environments). Chapter 4 highlights the issue of managing complexity in the map building process.

There is also a lack of a physical interface and a standard communication protocol between wheelchair input devices and add-on modules between vendors. It is hoped that this problem is partially alleviated by adopting a popular robotics framework amongst researchers (Player, see Section 3.3).

Furthermore, due to the lack of availability of smart wheelchairs on the market, they have yet to gain clinical acceptance (and vice versa) [1]. It is envisaged that this challenge will eventually be overcome as the technology becomes more mature and after extensive testing in clinical trials.

[1] `http://www.smilerehab.com/`
[2] `http://bookitec.co.kr/chariot.htm`
[3] `http://www.pridemobility.com/jazzy/index.asp`
[4] `http://www.mobilerobots.com/ResearchRobots/PioneerSDK/ARIA.aspx`

Chapter 3

Simulation environment and robotic framework

This chapter presents research on a variety of existing robotics frameworks. The motivation for integrating a framework into the project was primarily to rapidly speed up development by leveraging several years of collaborative research and community work. Furthermore, adopting a framework also forces the developed navigation system's software to adhere to a structure also used by other developers in the robotics community. This makes it easier for future development towards the navigation system described in this thesis. It is also common for robotics frameworks to come with a simulation environment. There are many benefits in simulating robots in virtual environments:

- Development can commence without having the sensors or an instrumented robot.
- The cost of prototyping is much cheaper and significantly less time-consuming.
- Visualisation and debugging tools are better.
- Hardware and software complexities can be separated.
- Testing can be performed anywhere.

However, simulation has its limitations. Essentially simulation attempts to turn a hardware problem into a software problem. The main drawback with simulation is that it is difficult to model many effects present in the real world [29]. Inadequate sensor models and lack of environment noise leads to substantial performance differences between control algorithms on simulated and physical robots [30]. To circumvent the drawbacks of simulation, developers tend to use a hybrid approach. One method is to cross-validate results from the simulation and the real experimentation and account for discrepancies [29, 31]. This approach is required to validate the simulated experiments, but requires either detailed simulation environments or overly simplistic real environments. Another method is to simply use each environment for its strengths, which is what was done in this project.

3.1 Robotic frameworks

With the ever increasing 'open-source movement', many engineering fields have benefited from free and high-quality software frameworks. This is especially the case for robotics, with several comprehensive open-source software packages available which make programming robotic applications easier [31–33]. Therefore, integrating one of these robotic frameworks into the project leverages several years of work, increasing the speed and ease of development.

This section reviews some robotic frameworks, and then discusses the selected framework (The Player Project [32]) in detail. Most of the reviewed frameworks are open-source and mature, as these are necessities for this project — it gives flexibility in development and deployment.

3.1.1 The Player Project

The Player Project [32] is a free open-source robotics framework often used in robotics and sensor systems research. Its core component is the Player network server, which has a comprehensive list of drivers supporting commercially available sensors and robots. Due to the use of the client/server model, robot control programs can be written in any programming language that provides a socket interfacing mechanism. The Player server can run on most machines, and the client application can be run on any computer with network connectivity to the robot. The Player Project (referred as Player herein) uses sockets for exchanging data within its modules, thus making it ideal for tele-robotic applications. Further, the Player server supports multiple clients, allowing for distributed and collaborative sensing, control, and computing.

Developers can add custom drivers to be used by the Player server through plugins, avoiding the need to re-compile the entire project. The simulation environments for Player, Stage (2-D), and Gazebo (3-D) are treated as plugins and are separate projects to Player. The client interface for communicating to a robot is the same, irrespective of whether it is a real or virtual robot in either the Stage or Gazebo simulators.

3.1.2 Pyro

Python robotics (Pyro)[1] is a open-source robotics framework designed to be easy to use. As opposed to compiled languages such as C/C++, Pyro uses Python which is an interpreted multi-paradigm language. This means that software development tends to be much faster, as the experiments with the robot can be done interactively (in 'real time'). However, the cost of using Python (or any other interpreted language) is that the resulting performance suffers when compared to compiled languages.

A key benefit of Pyro is that it provides a unified code base to control different mobile robots from different vendors. Pyro achieves this by using wrappers around

[1] `http://pyrorobotics.org/`

other robot frameworks and simulation environments [34]. For instance, Pyro uses Player for the Pioneer family of mobile robots, as well as Stage or Gazebo for the simulation environment. Pyro also comes with modules built in, such as: control methods, basic vision tasks (motion and blob tracking), and machine learning algorithms amongst others.

3.1.3 ROS

Robotic Operating System [33] (ROS) is an open-source operating system for robots. It provides operating system like features, such as a hardware abstraction layer (HAL), implementations of commonly used tasks, transferring of messages between processes, and package management. It runs on top of an actual operating system, namely Linux, however, ROS also supports other platforms. Functionality is added to ROS by installing ROS packages from a suite of user contributed packages. Packages include functionality to perform SLAM (see Chapter 4), obstacle avoidance, path planning, and perception.

ROS and Player have similar underlying concepts: they both provide a HAL and are designed to allow for distributed computing. ROS leverages a lot of code from Player, and ROS also supports the Stage and Gazebo simulation environments. ROS is designed to handle complex articulated mobile robot platforms, making ROS more complicated than Player.

3.1.4 USARSim

The Urban Search and Rescue Simulation (USARSim) [31] is a high-fidelity 3-D robotics simulator. It utilises the UnrealEngine2, obtained by purchasing the Unreal Tournament 2004 game (which runs on Windows, Mac OS X, and Linux platforms). Note that although the game engine is not open-source, the USARSim project is. A major benefit in using the UnrealEngine2 is that the game engine comes with a variety of specialised and sophisticated graphical development tools. This makes it relatively straight-forward to build detailed environments.

Originally developed as a research tool and the basis for the RoboCup rescue virtual robots competition [35], the USARSim framework has also gained popularity beyond the RoboCup community. Like the other frameworks, USARSim comes with a variety of models for common robots and sensors. It also includes ready to use 3-D test environments. USARSim can also be interfaced to Player [32]. Consirable emphasis is placed on USARSim's simulation environment, however, it too can control real robots.

3.1.5 OROCOS

OROCOS is the acronym of the Open Robot Control Software project[1]. Unlike the other frameworks reviewed so far, OROCOS uses lock-free buffers for exchanging data between its modules [36]. This makes it particularly suitable for deterministic

[1] `http://www.orocos.org/`

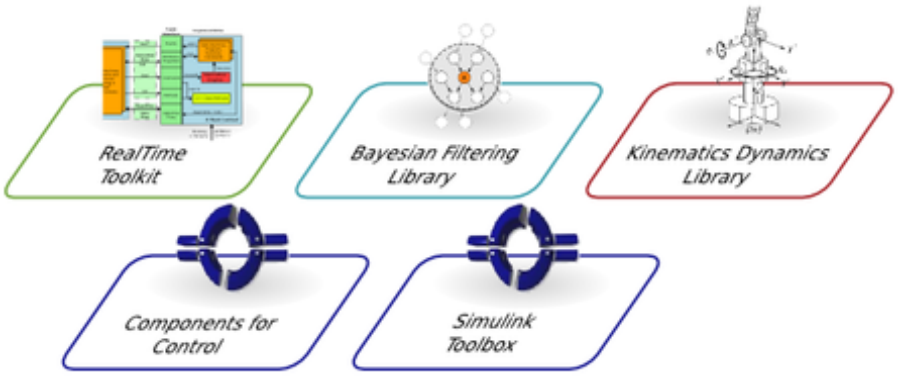


Figure 3.1: Core modules of the OROCOS framework (source: `http://www.orocos.org/`).

real-time applications. As shown in Figure 3.1, the main building blocks of OROCOS are its real-time toolkit, control modules, a filtering library, as well a dynamics library. It can also interface with Simulink, a powerful commercial toolbox for modelling dynamic systems.

Unfortunately, at the time of writing, OROCOS does not have any 2-D or 3-D simulation environment. OROCOS supports real-time and non-real time versions of Linux, although there are efforts in porting it to run on Windows.

3.1.6 MRDS

Another robotics framework is the Microsoft Robotics Developer Studio (MRDS)[1]. Although MRDS is not open-source, it is also popular amongst hobbyists and research projects. Applications are typically developed in C#, although MRDS also features a visual 'programming' environment, which enables non-programmers to create robotic applications. MRDS has a powerful 3-D physics-based simulation environment, enabling realistic rigid-body dynamics. MRDS does not have built-in components such as computer vision, artificial intelligence, or navigation systems. It instead uses partners to provide plugins for the framework [36], namely ProMRDS[2].

In addition to being closed-source, deployment of MRSD applications require robots to run a Microsoft operating system. This excludes the ARM architecture of microcontrollers often used on energy efficient embedded systems and limits the choice of the robot's on-board computer to mostly the x86 or x64 computer architecture.

3.2 Reasons for selecting Player

During the course of research, several robotic frameworks were evaluated. The Player Project was identified to be the most suitable framework to adopt for this

[1] `http://msdn.microsoft.com/robotics/`
[2] `http://www.promrds.com/`

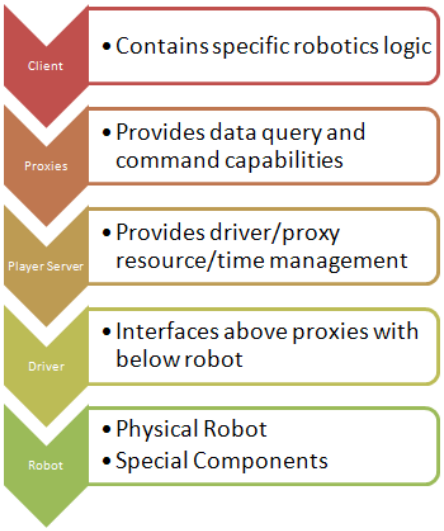


Figure 3.2: The layers of abstraction used in The Player Project (source: [37]).

project. This was because:

- The Player Project is mature. It was initiated in 2001 and has been developed since. It is still an active project.
- Player software is open-source. This is particularly important as it allows for flexibility that the project requires.
- It is considered to be one of the most popular frameworks amongst robot researchers. This means that Player has plenty of support via user forums.
- The client/server model works particularly well for tele-robotics. It also makes it easy to change between simulation and real environments and between local and remote (for debugging purposes) client-to-server connections.
- It has mature and stable 2-D and 3-D simulation and visualisation environments.
- The Player server already has an extensive list of support for sensors.

3.3 The Player Project in detail

This section provides an in-depth review of certain aspects of Player [32], including: the Player server, client interface, levels of abstraction, proxies, and drivers. The levels of abstraction between the client program and the robot's hardware (or simulated hardware) is shown in Figure 3.2.

3.3.1 Player server

The Player server is a core component of The Player Project. Essentially, the server sends and receives standard messages to and from the client program and the robot.

Communication between interfaces, other Player servers, and clients is accomplished through TCP/UDP sockets.

3.3.2 Client program

The Player server can be interfaced through any programming language with support for network sockets. The most mature client libraries that are ready to use support development in C (`libplayerc`), C++ (`libplayerc++`), and Python (`libplayerc_py`). In this research project, the C++ client library was used. A basic client obstacle avoidance program looks like:

```
#include <iostream>
#include <libplayerc++/playerc++.h>
#include "avoidance.h"

using namespace PlayerCc;
using namespace std;

int main(int argc, char *argv[])
{
  PlayerClient robot("localhost");
  LaserProxy lp(&robot, 0);
  Position2dProxy pp(&robot, 0);
  avoidance a(&lp); //Avoidance uses data from laser scanner

  while(true)
  {
    double speed, turnrate;
    robot.Read(); //Update data. Note this is blocking!
    a.process(&speed, &turnrate); //Set variables to avoid obstacles
    pp.SetSpeed(speed, turnrate);
    usleep(10);
  }
}
```

The key point here is that the client code is free from low-level hardware details. Note that the function call `robot.Read()` is blocking and only returns when there is new data published from the robot to the Player server.

3.3.3 Proxies

A proxy in Player is a defined standard communication for a particular interface. Proxies provide an additional layer between the client program and hardware drivers and are linked to the robot with Player 'drivers' (see Section 3.3.4). This is beneficial as it generalises the differences between different robot hardware. For instance, client code to read range data from the Player server is the same if the sensor was a SICK LMS-200 or a Hokuyo URG-04LX laser range-finder.

Player comes with many proxies, including `LaserProxy`, `CameraProxy`, `GpsProxy`, `ImuProxy`, `Position2dProxy` (sets motor speeds, gets the robot's position), and many others. Not all proxies connect directly to the robot's hard-

ware. Instead these proxies provide algorithmic features such as path-planning (`PlannerProxy`), vision processing (`BlobfinderProxy`), and a grid-based representation of the environment (`MapProxy`). There are is also a generic proxy (`OpaqueProxy`), useful for integrating custom developed drivers and hardware that do not adhere to the standards of any other proxy.

3.3.4 Drivers

The final level of abstraction between the client program and the robot comprises drivers. The role of a driver is to interface to the hardware, by translating hardware-specific data to a standardised communication format (for an appropriate proxy), and vice versa for sending commands to hardware. Note that these 'drivers' require system device drivers (such as kernel drivers in Linux: serial ports, USB cameras, etc.) to be installed properly.

Player comes with several drivers built-in for use with "off the shelf" sensors and robotic platforms. However, in the case of interfacing to custom hardware, Player allows the use of 'plugin' drivers. This saves the developer from having to re-compile the entire Player server each time new or altered hardware is added to the system.

Most of the existing drivers in Player run on their own thread; Player is multi-threaded. This allows sensors of different bandwidths to be attached to the robot, without 'slow' sensors bottlenecking 'faster' ones.

All drivers are compiled into shared libraries, loaded at runtime by the Player server. Drivers are configured by entering driver-specific details in a configuration file (*.cfg). This file is passed as a command line argument to Player, when starting the server.

Like proxies, Player also allows 'plugin' drivers that are created by developers and end users designed for a particular robot setup. Throughout this research project, several custom plugin drivers were developed and are discussed in Section 7.2.

3.3.5 Stage and Gazebo simulation plugins

Like most robotic development environments, Player comes with its own simulators. Having the ability to simulate a robot in a virtual environment has many benefits, as mentioned in Section 3. There are two such simulators for Player: Stage and Gazebo. Both of these simulators provide their own drivers that replace ones that Player would have otherwise used to interface to robot hardware. Just like the drivers pertaining to the physical robot, the simulator drivers mate with the Player server's proxies.

The Stage simulator provides a 2-D simulation environment, while Gazebo provides a 3-D one. Consequently Gazebo is computationally more expensive to run simulations than Stage. Due to this, Gazebo tends to be used for high fidelity simulations of single robots. Stage, on the other hand, is not demanding on modern systems and can be used to simulate swarms of exploration based robots.

Both simulators provide a GUI, containing a variety of adjustable settings. The

most useful is being able to navigate the user camera's view of the simulation. Note that although Stage is a 2-D simulator, it can be adjusted to give a perspective view of the environment as shown in Figure 3.3 (a). Although Gazebo is a 3-D simulation environment, it is generally easiest and quickest to use bitmaps representing the floor plan of the environment, as seen in Figure 3.3 (b). Gazebo extrudes the areas in the bitmap which have black pixel values, and the user has the option of applying 'texture' to the walls of the environment.

Gazebo makes use of several third-party libraries. These dependencies tend to make installation challenging. The following are the main third-party libraries used by Gazebo:

- Object-Orientated Graphics Rendering Engine (OGRE) for visual display[1].
- Open Dynamics Engine (ODE) for simulating rigid body physics and physical interactions with other objects in the environment[2].
- Bullet 3D physics library[3].
- Geospatial Data Abstraction Library (GDAL) for terrain builder utility[4].

Initially, Gazebo was preferred over Stage for its ability to generate a virtual video feed. This permits development and testing of computer vision algorithms. This is useful in a simulation environment as the ground truth of a value (e.g., depth to an object or pose of a camera) is easily found. However, it was found that the textures generated in the virtual environment are too repetitive to be useful for vision processing tasks. Furthermore, Gazebo is resource intensive, and many computers would struggle to run Gazebo by itself let alone the client application as well. For these reasons, Stage proved to be more useful, and Gazebo was rarely used.

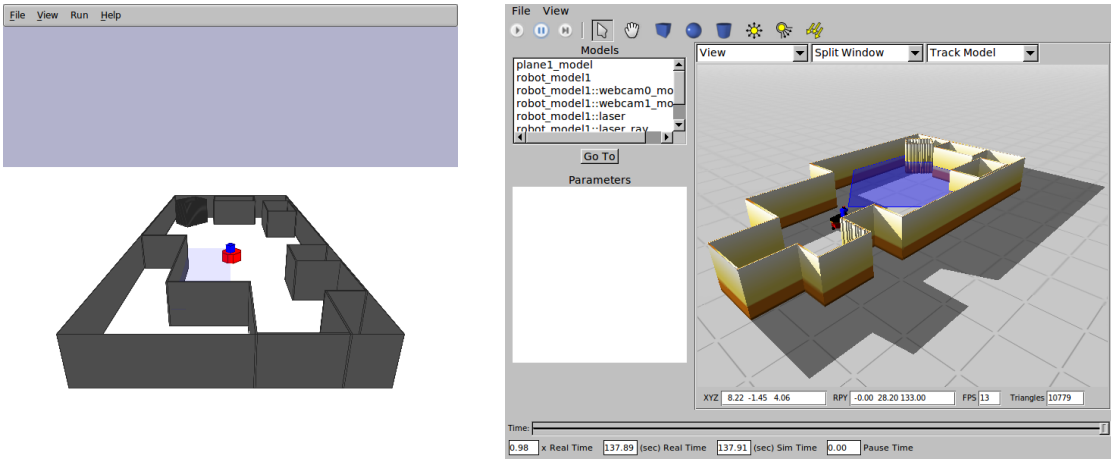
3.3.6 Utilities

Player also comes with a selection of useful debugging utilities. Using a combination of these utilities, one can immediately control robots in Player without even needing to write a client program. The main ones that proved to be useful throughout this research were:

`playerv` — allows quick visualisation of sensor data (including from camera and map proxies), as well as controlling devices such as motors.

`playernav` — provides control over `localize` and `planner` devices. This was particularly useful when testing Player's localisation driver (`amcl`, see Subsection 5.3). `playernav` can show all of its pose beliefs and test its ability to handle the kidnapped robot problem (see Section 5.2.3) by dragging the robot

[1] `www.ogre3d.org/`
[2] `www.ode.org/`
[3] `www.bulletphysics.com/`
[4] `www.gdal.org/`



(a) Stage simulation.

(b) Gazebo simulation.

Figure 3.3: Visualisations for The Player Project of a simple indoor environment.

to a different location on the map. `playernav` was also useful for testing path planning drivers: a goal can be easily set, and following that, display the waypoints to the goal.

`playerjoy` — permits driving the robot around using velocity control with a joystick device.

`playervcr` — allows control over data logging and playback, using `log` devices.

Chapter 4

Simultaneous localisation and mapping

Simultaneous localisation and mapping (SLAM) is a technique used by autonomous vehicles to build up a map of an unknown environment using sensory inputs, while concurrently localising themselves [38]. Once a map has been built, global navigation tasks such as travelling towards a destination autonomously can be accomplished in a relatively straight-forward manner (see Section 5.2).

For a smart wheelchair navigation system, a map of an environment generated through SLAM is very useful. Firstly it permits the system to adaptively learn the layout of an unfamiliar environment. It also allows for 'clever' navigation algorithms that do not get stuck in trap situations, or confused by maze-like environments. Moreover, it provides the ability for the system to autonomously navigate to a desired location in the map. Chapter 5 discusses how the map can be used in a navigation system. Note that this research project concerns indoor operation of an autonomous wheelchair. Therefore, this chapter concentrates on SLAM techniques suitable for indoor environments.

4.1 Challenges with SLAM

The two main tasks in SLAM (generating maps and localisation) create a 'chicken or the egg' problem [39]. An unbiased map is required for localisation, while an accurate position and pose estimation is needed to build a map. Moreover, errors in the map will propagate into the localisation estimate and vice versa [40]. These tasks make SLAM, a hard problem to solve in practice. Subsections 4.1.1–4.1.4 outline various challenges in SLAM, and techniques employed to overcome them.

4.1.1 Measurement errors and outliers

In any real world system, sensor measurements contain errors — with noise always being a contributing factor. This creates a challenge in SLAM, as noise from subsequent measurements is statistically dependent. This is the case as errors accumulate, the interpretation of future measurements depends on the interpretation of the pre-

vious measurements. Therefore, accommodating sensor noise is a key to building maps successfully and is also a key complication factor in SLAM [41].

A popular choice used to model noise is that of a normal (or Gaussian) distribution, and its probability density function (PDF) is:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad . \tag{4.1}$$

Equation (4.1) is often abbreviated as $x \sim \mathcal{N}(\mu, \sigma^2)$. In many cases, a normal distribution adequately models sensor noise and is assumed in filtering techniques including the Kalman filter (see Section 4.2.2).

The characteristics of measurement noise vary depending on the type of sensor being used. For example, with SLAM using sonar sensors, it is common to have dropped samples and false returns [42] (whereas with laser scanners, this does not tend to happen). Bailey [43] offers a solution to this: before a new measurement is added it is first cross-referenced with features already present in the map. The sample is kept for some time and added to the map if it is re-observed (and discarded if otherwise). Thus SLAM implementations are sensor specific, whereas the characteristics of a particular sensor's error uses heuristics.

Due to progress in technology, modern robots are also making use of information rich sensors such as cameras. However, cameras are notoriously susceptible to shot noise (or 'salt and pepper' noise), which if not dealt with will cause algorithms that assume normal distributions to fail. Fortunately, provided that the image is not swamped with shot noise (which occurs in low light levels), simple techniques such as applying a median filter work well. Another popular method to remove outliers in a data set is via the RANSAC (an acronym for random sample and consensus) algorithm [44].

4.1.2 Data association

Data association is the process of matching observations of a landmark to subsequent observations of the same landmark. It is also arguably the most critical aspect of the SLAM algorithm [43]. Correct alignment of landmarks is crucial for generating consistent maps, otherwise miss-assignment for extended Kalman filter (EKF) based SLAM methods diverge [45].

In certain environments, static landmarks such as indoor furnishings are often used for feature-based data association. Vision based SLAM systems often use SIFT or SURF algorithms to provide a similarity measurement of landmarks [46]. Landmarks can also be simple geometric primitives, such as points or lines that are distinguishable only by their location [43]. An alternative to this is to perform scan correlation [43], where subsequent scans of unprocessed sensor data (e.g., from a 2-D laser scanner) are matched. This is used in environments that lack 'good' features but instead have surface 'texture', such as tiled walls.

When landmarks or scans are incorrectly matched, this can introduce substantial errors. Thus to avoid invalidating the entire process, most SLAM systems go to considerable lengths to ensure that the data association process is reliable. Subsections

4.1.2.1–4.1.2.4 describe some of these approaches.

Gated nearest neighbour

One approach often used in tracking problems is the gated nearest neighbour (NN) algorithm [45]. This technique firstly uses the normalised squared innovation[1] test to determine feature compatibility, followed by applying the NN rule (smallest Mahalanobis distance) to select the best matchings. As outlined by Neira and Tardos [45], the advantages with gated NN for data association is its conceptual simplicity and $\mathcal{O}(mn)$ computational complexity. However, gated NN neglects to consider the correlation between features. According to Neira and Tardos [45], this makes gated NN sensitive to increasing position and sensor errors, and consequently the probability of a miss-assignment of features increases. Moreover, gated NN does not reject ambiguous measurements.

Joint compatibility branch and bound

An improvement to the gated NN algorithm is the joint compatibility branch and bound (JCBB) technique [45], a tree search-based algorithm. The JCBB makes use of the correlations between the measurements and mapped features, making it more robust to spurious matchings. However, it is an exponential time algorithm $\mathcal{O}(1.53^m)$ [45], so the number of observations needs to be limited for the algorithm to run in real time. In order to reduce computational complexity, heuristics (such as using the NN rule for branching) are used to reduce the search space.

Randomised joint compatibility

A simple extension to the JCBB technique (see Subsection 4.1.2.2) is to randomly split the tree into two data sets: a guess and a proof [47]. The approach is analogous to RANSAC and data fitting: a guess hypothesis is made by applying JCBB on a small number of measurements chosen at random, and this hypothesis is tested by using gated NN on the remaining data. The process is completed several times, after which the best hypothesis is selected using the same criteria as JCBB [48]. This is an improvement over JCBB as it significantly reduces complexity by reducing the size of the search tree's hypothesis space [47].

Scan matching

According to Bailey [43], feature based data association is viable provided that there are landmarks in an environment that can be classified as geometric primitives[2]. However, in some environments this may not be appropriate, and instead, more reliable association is possible using raw data. In the past, SLAM algorithms have used scan matching on raw laser scans with success. Further, they have been shown

[1] Innovation is the difference between the measurement and its expected value according to a model.

[2] The simplest geometric primitive is a point and a straight line segment.

to present a more general solution [49] where the main advantage is that they are not tailored for one particular environment.

4.1.3 Loop closure

Even with the best SLAM implementations, over time the error in the robot's pose grows without bound as the error from the sensors accumulate. When a previously visited place is re-visited, many approaches aim to use this information and correct the accumulated errors. This process is called loop closure and it can be seen as a form of data association. Therefore, in most SLAM systems, loop closure is essential for accurate long-term positioning. Loop closure is a difficult problem to solve, especially due to changes in the robot's view point and dynamic objects in the environment. An incorrect loop closure can ruin a map in a similar way to the case of data association.

In the past, the loop closure problem has been approached in many different ways. However, the viable methods are somewhat dictated by the choice of sensors. In the case of planar laser scanners, one approach is to perform raw sensor data recognition, where loops are detected by comparing laser scans. An example of laser scan matching is the work done by Granstrom et al. [49], where a scan is described by rotation invariant features. Scan matching is also covered in Subsection 4.1.2. Note that this approach is not suitable for many sensors.

In the case of monocular vision SLAM, Williams et al. [50] evaluates three different loop closure detection methods. The three methods compared were:

Map-to-map — a loop closure is detected if there is at least five common features between the first and last sub-maps. The geometric compatibility branch and bound algorithm (GCBB) is used to evaluate the number of correspondences between common features in different sub-maps.

Image-to-image — detects loop closures by comparing the similarity between the latest camera image and previous seen places. This is achieved by finding SURF[1] features in the current image to a visual vocabulary. The visual vocabulary is created also from SURF features in training images.

Image-to-map — loop closure is detected by finding a common camera trajectory between the first and last sections of a loop. The pose of the camera is found by finding correspondences between features in the image and features in the map. Since a single pose is not able to determine scale difference[2], the camera is tracked for some time.

Williams et al. [50] concludes that the map-to-map method is unsuitable for sparse maps, the image-to-image method works well, but the image-to-map method

[1]SURF is an acronym for the Speeded Up Robust Feature computer vision algorithm often used in object recognition.

[2]Since monocular SLAM often lacks odometry and only gives bearing measurements, the map being created contains scale ambiguity. Thus to cope with scale ambiguity, a common trajectory can be used to accommodate for scale differences.

is found to work best. This is said to be the case as the image-to-map method makes use of geometry information to prune more false positives and that "it is best to take as much information as is feasible into account when detecting loop closures" [50].

4.1.4 Computational complexity

A significant obstacle for implementing and deploying a real-time SLAM system to operate, in large and cluttered indoor environments, is managing computational complexity. A representation of a detailed 2-D or 3-D map of an environment requires thousands of numbers. Thus from a statistical point of view, the mapping problem can be extremely high-dimensional [41]. In the case of EKF-SLAM, performing an update of a covariance matrix[1] with K landmarks is $\mathcal{O}(K^3)$ [51]. The complexity can be reduced to $\mathcal{O}(K^2)$ by taking advantage of the sparse nature of typical observations. Even with this improvement, real-time EKF-SLAM (and many other SLAM approaches) clearly becomes intractable with large numbers of features.

There are several options to reduce the computational complexity of SLAM, including: limiting the number of features, sub-optimal updates, and using alternative map representations. In MonoSLAM [52] and other vision-based SLAM approaches, only 'visually salient features' are used, thus providing an effective means of limiting the number of features. Sub-optimal updates are also used to reduce complexity, although typically at the loss of information [51]. Guivant et al. [53] notes that only a subset of the map's features during a particular observation needs to be updated, resulting in a significant reduction of computation for large maps. Another approach to reduce computation is to use an alternative map representation, and in particular one that is more suited to the sensor being used. More on alternative map representation is discussed in Section 4.3.

4.2 Filtering techniques

According to Thrun [41] virtually all of the 'state of the art' SLAM algorithms are probabilistic. This comes from the fact that robot mapping is characterised by uncertainty and sensor noise. Such algorithms often employ probabilistic models of the robot, its sensors, and its environment. Also according to Thrun [41]: "the basic principle underlying virtually every single successful mapping algorithm is Bayes rule". Prior to discussion on filters used in SLAM (e.g., Kalman and Particle filters), a summary of Bayes filters from [41] is provided as background.

4.2.1 Bayes filter

Suppose the quantity we want to estimate is x (e.g., the robot pose and the map), given d (e.g., the sensor data), the prior $p(x)$ (e.g., the previous pose and map), and

[1] A covariance matrix is the relation between two variables, e.g., the robot's pose or the landmarks in the environment at some time t.

the normalisation constant η. Bayes rule is:

$$p(x|d) = \eta p(d|x)p(x) \ . \tag{4.2}$$

Equation (4.2) can be extended to a generic Bayes filter, as shown in (4.3). Note that d has been separated into sensor measurements (z) and control commands (u), and the Bayes filter is recursive,

$$p(x_t|z_{1:t}, u_{1:t}) = \eta p(z_t|x_t) \int p(x_t|u_t, x_{t-1}) \ p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) dx_{t-1} \ . \tag{4.3}$$

In the SLAM problem, the state x_t contains both the unknown map and the robot's pose. Since both of these quantities may influence sensor interpretation at any instant, both the map and the pose need to be estimated at the same time (hence, simultaneous localisation and mapping). Suppose that the map is m, the robot's pose is s, and that the environment is assumed to be static, the Bayes filter for the SLAM problem becomes:

$$p(s_t, m|z_{1:t}, u_{1:t}) = \eta p(z_t|s_t, m) \int p(s_t|u_t, s_{t-1}) \ p(s_{t-1}, m|z_{1:t-1}, u_{1:t-1}) ds_{t-1} \ . \tag{4.4}$$

Notice that the posterior in (4.4) is intractable, as it involves a probability distribution over a continuous space. To overcome this, practical SLAM algorithms make further assumptions and approximations.

4.2.2 Kalman filter

The Kalman filter (KF) [54] was introduced in the 1960's and has since has become an instrumental tool in control theory and robotics. The KF is a special case of Bayesian filtering under the linear-quadratic Gaussian (LQG) assumption [55]. That is, KFs are Bayes filters with posteriors represented by linear functions with added Gaussian noise. Taking (4.3) for instance, (4.5) replaces the state transition posterior $p(x_t|u_t, x_{t-1})$, and (4.6) replaces the observation posterior $p(z_t|x_t)$:

$$x_{t+1} = A_t x_{t-1} + B_t u_t + \epsilon_t \ , \tag{4.5}$$

$$z_t = C_t x_t + \delta_t \ , \tag{4.6}$$

where A_t is the state transition model, B_t is the control input model, C_t is the observation model, ϵ_t and δ_t represent zero-mean Gaussian noise (i.e., $\epsilon_t \sim \mathcal{N}(0, R_t)$ and $\delta_t \sim \mathcal{N}(0, Q_t)$, where R_t and Q_t are covariances) introduced by the state transition / observation process.

Kalman filtering consists of an iterative prediction-update process, as depicted in Figure 4.1. This process is usually carried out alternatively, but if for some reason observation data is lacking in an update cycle, it may be skipped and consecutive prediction steps may be performed [56].

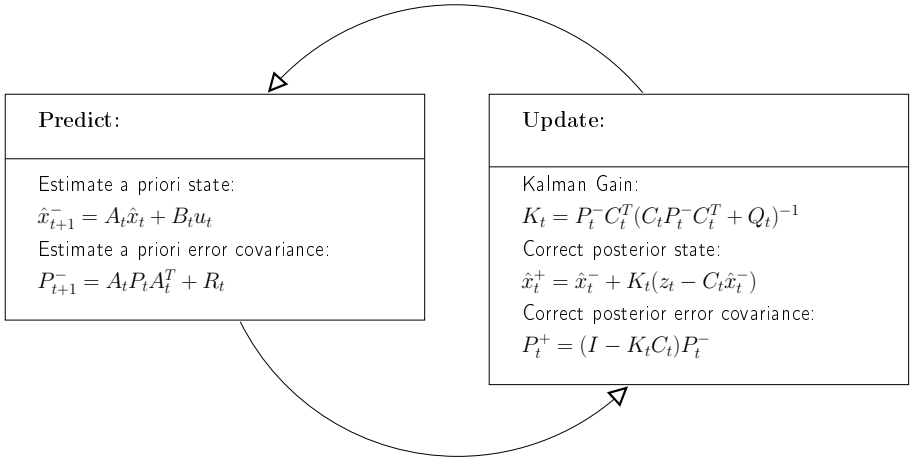


Figure 4.1: Illustration of the Kalman filter's predict-update process.

Extended Kalman filter

Many robotic applications require modelling of a non-linear process, which violates the linearity assumption used in the KF. A solution is the extended Kalman filter (EKF), which relaxes this criteria. This is done altering (4.5) and (4.6) to use arbitrary non-linear, differentiable functions g and h:

$$x_t = g(u_t, x_{t-1}) + \epsilon_t \ , \tag{4.7}$$

$$z_t = h(x_t) + \delta_t \ . \tag{4.8}$$

Note that to conserve the Gaussian model, g and h are linearised about the point of the state mean, using a first order Taylor series expansion [43, 55]. The general structure of the equations in Figure 4.1 is similar for EKFs but with A_t, B_t and C_t being replaced by their respective Jacobians. A derivation of the EKF can be found in [57].

Complexity

The KF is a recursive estimator and, compared to other estimating techniques, does not require storing previous observations and/or estimates. The main complexity in the KF or EKF is finding the inverse of a matrix, which is required to compute the Kalman gain K_t. The fastest algorithm for finding the inverse of a matrix is the Coppersmith-Winograd algorithm, which has a complexity of $\mathcal{O}(n^{2.376})$. However, in the case of SLAM, the most expensive operations are instead matrix multiplications [41], which have a complexity of $\mathcal{O}(K^2)$ where K is the number of features in the map. With a quadratic complexity, clearly KF or EKF based SLAM becomes intractable when dealing with a high number of features.

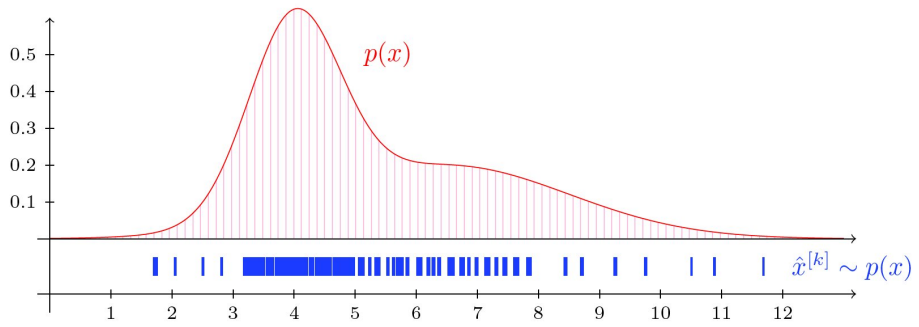


Figure 4.2: Illustration of the proposal distribution, where the re-sampling step of a particle filter draws particles with the probability proportional to their importance weights (source: [60]).

4.2.3 Particle filter

The particle filter (PF) is an alternative non-parametric implementation of the Bayes filter. Instead of assuming a uni-modal Gaussian PDF (as in the case of KF/EKF), the PF approximates the posterior with an arbitrary, non-Gaussian, multi-modal PDF [58], using a set of particles

$$\mathcal{X}_t = \{x_t^{[1]}, x_t^{[2]}, x_t^{[3]}, \ldots, x_t^{[M]}\} \ , \tag{4.9}$$

where M is the number of particles. In other words, each particle in $\mathcal{X}_t$ represents a possible state i.e., a robot trajectory and a map.

As described by Stachniss et al. [59], there are three main steps in a PF:

1. Compute the next state distribution by sampling from the proposal distribution π. The proposal distribution $\pi(x_t|z_{1:t}, u_{1:t})$ approximates the true proposal distribution $p(x_t|z_{1:t}, u_{1:t})$ [38].
2. Assign importance weights to each particle. The weights account for discrepancy between π and the true proposal distribution [59]. Particles whose predictions match the observation are given higher weights.
3. If required, perform re-sampling. The probability that a particle survives is proportional to its importance weight, as depicted in Figure 4.2. Particles that survive are then assigned a uniform weight [38]. Subsection 4.2.3 describes the re-sampling process in more detail.

Subsections 4.2.3–4.2.3 cover particular aspects of particle filtering in more detail, in particular: the re-sampling step, exploiting state space dependencies, and complexity.

Re-sampling

A major problem with particle filters is that the particle weights degenerate. In addition, maintaining lowly weighted particles is a waste of computational resources.

The solution is to re-sample, where the aim is to force the PF to concentrate on areas that matter most [60], i.e., on particles with higher weights. Thus re-sampling removes lowly weighted particles and replicates highly weighted particles (and the number of replications is proportional to the weight).

However, the re-sampling process creates another problem known as 'particle depletion'. Particle depletion is where after a few iterations, all but one particle will have negligible weight [60, 61]. In the case of PF-based SLAM, a lack of particle diversity creates issues for active loop closing schemes and leads to map divergence [62]. Numerous re-sampling techniques have been invented to maintain a diversity of particles, such as the approach described by Grisetti et al. [62] in GMapping (see Subsection 4.4.4).

Rao-Blackwellisation

Although a PF can be used to solve both the localisation and mapping problem in SLAM, the number of particles required to represent the posterior grows exponentially with the dimension of the state space [63]. This is a severe limit of a standard PF, which is not suited to the high dimensionality of the mapping problem. Using the observation by Murphy [64], FastSLAM (see Subsection 4.4.2) exploits the idea that knowledge of the robot's true path allows landmark positions to be conditionally independent. Thus to make PF-based SLAM tractable, FastSLAM factorises the Bayes posterior by using a particle filter to estimate the robot's path, and a KF/EKF to track a feature in the map [63]. This is a similar case with other PF-based SLAM algorithms. The factorisation is also known as Rao-Blackwellisation [38, 63–65], and thus the combination of particle and Kalman filtering in SLAM becomes a Rao-Blackwellised particle filter (RBPF)[1].

Complexity

Since particle filters approximate the posterior with a finite number of samples, the complexity of a PF can be regulated during the re-sampling phase (see Subsection 4.2.3). Techniques that actively control the number of samples used to represent the posterior are called adaptive [60]. Adaptive techniques are thus well suited real-time robotics applications, as they adapt to the computational resources available [57]. The more samples used, the closer the estimated posterior becomes to the true posterior; PF allows trade-off between accuracy and computational effort.

The complexity of a PF is proportional to the number of particles M used i.e., $\mathcal{O}(M)$. The number of particles required depends on the noise parameters and the structure of the environment [57, 60, 63, 66]. The complexity also depends on how the PF was implemented. For instance, FastSLAM exploits dependencies of the state space. This discussed in Subsections 4.2.3 and 4.4.2.

[1]The RBPF is also known as the marginalised PF.

4.3 Map representation

There are many approaches to map representation in SLAM. However, choice is often dictated by the sensor(s) being used and the size of the environment. In the case of range-bearing laser scanners, occupancy grids (see Section 4.3.1) is often a popular choice. Instead, if a vision based approach with a camera is being used, feature maps (see Section 4.3.2) are preferred. This section reviews a selection of map representations and analyses their suitability to a dynamic, medium sized indoor environment.

4.3.1 Occupancy grids

The occupancy grid (OG) mapping technique uses a fixed resolution grid to represent the environment. Each cell of the grid is used to store the probability of whether there is an obstacle there or not. Introduced by Elfes [67] in the late 1980's, occupancy grids have since been widely used due to its simplicity. They are well suited to mapping 2-D indoor environments. Compared to other mapping techniques, grid mapping has several advantages including:

- Simple to implement and easy to view for debugging purposes.
- Dense representation of the environment.
- Allows the use of an *a priori* map. Thus if the environment is known, a user could input a floor plan of a building along with the pose of the robot. Occupancy grid mapping can also build on an incomplete map.
- Explicit representation of occupied and free space, which is useful for path planning methods [43].

An approach by Schultz et al. [68] uses both short term and long term grid maps. The short term map contains recent sensor data, and as it does not contain significant odometry error, is used for obstacle avoidance and localisation. Once a short term map has matured, it is then added to the long term map, which is used for navigation and path-planning. This method has been shown to improve the robustness of grid mapping in dynamic environments [43].

Although occupancy grids are able to represent sensor errors, they cannot represent the vehicle pose uncertainty or their correlation. Other disadvantages include:

- A rectangular grid is not an efficient representation of a non-rectangular environment.
- There is a trade-off between grid resolution and computational complexity. Thus for large environments, it is not computationally feasible for OG to create fine detailed maps.
- Limited ability to cope with dynamic environments.

- Assumes that errors associated with sensor readings are independent. However, this is not the case. It becomes apparent for large environments when errors accumulate. Hence, occupancy grids cannot provide consistent global maps when dealing with large environments.

- It is not possible to perform loop closure, which is the main downfall of grid based SLAM. As Bailey [43] mentions, even if it was possible for a loop to be detected in grid mapping (it would be extremely computationally expensive), there is still no mechanism for correcting accumulated errors.

4.3.2 Feature maps

Unlike occupancy grid mapping (see Section 4.3.1), feature maps [69] only keep track of features (and not free space). This leads to a very efficient representation of the environment. Features are represented as geometric primitives: if a 2-D laser scanner is used, the raw data is pre-processed for landmarks such as corners, lines, etc. In the case of vision based approaches, a frame is often analysed for SURF or SIFT based descriptors. Localisation is achieved by tracking these suitable features. Note that each feature in the map has a set of coordinates in Cartesian space.

Unfortunately, feature maps also have their disadvantages. Since they do not store any information about free space, feature maps cannot be immediately used for path planning or obstacle avoidance. In order to do so, an occupancy grid representation of the feature map must be performed [70]. Other disadvantages include:

- To some extent, its implementation is environment specific. For instance, in the case of a laser scanner, if the walls of a building contain arbitrary curves, extraction and classification of the environment as points and lines will fail. In the case of vision based method, a plain wall also creates a similar problem (known as the aperture problem).

- Since the data is processed for features, there is less information to prune false positive data associations. If data association fails, this leads to significant increases in error in both the pose and map estimation.

- *A priori* maps are harder for a user to input, as they cannot simply be floor plans of a building. They need to contain only features that the feature extraction method used would have extracted.

4.3.3 Topological maps

Topological maps (or graph based maps), unlike occupancy grid or feature maps, store the map of the environment using a graph data structure. In the graph, each vertex represents robot and landmark poses, while the edges contains constraints on the relative poses of the two nodes. Since topological maps do not rely on metric measurements, it avoids some of the difficulties scaling to large environments. Additional advantages include:

- Efficient and compact representation of the map.
- Logical organisation for tasks such as path planning. The map is stored as a graph and this makes it possible to use existing graph algorithms, such as finding the shortest path between non-adjacent vertices [43].
- Loop closure on a local graph-based map is automatic and straight forward, as it just involves adding an edge between the start and end vertices [40]. When a global map is required, a t-spanner [71] can be used to close distant gaps introduced by the loop closure process [40].

However, just like any other map representation method, topological maps have their disadvantages. Place recognition is a form of data association. Thus the key weakness in the method is that each place (vertex) in the environment (map) must be able to be described uniquely. Otherwise, if two different places appear similar, a loop closure will be attempted, resulting in data association failure. According to Bailey [43], most topological SLAM systems use one or more types of place recognition. An example is a system that uses both range-bearing and vision sensors. Bailey [43] also suggests that this problem can be overcome by introducing metric information.

4.3.4 Hybrid maps

Hybrid maps, which consist of both metric and topological maps, have been developed to take advantage of the complementary strengths that metric and topological maps provide [43]. Metric mapping techniques (such as occupancy grid and feature mapping) provide: high local accuracy, a detailed representation of the environment, and pose constrained data association. Whereas, topological mapping are beneficial as they provide: low computation and storage, automatic loop closure, correction for accumulated errors after a loop closure, and scaling to large environments.

An example of an early hybrid mapping system is that introduced by Thrun [72]. It uses a artificial neural network (ANN) to map several sensor measurements into probabilities of occupancy in an occupancy grid. A topological map is then created on top of the grid map, firstly by finding equidistant paths in between obstacles in the environment. The path is then broken up in 'critical' points and lines, giving the topological graph. The advantage of turning a grid-based map into a topological map is to use it for path planning. However, as Bailey [43] points out, this is still limited to small environments as it only creates a single grid map for the entire environment.

Another hybrid mapping scheme is the *Atlas* framework [73]. *Atlas* uses existing metric mapping techniques to create independent local maps, and then uses a graph structure to store the transformation between each local map[1]. Due to this, *Atlas* is often referred to as a sub-mapping technique [74]. To produce a global map, the Dijkstra projection is used to find the pose of each submap relative to a single reference frame. However, if there are loops in the graph, the Dijkstra algorithm

[1]Bailey [43] also develops a similar mapping technique, called 'Network Coupled Feature Maps'.

will not produce a consistent global map as it does not consider all edges in the graph. To fix this, it is followed by a non-linear least squares optimisation [73].

Since the size of the local maps is often limited[1], many hybrid mapping schemes are fast, incremental, and can run in real-time. However they often do not require constant time: since the time it takes to correct a loop depends on its size, the hybrid approach in some cases [41, 76], strictly speaking, is not a real-time algorithm.

4.4 Open-source SLAM implementations

There are several freely available open-source SLAM implementations available, and many of these are publicised through the OpenSLAM[2] website. My intention is to either adopt or extend an existing SLAM implementation, as well as getting it to work with Player (see Section 3.3). This section reviews a number of open-source SLAM packages primarily found on the OpenSLAM website.

4.4.1 EKF-SLAM

The pioneering SLAM work by Smith and Cheeseman [77], carried out in the late 1980's, used EKFs to build a stochastic map[3] of spatial relationships. Other pioneering work by Leonard and Durrant-Whyte [69] followed, which also used EKFs to estimate the point locations of environment features in a global reference frame. At the time, EKFs were used as they gave probabilistic representations of spatial uncertainty, as opposed to min-max bounds [78] that simply gave worst case error bounds.

A key weakness of pure EKF-based approaches is that they make strong assumptions about the robot motion model and assume sensor noise to be normally distributed. Moreover, they also assume that landmarks are uniquely identifiable. If any of these assumptions are violated, the filter is likely to diverge, resulting in an inconsistent map [62]. Also, since readings are not independent, one incorrect measurement will affect all other measurements.

EKF-SLAM has quadratic complexity for each observation, thus for K landmarks, it requires $\mathcal{O}(K^2)$ computation and storage[4]. Also, the failure of EKF-SLAM has been proven to be time dependent, thus making it unsuitable for long term SLAM [79]. Note that although there are numerous open-source implementations of EKF-based SLAM for range-bearing sensors and odometry available, it has been proven numerous times that EKF-SLAM is outperformed by modern SLAM techniques (namely RBPFs, such as FastSLAM) [63, 80].

[1] Approaches to limit size of a sub-map include restricting its radius [75], or imposing some limit on a measure of map complexity [73].

[2] `http://www.openslam.org/`

[3] The stochastic SLAM algorithm stores the robot's pose and the locations of the features in a single state vector. A matrix of covariances between the estimates in this vector is also maintained [77].

[4] Recent methods have been applied to EKF-SLAM to reduce complexity, namely sub-mapping and through a process called locally partitioned data-fusion [79].

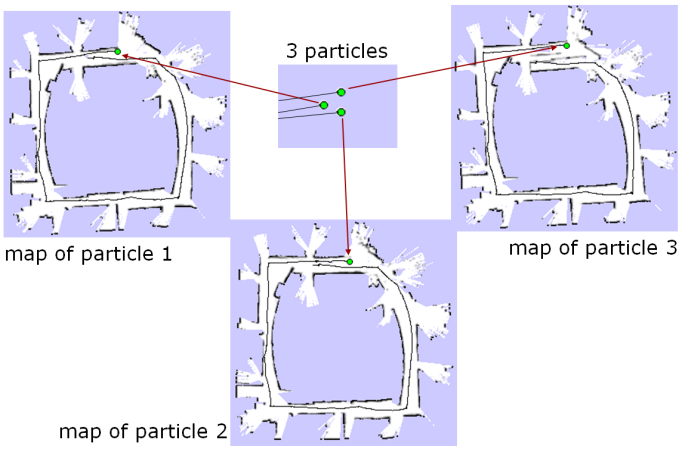


Figure 4.3: Each particle in FastSLAM has its own map representation (source: [82]).

4.4.2 FastSLAM

The FastSLAM algorithm [63], unlike other EKF-based SLAM algorithms at the time, was the first to directly incorporate a non-linear process model and non-Gaussian pose distribution [38]. This was achieved by using a RBPF, where each particle in the PF represents a hypothesis of the vehicle's trajectory (and each trajectory has K EKF's for K landmarks the map). FastSLAM was introduced to overcome the problems with EKF-SLAM, namely: computational complexity, non-linearity, and data association [81]. Originally, FastSLAM created feature maps, however, it has been extended to use occupancy grids (i.e., DP-SLAM, see Subsection 4.4.3). Although there are no 'official' releases of FastSLAM code from the original authors, other researchers have posted their own implementations of FastSLAM online which are available for use.

There are two main problems in SLAM: localisation and landmark location estimation. The approach in FastSLAM is to use separate estimations for the two problems. FastSLAM's factored posterior is recursively computed by:

$$p(s_{1:t}, l_{1:K}|z_{1:t}, u_{0:t-1}) = p(s_{1:t}|z_{1:t}, u_{0:t-1}) \cdot \prod_{i=1}^{K} p(l^{[i]}|s_{1:t}, z_{1:t}) \ , \tag{4.10}$$

where K landmarks make up the map m, and $l^{[i]}$ is the i-th landmark. The landmark estimation uses an EKF for each feature in the map. On the other hand, the path[1] estimation uses a PF, where each particle has its own map representation as shown in Figure 4.3. Sampling multiple hypothesis of the robot's path allows data association to be made on a per-particle basis. It also allows FastSLAM to be used in environments with highly ambiguous landmark identities [63]. Particles that predict the correct data association are weighted higher and, therefore, give better estimates of the robot's path. Consequently, lowly weighted particles are removed.

A key improvement that FastSLAM provides over EKF-SLAM approaches is

[1]Particles in FastSLAM represent an entire robot path history and associated map of landmarks, as opposed to a momentary robot pose [81].

that it is significantly faster and therefore can handle larger and/or more detailed environments. The separation of the pose and landmark estimations plays a key role in the speed of FastSLAM, as it allows the position of landmarks to be conditionally independent (and thus only low-dimensional EKFs are required). Furthermore, a balanced binary tree is used to store landmark estimates, thus for K landmarks, the total complexity to update M particles is $\mathcal{O}(M \log_2(K))$. Note that the worst case is $\mathcal{O}(MK)$, which occurs if the landmark tree becomes grossly unbalanced.

FastSLAM is capable of performing loop closure, where the ability to effectively close loops depends on M. Note that the 'online' FastSLAM usually only applies the reduction in uncertainty to the current pose (although it could be applied to reduce the uncertainty of the landmarks in the map as well) [63]. FastSLAM can also be adapted to handle dynamic environments. This can be achieved, once a map of the environment has been created, by using particle filters to model people and other moving landmarks [63].

Despite the substantial improvements that FastSLAM had over other SLAM algorithms at the time, it still has its draw backs. Bailey et al. [81] shows that the algorithm degenerates with time. This is due to a property of particle filters: the variance of the particle weightings increases with time, and eventually all but one particle will be of negligible weight. Although this problem is alleviated by re-sampling, the particles that are not re-sampled results in loss of landmark estimates [81].

4.4.3 DP-SLAM

The distributed particle mapping algorithm (DP-SLAM) [66, 83], is an extension to FastSLAM. The key difference between itself and FastSLAM lies in its representation of the world, where it uses probablistic occupancy grids (as opposed to EKF tracking of features/landmarks). This permits DP-SLAM to be operated where feature tracking is difficult, such as in environments that lack features or have repeating patterns. DP-SLAM is designed to be used with a highly accurate laser range-finder and a deterministic environment [83]. DP-SLAM has also been released to the public through the OpenSLAM website.

In practice, DP-SLAM uses a couple of magnitudes more particles than FastSLAM does. Note that the meaning that each particle carries also differs between the two methods. In DP-SLAM, each particle tracks only the robot pose (instead of the robot path in FastSLAM). Also, to permit real-time operation, DP-SLAM only generates a single occupancy grid map[1]. Each cell of the grid map has a tree containing an ancestry of observations from many particles. Provided that the ancestry tree remains balanced, for P particles it takes $\mathcal{O}(\log_2(P))$ to insert a new observation. The ancestry trees are maintained to prevent them growing indefinitely [83].

Just like FastSLAM, DP-SLAM actively closes loops and thus requires no special loop closure detection. This is of course provided that the loop is not too big; in the absense of loop closing heuristics, there must be sufficient particles for loop closure. Eliazar and Parr [83] mentions that DP-SLAM can close loops of over 50 m.

[1]Having a map for each particle in DP-SLAM is not practical, as a lot of time would be wasted in copying data [83].

As with any approach that uses occupancy grids for map representation, DP-SLAM is more demanding on resources than feature tracking methods like FastSLAM. For fast access to the ancestry trees, the entire occupancy grid must be kept in memory. A 'naïve' implementation of DP-SLAM requires both time and space complexities of $\mathcal{O}(MP)$, where M is the size of the map. Note that the complexity can be reduced by imposing bounds on the depth of the ancestry trees.

Grid resolution is also a key factor in complexity. Eliazar [66] notes that coarser grids are faster and use less memory, at the expense of an increased amount of drift and the inability to represent fine details. Grids that are too fine will also lead to accuracy problems, due to the model in DP-SLAM of the laser scanner rays. A grid size of 3–5 cm was reported to work best, and 10 cm grids were also sufficient.

4.4.4 GMapping

Like FastSLAM, GMapping [62] also uses RBPFs where each particle has its own map. Instead of tracking features, GMapping uses scan matching, which is known to produce more reliable data association in environments where it is difficult to define parametric feature models [43]. GMapping uses a grid-based map representation, and despite its computational disadvantages compared to feature based maps, it can represent arbitrary features [62]. GMapping has also been released to the public through the OpenSLAM website.

The key contribution of GMapping is its adaptive re-sampling technique, which is targeted at reducing the problem of particle depletion. GMapping uses two approaches to combat particle depletion. The first is a better proposal distribution for the new particles, by taking into account the last laser scan (as opposed to just the last odometry reading, as in the case of FastSLAM and DP-SLAM) [62]. The second is an adaptive re-sampling technique, which only performs re-sampling when needed, and thus a diversity of particles is maintained.

A study performed by Balaguer et al. [84] compared the performance of GMapping and DP-SLAM, on both simulated and real datasets. In that study, GMapping was found to be more reliable as it produced consistent maps. The tests also included the effect of additive Gaussian noise to the odometry. Although this type of noise is not a good model for errors in odometry readings, it was found that GMapping was able to handle additive noise of up to 10 %.

4.5 Reasons for selecting GMapping

During the course of research, several SLAM implementations were evaluated. GMapping was found to be most suitable to this project, primarily because its core algorithms are avaliable as a library. This makes integrating it into Player a relatively straight-forward that requires development of a wrapper (which is discussed in Subsection 7.2.6). Also, a GMapping driver has been implemented in ROS. This is useful as ROS has a similar structure to Player, and thus provides a useful reference to interfacing to the GMapping library. Moreover, GMapping uses relatively modern SLAM techniques (RBPFs) and has been reported to perform well [84].

Chapter 5

Navigation

A crucial component of any autonomous/semi-autonomous robotic system is its ability to navigate its way through an environment. Navigation involves two main areas:

Local navigation — used for navigation over short distances, often less than the maximum range of the environment / obstacle detecting sensor(s). The main role of a local navigation module is to provide obstacle avoidance. Current sensor data is used, enabling it to handle dynamic environments.

Global navigation — also known as path-planning, is used for travelling between two points, and to do so, requires a map (i.e., one built during SLAM). This permits the system to autonomously navigate to a user specified destination, providing a high level of autonomy. Global navigation methods also have the ability to return efficient routes to a destination, essential for operation in challenging environments.

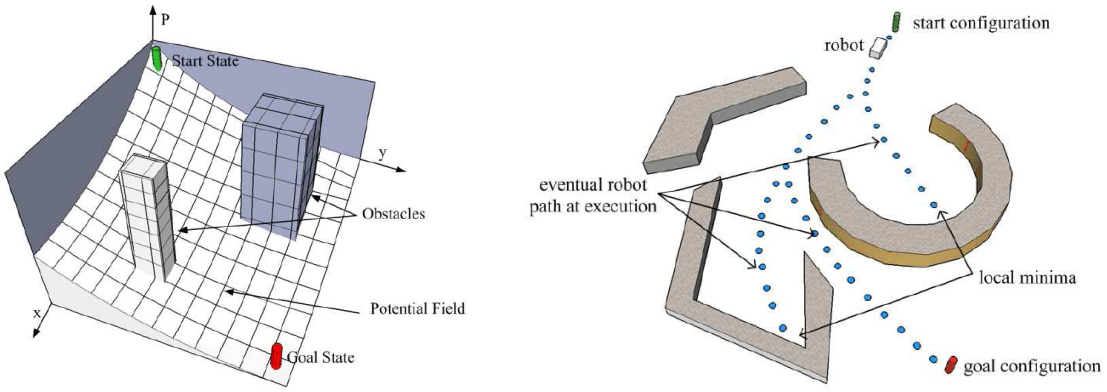
Sections 5.1 and 5.2 provide a review of the navigation task, while Section 5.3 reviews existing navigation drivers under the Player framework (see Section 3.3).

5.1 Local navigation

Local navigation based approaches are used for reactive collision avoidance. Since they only consider the immediate environment surrounding the robot, such methods generally have low computational complexity and are thus suitable for real-time operation in dynamic environments. This section provides an overview of several local navigation algorithms.

5.1.1 Potential fields

The potential fields navigation method, as described by Khatib [85], uses the philosophy of attraction and repulsion of a charged particle in a potential field. It models the robot as a charged particle while obstacles provide repulsive forces. Attractive



(a) A visualisation of potential fields.

(b) Obstacles forming troublesome 'local minima'.

Figure 5.1: Obstacles in the the potential fields method exert 'repulsive forces' on the 'positively charged' robot. Navigation is accomplished by moving towards a lower potential. Despite its elegance and simplicity, it suffers from local minima created by U-shaped obstacles (source: [88]).

features of potential fields are that they are conceptually simple, and straightforward to implement [86]. A resultant force from the field on the robot is then used to provide suitable navigation commands. A visualisation of the Potential Field navigation method is shown in Figure 5.1 (a).

Despite their elegance and simplity, potential fields suffer from several problems. As outlined by Koren and Borenstein [86], the main problems with potential fields are:

- It gets stuck in local minima, which arises if the robot runs into a dead-end created by a U-shaped obstacle (see Figure 5.1 (b)). Note that the potential function can be altered to ensure that the only minima is that of the destination [87]. However, this makes the algorithm much more complex, thus losing its main advantage.
- It struggles to pass through closely spaced obstacles. This is due to their large contribution towards the 'resultant force', and as a result they often make the robot reverse from such situations.
- It exhibits oscillatory behaviour, in the presence of obstacles and in narrow passages.

5.1.2 Vector field histogram

The vector field histogram (VFH) is a popular real-time motion planning algorithm, originally proposed by Borenstein and Koren [89] in 1991. Since then, it has had several performance enhancing revisions added to it and is one of the most popular local planners used in mobile robotics today.

The VFH method achieves real-time obstacle avoidance by firstly maintaining a 2-D histogram around the robot, where sensors such as ultrasonic or laser scanners

that give range-bearing measurements are typically used. Note that only one cell is updated for each new sensor reading[1], where each cell also contains a history of previous readings (and thus the use of histograms in VFH). A copy of the data from the 2-D histogram is then transformed into a 1-D polar histogram, where the magnitude of each sector represents a measure of obstacle density in that direction. A final stage of data reduction is then performed, by thresholding a smoothed version of the polar histogram. The resulting steering angle is found by finding an appropriate 'valley'[2][3] of the polar histogram with which the robot can be aligned.

One of the great benefits of VFH is that it is relatively robust to bad sensor measurements because readings are averaged out in the data reduction processes. Another key advantage with VFH is its low computational requirement. Unlike occupancy grid approaches [67], where a range reading is projected through the affected cells, VFH only updates one cell. However, this requires sensors that have higher bandwidths, and/or a reduction in the maximum allowable speed of the robot. The VFH method also requires the manual tuning of the threshold value (which decides whether a sector in the polar histogram is a peak or a valley). Although, according to the original authors Borenstein and Koren [89], this value only needs to be finely tuned when navigating in complex, cluttered environments.

Numerious extensions to the basic VFH algorithm have been developed. The extended VFH (VFH+) [90] considers the robot dynamics and uses a four-stage data reduction process, resulting in "smoother robot trajectories and greater reliablity" [90]. A further enhancement is VFH* [91]. It is designed to handle problematic situations such as instances where two equally sensible valleys are found, but one of them is later blocked by an obstacle. VFH* achieves this by using the A* search algorithm, where the immediate steering output is deferred, as it projects each potential trajectory of the robot ahead of time, and evaluates the consequences going down each path.

5.1.3 Dynamic window approach

Unlike VFH and other local navigation techniques that use distances to obstacles as their main input, the dynamic window approach (DWA) [92] operates directly in the velocity space. This is done to "correctly and elegantly incorporate the dynamics of the robot" [92]. DWA has two key steps: a) find the set of possible (translational and rotational) velocities that satisfy some criteria, and b) maximize an objective function.

In the first step of DWA, the method reduces the search space for possible velocities with the following three criteria:

- Circular trajectories: only circular trajectories described by pairs of transla-

[1]This is in contrast to sensor readings being projected through the 2-D grid, as in the case of occupancy grids [67].

[2]A valley in the thresholded polar histogram is a region with several consecutive sectors with a magnitude less than the threshold value.

[3]An appropriate valley is one that is in the right direction to reach the target destination and also one that is sufficiently wide for the robot to drive through (wider sectors represent larger gaps between obstacles).

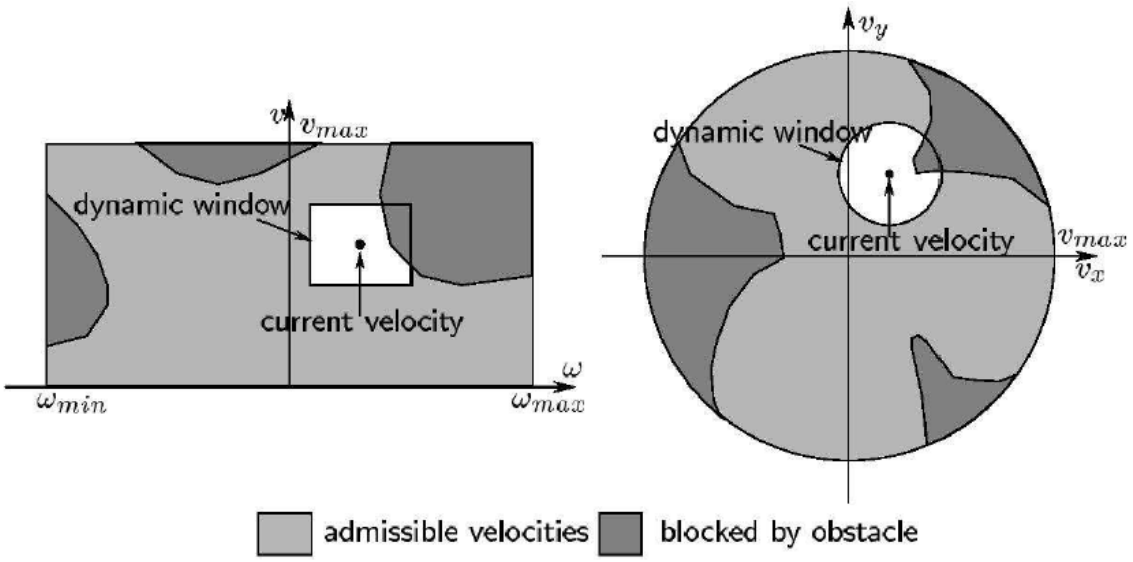


Figure 5.2: Search space of potential velocity pairs in the dynamic window approach, for a homolonomic (left) and non-homolonomic (right) drive system. The non-admissible velocities are shown in dark grey (source: [88]).

tional and rotational velocities are considered.

- Admissible velocities: only safe pairs of velocities are considered, i.e., velocities which allow the robot to stop before it reaches the closest obstacle on a trajectory.
- Dynamic window: restricts the velocity pairs that are possible to achieve within a short time interval, given the acceleration limits of the robot.

Once the search space for the velocity pairs has been reduced by the criteria (see Figure 5.2), the most suitable pair is selected by finding which one maximizes the objective function. The objective function is a weighted sum of three components: heading to the target, clearance between the closest obstacle, and forward velocity of the robot.

Although the original DWA paper uses a 'synchro-drive'[1] robot, it has also been adjusted to work on other configurations which travel in arc trajectories [93–95].

5.1.4 Nearness diagram

The nearness diagram (ND) [96, 97] is a navigation technique that consists of analysing a situation from two polar diagrams, which navigates based on gaps in the environment. These diagrams are constructed from both current and previous range-bearing sensor readings. The first diagram (ND from the central point, PND)

[1] A synchro-drive setup typically has three or four wheels but only two motors. One of the motors turns all wheels at the same speed, while the other motor adjusts the angle of all of the wheels. Its main advantages over a differential drive system are guaranteed straight line motion and the body of the robot does not rotate when it is turning.

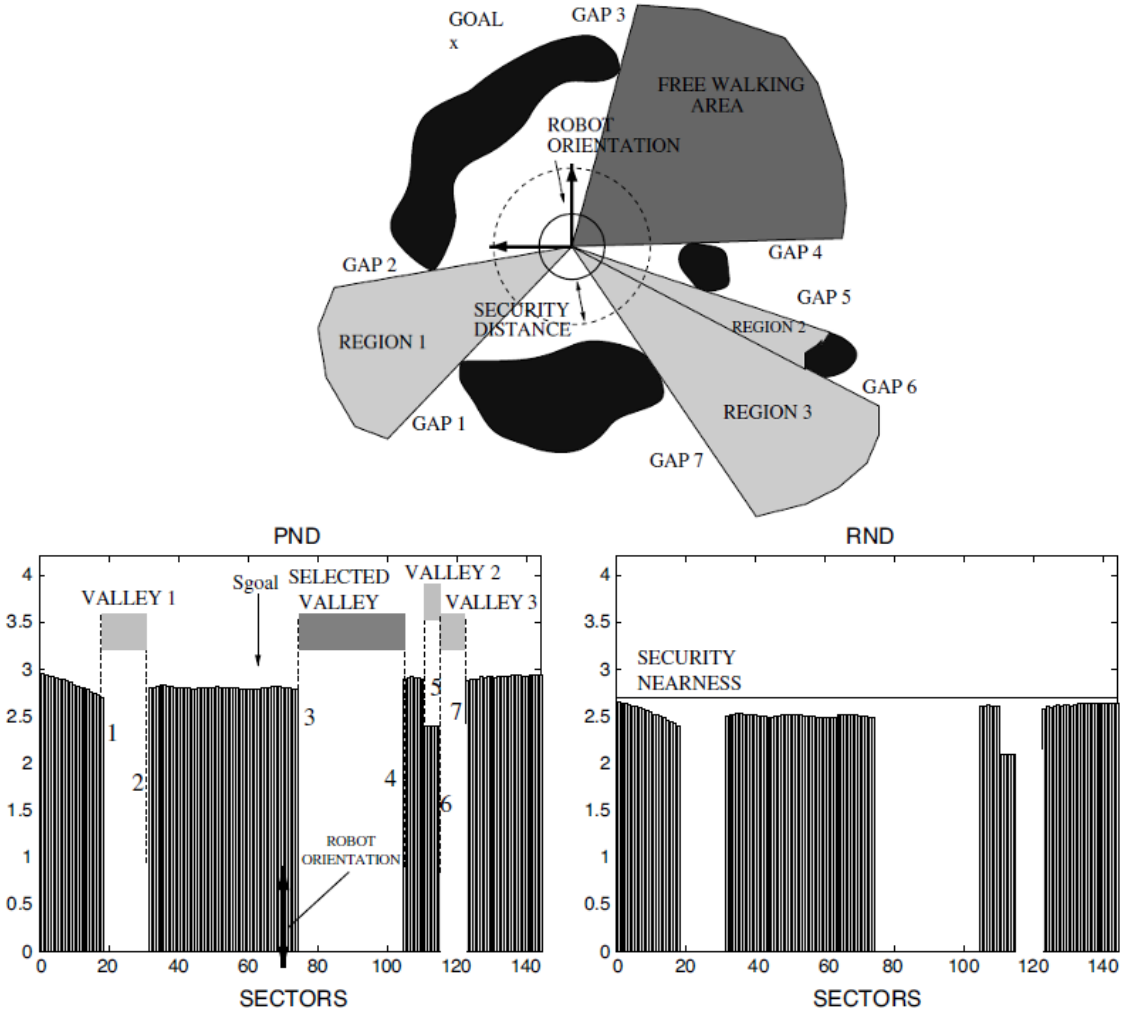


Figure 5.3: The nearness diagram method. A situation is firstly broken into regions (top), which is then used to find the PND (bottom left) and RND (bottom right). A situation is classified from information extracted from both the PND and RND (source: [96]).

is used to find 'discontinuities' and 'valleys' in the environment (similar idea with 'peaks and valleys' with VFH, see Subsection 5.1.2). The second diagram (ND from the robot, RND) is used to identify the safety level between the robot and the nearest obstacle. See Figure 5.3.

Information extracted from both the PND and RND is then used to classify the situation as one of five general situations[1], which "covers all the possiblities among the robot location, goal location, and obstacle configuration" [96]. Following this, appropriate translational and rotational velocity navigation commands are returned.

The ND+ (an extension of ND) method is the basis for another navigation technique called the Smooth Nearness-Diagram (SND) [98]. A key difference between ND+ and SND is that SND uses only a single motion law that applies to all naviga-

[1]For a description of each of these general situations, see [96].

tional situations. This results in smoother trajectories, as it removes abrupt transitions in behaviour when switching between the different motion laws in ND/ND+.

5.2 Global navigation

Unlike local navigation methods, global navigation techniques do not tend to get trapped in 'local minima' (i.e., convex shaped obstacles) and can yield optimum trajectories for the robot to reach the destination. Although they are computationally expensive to run, paths are typically computed either offline and do not need to be run often / in real-time. However, prior knowledge of an environment is required in global navigation and is often in the form of a map (such as one generated by SLAM, see Chapter 4). This section reviews prominent global navigation techniques (also known as path planning algorithms).

5.2.1 Path planning directly from grid maps

In metric path planning methods, it is common to manipulate the map slightly before running the path planning algorithm. The alteration that is performed is to grow the obstacles in the environment, by the radius of a circle that encompasses the robot (assuming it can turn in-place). This new map representation is often known as the configuration space (C-space) and it represents the legal positions of the robot in the environment. This also makes it easier for motion planning algorithms, as it reduces the robot to a single point. This subsection reviews A* and Wavefront path planning algorithms.

A* path planning

A popular algorithm used in path finding and graph traversal is the A* algorithm [99]. Using A* is also convenient in that it can operate directly on grid maps, which is a popular choice of map representation in SLAM (see Subsection 4.3.1). It has been successfully applied to many robotic applications that require finding an optimum path through free space between two points [95, 100].

A* is a best-first search algorithm, as it explores the most promising node when travelling towards the goal. Note that it is not a greedy algorithm, as it also takes the distance already travelled into account. To find the most promising node, the algorithm uses cost metrics and seeks to find the minimum cost to the goal. The total cost (F) is the sum of the cost from the start to the current node, plus an estimate (H) of the distance from the current node to the destination. A popular choice for H is the 'Manhattan distance', which is the sum of the horizontal and vertical distance components to the destination. The basic operation of A* (adapted from [101]) is

shown in Algorithm 1.

```
Input  : a 2-D bitmap (the C-space), with the start S, and the destination D
Output: an optimum trajectory from S to D

Empty the open and closed lists, and add S to the open list
while open list is not empty and current node is not equal to D do
    Current node C = node with lowest F in open list
    Add C to closed list
    foreach neighbour n of current node do
        Cost = G(C) + H(C, n)
        if n is in closed list then
            continue
        end
        if n is not in open list then
            Set current node as parent of n
            Add n to open list
        else if cost is less than G(n) then
            Set C as parent of n
        end
    end
end
The optimum path is found by using the references to the parents of C
```

Algorithm 1: The A* algorithm used to compute an optimum path to destination in a grid map.

Wavefront path planning

The Wavefront algorithm (also known as Distance Transform [102]) considers the C-space (generated from the grid map of the environment) to be like a heat conducting material. Obstacles have zero conductivity, while open space has an infinite conductivity. The Wavefront algorithm works by systematically expanding wavefronts that emanate from the goal, and thus it is a breadth-first search. The cost assigned to a node is larger than a neighbouring node that the wavefront reached first. The search stops when a wavefront has hit the current position, or there are no more nodes to explore (i.e., in this case there is no path). If a wavefront has hit the robot, the path to the goal is then found by exploring the node with a lower cost until the goal is reached [103]. Algorithmically, a basic Wavefront method is shown

in Algorithm 2.

Input : a 2-D bitmap (the C-space), with the start S, and the destination D
Output: an optimum trajectory T from S to D

For each node in the map, assign free space and S as 0, obstacles as 1, and D as 2
Empty the current node list, and add D
foreach current node C in the current node list **do**
    foreach neighbour n of the current node **do**
        if cost(n) is not 0 **then**
            continue
        end
        if n is the S **then**
            break
        end
        cost(n) = cost(C) + 1
        Add n to the end of the current node list
    end
    Remove C from the current node list
end

Empty trajectory T, and add S
if cost(S) is not 0 **then**
    Current node $C = S$
    while C is not D **do**
        C = minimum cost of the neighbouring nodes of C
        Add C to the end of T
    end
else
    No path found between S and D
end

Algorithm 2: The Wavefront path planning algorithm.

Wavefront methods also have the ability to handle different 'terrains'. For example, if a region in the map is unsafe to navigate for some reason (has a lot of obstacles, uneven surface, known to be congested, etc.), one can lower an area's conductivity. This will influence the path that the Wavefront will find. There are also other enhancements that can be made to the basic algorithm. In order to reduce processing time, waves can be propagated from both the start and goal (i.e., a dual Wavefront method), as they tend to cover less area compared to one that uses a single wavefront [104].

5.2.2 Path planning indirectly from grid maps

An alternative approach to path planning is to build topological maps on top of grid based ones. Topological maps, as described in Subsection 4.3.3, represent the environment as a graph. Path planning is then performed simply by traversing the graph created. This method has the following key advantages:

- There is a significant reduction in path planning computation. This is the case as sections of the environment (corridors, rooms, etc.), which require many cells in a grid map, are often represented by a couple of nodes in the graph.

- Topological maps can be 'simplified' (and thus the resulting optimal path) by graph pruning techniques.

- It permits the use of graph based search algorithms, including: A*/D* and their derivatives.

- Navigation using topological maps handle dynamic environments better, largely due to the fact that alternative paths can be generated quickly [100]. Note that this assumes that either the construction of a graph based representation of a grid map is fast or the map is natively stored as a topological map.

There are several publications detailing methods that extract topology-based maps from grid maps. For instance, Fabrizi and Saffiotti [105] use image-based morphology techniques to extract regions from a map, such as: dilation, erosion, opening, and closing. Another approach is to construct a Visibility Graph, such as the work done by Gao et al. [106]. A Visibility Graph works well for polygon-like obstacles, where a graph is constructed by connecting each 'visible' vertex of an obstacle with another. The optimum path is often found, as the algorithm's bias is to follow straight lines that are as close as possible to obstacles. Another popular approach is to construct a Voronoi Diagram[1] from the map [108]. Voronoi Diagrams are in contrast to Visibility Graphs, in that they find paths that maximise the distance between obstacles. Thus they provide a trade-off between finding the safest path, versus requiring long range distance sensors.

The publication by Thrun [108] details an algorithm to construct a topological map from a grid map (using a Voronoi Diagram) and is outlined as follows (see Figure 5.4):

1. Apply a threshold over each cell in the occupancy grid map.

2. Construct a Voronoi Diagram.

3. Locate the critical points: these are points that lie on the Voronoi Diagram and are local minima with regards to clearances between obstacles (and that with the neighbouring points).

4. Construct critical lines: these are simply lines extended from each critical point to their nearest obstacle in the thresholded map.

5. The topological graph is then constructed by assigning a node for each region separated by critical lines and obstacles, while the Voronoi Diagram specifies how the nodes are connected.

[1] A similar approach is the 'thinning' approach as described by Kwon and Song [107].

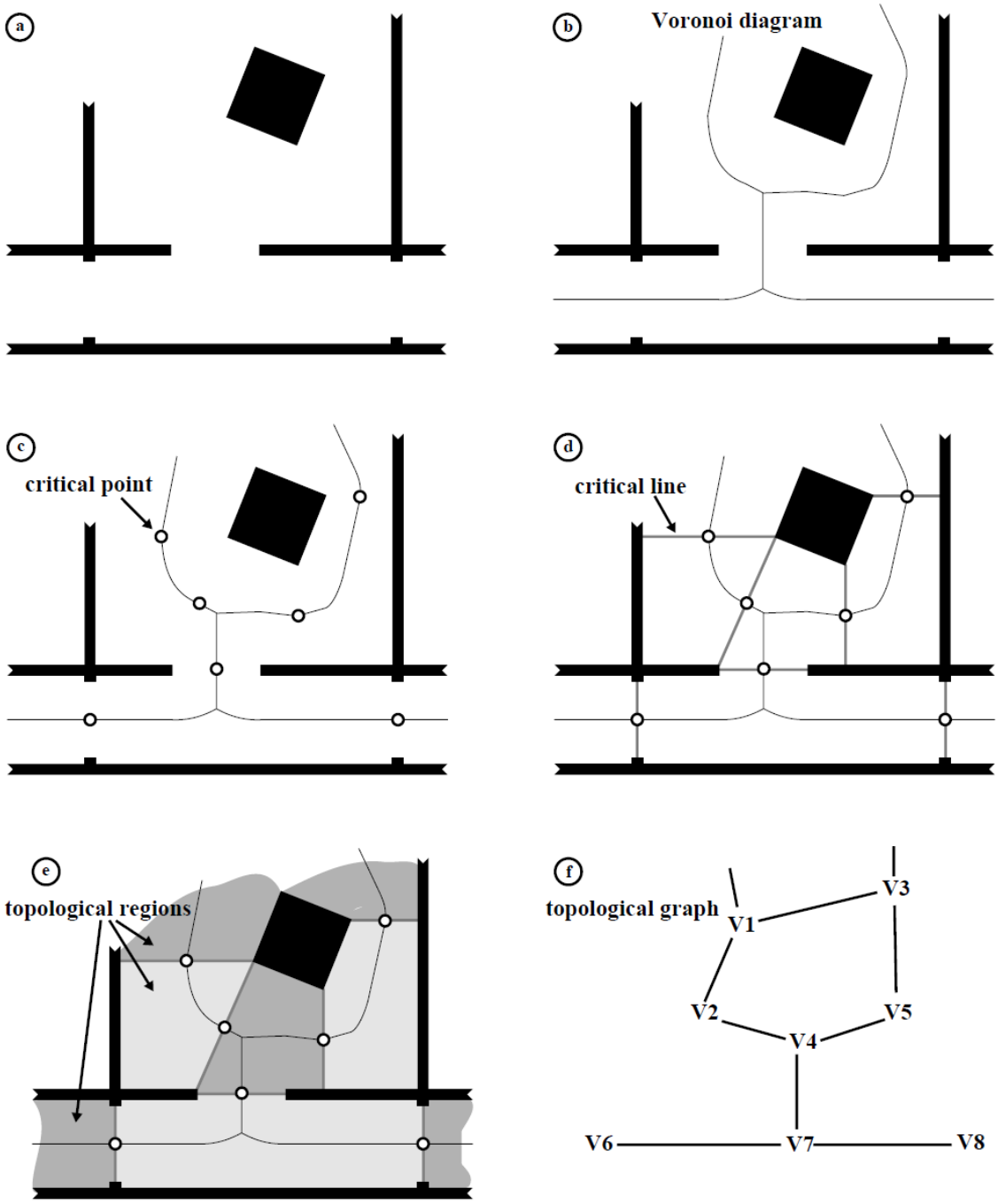


Figure 5.4: Constructing topological maps from occupancy grids (source: [108]).

5.2.3 Localisation

In order for a global navigation algorithm to be capable of directing a robot to travel to a user-specified destination, prior knowledge of the environment must be provided (such as a map of the area). Knowledge of the pose of the robot relative to the map is also just as important. If the map is being or just been created with SLAM, the pose of the robot in the map is already known. However, in many situations the pose may not be known well or at all, such in the following cases:

- To free up resources, the SLAM module might be disabled once the environment has been adequately mapped. However, this will rely on dead-reckoning alone to estimate the pose of the robot. It is well known that errors from this process will accumulate over time (sensor and measurement errors, wheel slipping, rounding/quantization errors, etc.), eventually rendering the information from dead-reckoned useless.
- An existing map (from a previous SLAM session, or from a floor plan drawing) is provided to the navigation system, without knowledge of where the wheelchair is in it.
- The navigation system for some reason happens to be disabled for some time while the user is manually navigating the wheelchair. If the pose was known before, when the navigation system is turned on again the wheelchair will have appeared to have tele-ported to a different position. This is often known as the kidnapped robot problem [109].

There are several well established methods in position tracking used to mitigate the effect of accumulation of errors, including: scan matching, Kalman filtering, and fusing readings from different sensors together. However, they make restrictive assumptions about the size of the error and shape of its uncertainty [110]. Such methods are unsuitable to global localisation tasks such as finding the robot in the map and the kidnapped robot problem. In the case of an unknown initial pose, since one cannot assume a bound on the pose error, a unimodal probablity distribution is inappropriate [111].

The papers [110–112] describe the Monte Carlo Localisation method (MCL) towards solving the global localisation problem. MCL makes use a particle filter (see Subsection 4.2.3), where potential robot poses are represented by particles. Initially, the robot pose is unknown and the particles are uniformly distributed over C-space. As new sensor information arrives, the weight of these particles are updated in accordance to the observations. Particles that support the observation receive a higher weighting, while particles that do not are often removed. The hypothesized robot pose is found from a set of weighted, randomly sampled particles. MCL methods are applicable to both local and global localisation problems.

A favourable characteristic of using a particle filter in MCL is that the update computational complexity is linear to the number of particles needed for the state estimation [111]. Thus it is beneficial to adaptively change the number of particles. When the pose is uncertain, the number of particles must be increased to provide more pose hypotheses. On the other hand, when the pose is well determined, the

number of particles may be reduced to save resources. The paper by Fox [111] describes the Adaptive Monte Carlo Localisation (AMCL) method, where the sample size is adjusted such that the error between the true posterior and the sample based approximation is less than a threshold[1]. Despite the substantial performance benefits with the AMCL method, it should be noted that KLD-sampling tends to increase the chance of premature convergence [113].

Despite MCL techniques being the most robust robot localisation algorithms, they are not without their limitations. In the case of simple MCL, if the initial pose is specified accurately but incorrectly, or if localisation temporarily fails, the estimated pose is unlikely to converge to the correct pose. This problem can be overcome by making some modifications to the localisation algorithm to be able to handle the kidnapped robot problem. As [110] outlines, some of the approaches include:

- Adding in particles sampled uniformly throughout the environment.
- Generating samples consistent with the most recent sensor observation.
- Assuming a higher level of sensor noise than there actually is.

Another issue with MCL localisation is that an underlying assumption in particle filters is that the environment is static (also known as the Markov assumption). This is done so that past and future data can be independent if the current state is known, allowing for recursive state estimation. Thus, many such localisation techniques fail in highly dynamic environments, as moving objects tend to create large discrepancies between the map and the observation. Unfortunately, there is no easy way to overcome this, apart from using sensor data that is not affected by the moving objects. An ingenious approach is to use information obtained from a camera pointed towards the ceiling of an environment for global localisation [110].

5.3 Navigation in Player

Fortunately, Player (see Section 3.3) comes with some useful and relatively stable navigation related drivers. A brief description and research into such drivers is provided below. Note that the evaluation of some of these drivers on both simulated and physical robots is provided in Chapter 8.

`amcl` — implements a basic adaptive Monte Carlo localisation algorithm (see Subsection 5.2.3). The driver requires the following Player interfaces: `laser`, `position2d` (for odometry information), and the `map`. Essentially, localisation is performed by matching the current laser scan with what it would appear to be at each potential pose in C-space. Potential poses are stored as particles, which are initially randomly spread uniformly throughout the map. By making use of both odometry and previous scan data, incompatible particles will be removed. This will leave a cluster of particles that represent the

[1]The difference between the true posterior and the sample based approximation is found by the Kullback-Leibler distance (KLD-sampling).

most probable location of the robot within the map (which can be accessed through the `position2d` interface `amcl` provides). Note that it is mentioned in the driver's documentation that the initial pose of the robot within the map must be known to some extent, otherwise `amcl` will converge to the incorrect location.

vfh — the vector field histogram histgram driver provides real-time obstacle avoidance. It implements the VFH+ method (see Subsection 5.1.2). It requires data from either a `laser` or `sonar` interface. The driver provides a `position2d` interface, which is used to give position or velocity control commands. After performing obstacle avoidance, `vfh` will then issue commands to the underlying `position2d` robot interface.

nd — implements the nearness diagram navigation algorithm, as described in Subsection 5.1.4. `nd` is an alternative to the `vfh` driver, which is also used for local path planning and obstacle avoidance.

snd — smoothness nearness diagram navigation driver. It is an improvement over the `nd` driver, in that it removes oscillatory patterns in the resulting robot trajectory.

wavefront — used in conjunction with a local path planner (such as `vfh`, `nd`, or `snd`), to provide global navigation. Subsection 5.2.1 describes the basic algorithm that is implemented in `wavefront`. Provided that the robot has been localised, the driver accepts a goal throught the `planner` interface. For `wavefront` to successfully find a path to the goal, there has to be a suitable route to it within the map that is on the `map` interface. If a path is found, and if enabled, the `wavefront` driver will then issue velocity commands to a local path planning `position2d` interface. `wavefront` also has the option of using a `laser` interface, which is used to improve the operational safety when issuing velocity commands to the local path planner driver.

Chapter 6

Hardware

As discussed in Chapter 3, simulating a robot in a virtual environment offers many benefits. However, it is difficult to closely simulate a real-world environment. Results from simulation only provide speculation to how well the software will actually perform on an actual wheelchair.

Prior to testing the navigation system (described in Chapter 7) on an instrumented wheelchair (see Section 6.2), the software was first validated on a Pioneer 2-DX virtual robot in the Stage simulation environment (see Subsection 3.3.5). As an intermediate step between simulation and the actual wheelchair, the navigation software was tested on a small wheeled robot. This was useful as:

- At the time, the wheelchair needed to be instrumented (while the small robot was mostly working).
- Performing full-scale tests is time consuming.
- The small robot can be safely operated in an office environment, whereas the wheelchair needs a large and dedicated test area.

This chapter discusses the hardware setup of the physical robots used to validate the navigation system. Section 6.1 describes the small robot's hardware briefly, while Section 6.2 describes the robotic wheelchair. Note that both of these robots have common attributes: they are both differential drive, use microcontrollers to control low level hardware, communicate to the navigation system through a UART interface, and share the same Player plugin and communication protocol (described in Subsection 7.2.3).

6.1 Small robot

The small robot, depicted in Figure 6.1, was developed in my spare time as a hobby project throughout my undergraduate engineering course. Since it has similar dynamics to a wheelchair, it was used as a intermediate step in evaluating the navigation system between simulation and an electric wheelchair. This section introduces the hardware of the small robot.

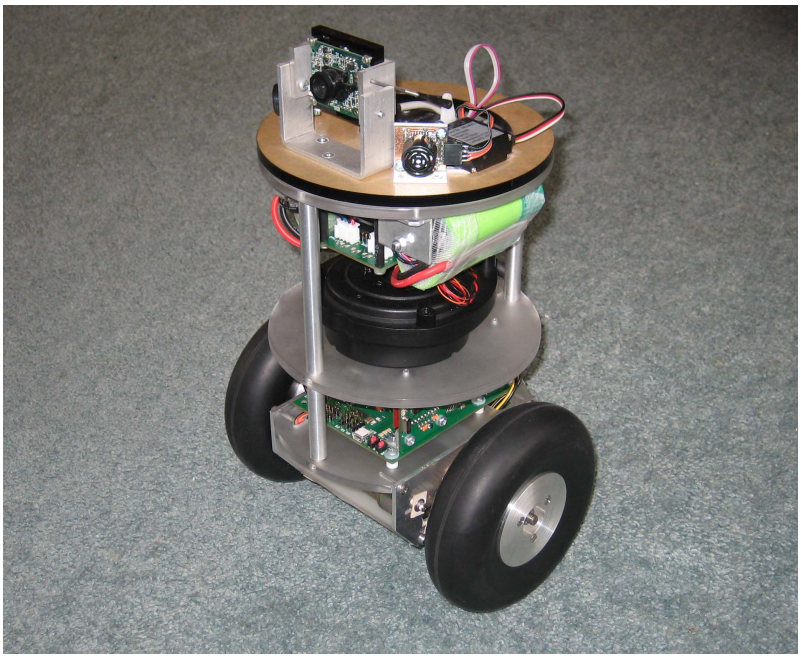


Figure 6.1: The small robot was used as intermediate step between a simulated robot and the instrumented wheelchair while evaluating the navigation software.

The robot consists of several microcontrollers interfacing to sensors, actuators, and other low level hardware. It has the following sensors: wheel encoders, a six degree of freedom (DOF) inertial measurement unit (IMU), a web-camera, a XV-11 laser scanner (see Subsection 7.2.4), and two sonar rangefinders. An I2C bus is used for data communication between many of these components. Figure 6.2 provides an overview of the main hardware modules.

A Beagleboard[1] is installed on the robot, and is running a customised image of the Angstrom[2] distribution. Instead of running Player and the navigation software on the Beagleboard, the data was streamed back to the development computer (that ran Player and the navigation software). A USB wireless dongle attached to the Beagleboard was configured as an access point to which the development computer was connected. The video stream and serial data was transferred wirelessly, as described in subsections 7.2.1 and 7.2.3 respectively.

6.2 Electric wheelchair

An Invacare Pronto M51 centre drive wheelchair[3] was used for evaluation of the navigation system presented in this thesis. Although front wheel drive wheelchairs are known to more manoeuvrable than other types [114], a centre drive configuration was chosen as it has similar dynamics as the small robot (see Section 6.1) and the Pioneer 2-DX robot used in the Stage simulation environment (see Subsection 3.3.5). Support for front and rear wheel drive wheelchairs could be added, which would

[1] `http://beagleboard.org/`
[2] `http://www.angstrom-distribution.org/`
[3] `http://invacare.co.nz/index.cfm/1,134,311,50,html`

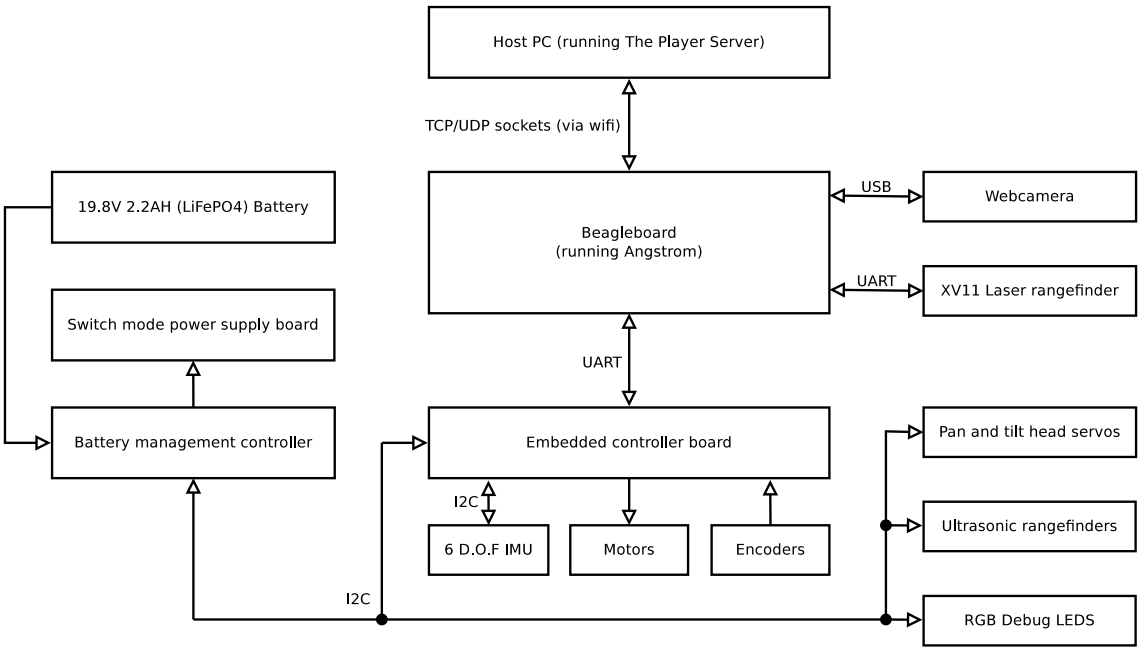


Figure 6.2: A high level block diagram depicting the small robot's hardware.

require making modifications to the navigation software (in particular to vehicle motion models and the dead reckoning equations).

This section illustrates the modifications made to the Pronto M51. Note that the interface to the wheelchair's motors and joystick that this navigation system uses is through the DX2 system[1], discussed in Subsection 6.2.1.

6.2.1 DX2 system

The DX2 system is Dynamic Control's current line of controllers for electric wheelchairs. Fundamentally, it consists of a power module and a master remote, shown in Figure 6.3. Although it may have been simpler to interface to a standard brushed motor controller and joystick, the DX2 provides several features useful to this project. This subsection outlines these features and the interface to the DX2 system.

A key task of the power module (DX2-PMA) is to regulate the speed of the wheelchair's left and right brushed DC motors. The key features of its motor control are: speed and yaw rate control (with smoothed profiles), current limiting, veer compensation, load compensation (automatically adjust the speed when driving over curbs or up slopes), and roll-back prevention on inclined planes. Closed loop control is achieved by using feedback from the motor's back EMF. Since the DX2 system is a Class II medical device, the DX2-PMA has an extensive set of protection and fail safe mechanisms. These include protection against: external short circuits, stalled motors, reverse battery, battery under voltage protection[2], and detection of

[1] `http://www.dynamiccontrols.com/dealers/products/dx2`

[2] This is primarily concerned with throttling the drive motors, and does not protect against over discharge via quiescent power drawn by other modules.

Figure 6.3: The fundamental components of the DX2 system: a master remote and the power module, and an optional actuator module (source: Dynamic Controls).

system faults.

Another main component of the DX2 system is the master remote (DX2-REM55X). The DX2-REM55X is the user's interface to the wheelchair: the joystick allows the user to set the speed and yaw rate of the wheelchair, while the onboard LCD communicates the system status to the user (battery level, settings, fault codes, etc.).

The DX2 modules communicate with each other via Dynamic Control's DX BUS (which uses the CAN serial bus). Due to the stringent safety requirements that the wheelchairs need to comply to, the DX BUS protocol is complicated, and interfacing directly to the DX2 network would be difficult. Instead, a modified General Purpose SLIO[1] Board (GPSB) was used to provide a simple interface to the DX2 system. Figure 6.5 depicts the integration of the GPSB device in the instrumented wheelchair's hardware.

The GPSB device is usually embedded in auxiliary DX2 modules, and not retailed by itself. For the purposes of this project, Dynamic Controls kindly provided a modified GPSB device. When the DX2 system detects the presence of the modified GPSB device, the master remote's joystick no longer has control of the wheelchair. Instead, the master remote relays the joystick position to the GPSB, which in turn makes this data available to an external controller (i.e., the embedded controller, see Subsection 6.2.2). As a safety measure, the GPSB device needs be active at startup and it needs to receive a data packet from the external controller at least once a second, otherwise control is given back to the master remote. The physical interface to the GPSB is TTL serial.

[1]Serial linked I/O.

(a) The electric wheelchair. (b) Front view. (c) Side view.

Figure 6.4: The test bed wheelchair, a Pronto M51, has been retro-fitted with a laptop, an ipad, wheel encoders, a Kinect sensor, and URG-04LX and XV-11 laser scanners. Note the emergency stop button on the left arm rest.

6.2.2 Instrumentation

The instrumented Pronto M51 (see Figure 6.4) was used as a final stage validation of the navigation system. This subsection describes the modifications made to the wheelchair and the sensors and electronics added to it. A high level block diagram of the modified wheelchair's hardware is shown in Figure 6.5.

In many robotic applications that rely on localisation in an environment, dead reckoning is required. For wheeled robots, this can be accomplished using wheel encoders as inputs. The Pronto M51 was instrumented with two industrial-grade 1024 counts per revolution (CPR) quadrature optical encoders. Since these were installed directly on each wheel shaft, with a nominal wheel diameter of 0.250 m and using $\times 4$ decoding, the resolution is 0.192 mm per edge trigger. See Subsection 7.1.2 for the dead reckoning equations.

The wheelchair was also fitted with three different types of laser scanners. Although the navigation system only requires one laser rangefinder, comparisons of their suitability and performance are later made in Chapter 8. One of the lasers used is the popular Hokuyo URG-04LX[1] (costing around $1,300 USD). Another laser scanner that was evaluated is the XV-11 (see Subsection 7.2.4). Both the URG-04LX and the XV-11 were mounted on the wheelchair's foot rest, as shown in Figure 6.6 (a). The third laser scanner was derived from the depth map given by the Kinect[2] (this process is described in Subsection 7.2.5). The Kinect is mounted

[1] `http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html`

[2] The Kinect is a sensor marketed by Microsoft for use with their Xbox 360 gaming console.

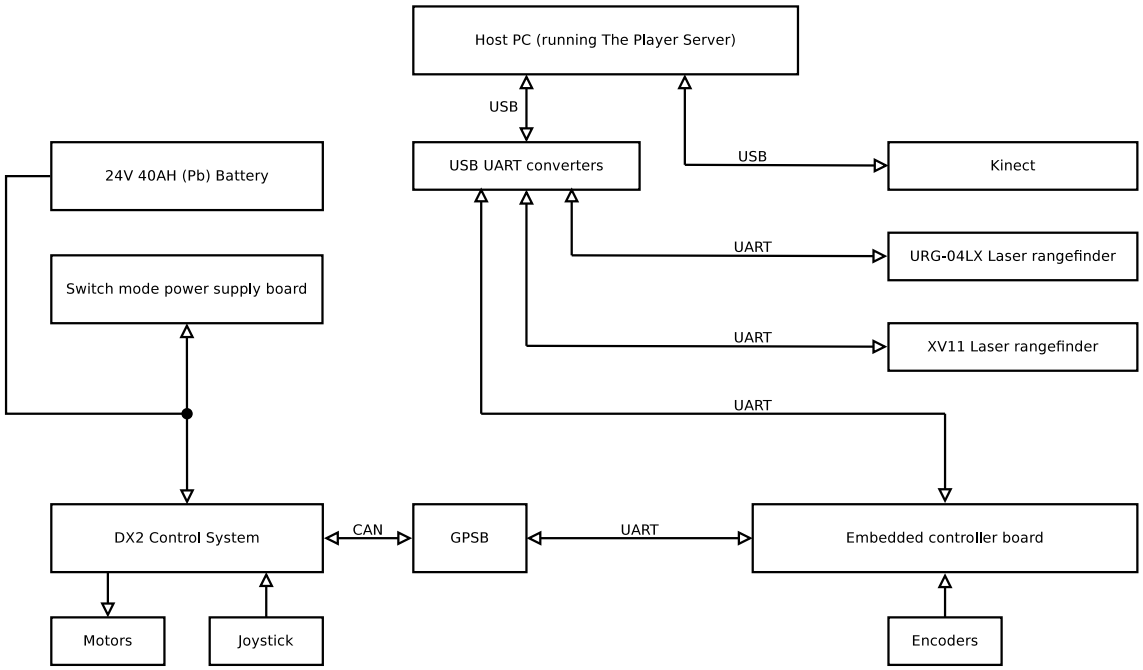


Figure 6.5: A high level block diagram depicting the instrumented electric wheelchair's hardware.

above the user's head to a pole attached to the wheelchair base, and the Kinect's mount is shown in Figure 6.6 (b).

A modern laptop (HP 6730b with a T9300 2.5 GHz Core 2 processor and 4 GB of DDR2-800 RAM) was also installed on the wheelchair, to run the navigation algorithms. This was firmly attached to a foldable perspex table using Velcro, as shown in Figure 6.7. All sensors and low level hardware devices (i.e., the embedded controller unit, shown in Figure 6.8) were interfaced to the navigation software through the laptop's USB ports. The user interface to the navigation system, with the exception of the wheelchair's joystick, is provided through an ipad via a remote desktop connection to the laptop.

The last addition to the electronic hardware added to the Pronto M51 is the embedded controller unit, which is shown in Figure 6.8. This device handles the interface to the wheel encoders, provides dead reckoning, and interfaces to the GPSB module. It consists of the following:

- An Arduino Uno I/O board with a ATmega328 microcontroller running at 16 MHz, powered off the laptop's USB supply. Note that the software running on the Arduino was developed in C and used the `avr-gcc` toolchain, as opposed to Arduino's native Wiring language. The embedded controller software is discussed in Section 7.1.

- A modified GPSB device (see Subsection 6.2.1).

- +12 V and +5 V switch mode power supplies (SMPS), for providing power to the Kinect and URG-04LX sensors respectively. The +5 V SMPS also powers the XV-11's motor, through a PWM controlled MOSFET.

(a) The URG-04LX and XV-11 mounting.

(b) The Kinect mount.

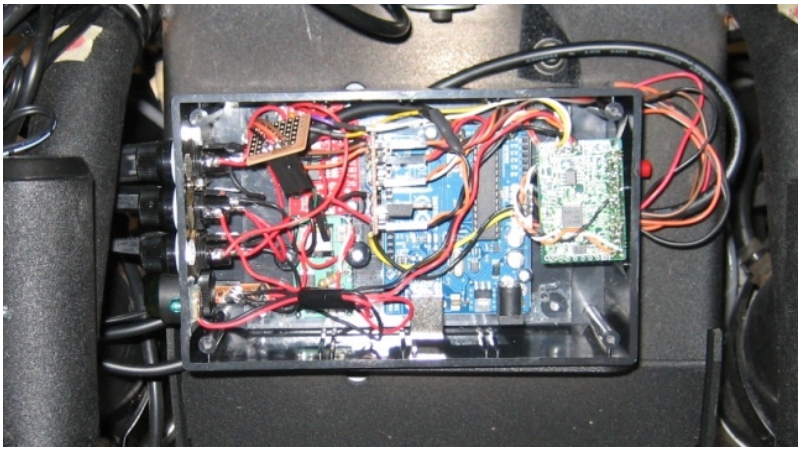
Figure 6.6: Mounting of the laser scanners and Kinect on the instrumented wheelchair.

(a) Folded in.

(b) Folded out.

Figure 6.7: Mounting of the laptop on the instrumented wheelchair.

(a) The embedded controller unit.



(b) The internals of the embedded controller.

Figure 6.8: The embedded controller of the electric wheelchair. A description of the duties it performs is provided in Subsection 7.1.

- Inline fuses for the input (+24 V, from the DX BUS) and SMPS outputs.
- A switch that disconnects power from the entire embedded controller unit.

Chapter 7

Software

A vital part of any complicated electromechanical system is its software. It is becoming increasingly more common to implement a device's intelligence in software (as opposed to in clever physical mechanisms or in electronic filtering circuits), mainly due to the flexibility that software provides. Furthermore, with the availability of cheap computational power, it is now practical to implement algorithms and filters in software. This is certainly the case with the navigation system presented in this thesis. This chapter is devoted to describing, in a high-level manner, the navigation software system. Figure 7.3 provides a block diagram of the Player server setup and is elaborated in Section 7.3, whereas Figure 7.4 shows how the Player client and user interface programs interfaces to the server. The navigation system involved development of several different layers of software:

Embedded software — the lowest level of software, responsible for interfacing to wheel encoders and the DX2 system (via the GPSB module). This level required stringent reliability, latency and deterministic requirements, making it only practical to implement on a real-time embedded system. This program, described in Section 7.1, was developed in C (using the `avr-gcc` toolchain), and engineered to be efficiently executed on an ATmega microcontroller.

Custom Player plugins — although Player (see Section 3.3) comes with a variety of drivers, several had to be developed for interfacing Player to customised hardware and external software packages. Section 7.2 describes these plugins, developed in a mixture of C and C++.

Player client — the program that interfaces to the Player server. As described in Section 7.4, it is responsible for implementing tasks particular to wheelchair navigation, as opposed to interfacing to hardware and general robotic algorithms (that are usually implemented in Player drivers/plugins). The client program was developed in C/C++.

User interface — another level of abstraction was developed between the client program and the user interface. This was done to allow the use of Python, as it is well suited to front-end programs that require a graphical user interface (GUI). Although Player comes with Python bindings, a design decision was made that it was best to separate the client and user interface applications.

This was achieved by compiling the client program as a shared library and using Python's `ctypes` package to interface to it. The user interface is discussed in Section 7.5.

7.1 Embedded controller software

This section describes the software running on the embedded controller (an ATmega328 running at 16 MHz) shown in Figure 6.5. Note that the embedded software running on the small robot outlined in Section 6.1 is similar. Subsection 7.1.1 describes event tasks, while Subsection 7.1.2 describes the polled tasks.

7.1.1 Event tasks

An important task carried out by the embedded controller is to provide dead reckoning, using encoders mounted to the wheelchair's wheel shafts. In order to reliably perform dead reckoning, it is imperative that the controller captures all encoder pulses. Thus the interface to the encoders was interrupt driven. Both channels of the 1024 counts per revolution (CPR) quadrature encoders were connected to the microcontroller's pins to generate separate any-edge interrupts. This setup gives the maximum attainable resolution from the encoders, i.e., x4 decoding[1]. The number of encoder counts was maintained by either incrementing or decrementing a signed variable, based on the phase difference between the channels during the interrupt.

The serial driver was also implemented using interrupts. When a receive character interrupt occurred, the received character was added to a buffer. The transmission of characters was interrupt driven: polled tasks added characters to another buffer, which was emptied after a previous character had been transmitted. Since the ATmega328 has only one hardware serial port, a second serial port for the GPSB was added using software. This was efficiently implemented using a free running timer for timing and the input capture pin for receiving characters.

7.1.2 Polled tasks

The microcontroller is also responsible for carrying out other tasks, including: updating the dead reckoning system, sending information back to the navigation system (odometry estimate and joystick information), and setting the speed of the wheelchair's motors. Although these tasks require determinism and fail-safe operation (and thus are not suitable to implement on the laptop), unlike the event driven tasks they do not require low latency[2]. In the main loop, the polled update functions (50 Hz, 10 Hz and 5 Hz) were provided by continually checking a flag set by a free-running timer. This subsection describes the main aspects of the polled tasks.

[1] http://www.usdigital.com/support/glossary

[2] The Nyquist frequency for a 4096 interrupts per revolution encoder mounted on a 0.25 m diameter wheel travelling at 2 m/s is about 20 kHz.

50 Hz tasks

The dead reckoning module approximates a curved trajectory with a series of straight line segments. In each update, the difference in encoder counts for the left (T_L) and right (T_R) wheels are found. Thus the distance travelled by the left and right wheels are $d_L = (C_L T_L)/T$ and $d_R = (C_R T_R)/T$, where T is the total encoder counts for a full wheel revolution and C_L and C_R are the left and right wheel circumferences. Dead reckoning for a differential drive robot is computed using the following (simplified) equations:

$$\Delta\theta = \frac{d_R - d_L}{D} \ , \tag{7.1}$$

$$s_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{t-1} + \begin{bmatrix} 0.5(d_R + d_L)\cos(\theta_{t-1} + \Delta\theta) \\ 0.5(d_R + d_L)\sin(\theta_{t-1} + \Delta\theta) \\ \Delta\theta \end{bmatrix} \ , \tag{7.2}$$

where s is the robot pose (described by Cartesian coordinates, x and y, and θ to denote the heading) at time index t, D is the separation between the wheels.

In order to interface Player with the custom robots developed (i.e., the small robot and the electric wheelchair), a flexible and extensible serial-based protocol was developed. This is discussed in Subsection 7.2.3. Due to the scarce amount of RAM on the microcontroller (and thus limited space for message buffers), messages had to be processed at a reasonable rate. The custom protocol interface was updated at 50 Hz, which includes both the processing of received packets and transmission of messages.

10 Hz tasks

As introduced in Section 6.2.1, the navigation system makes use of the DX2 system. The DX2 uses the controller area network (CAN) bus as the hardware interface to transmit messages between its modules. However, due to the stringent safety requirements that the wheelchairs need to comply to, the DX2 protocol is complex and interfacing directly to the DX2 network is difficult. Fortunately, a modified GPSB was developed to allow a simple serial interface to the DX2 system.

With simplicity in mind, the GPSB uses a binary protocol which has only one receive and one transmit packet format. The receive packet (GPSB to the microcontroller) consists of a start byte, the joystick co-ordinates, a configuration byte, and a checksum byte. The transmit packet (microcontroller to GPSB) is of a similar format but uses speed and yaw rate in place of the joystick values. Note that a packet is only sent to the GPSB if the microcontroller has since received at least one packet. This ensures that the GPSB has been initialised before the navigation system issues commands to it. Also note that once initialised, as a safety feature, the GPSB will shut down the DX2 system if it does not receive a packet at least every second.

5 Hz tasks

Since the DX2 system relies on 'human-in-the-loop' control, speed and turn rate commands to the system result in the chair travelling at relative speeds and turn rates. In order to achieve velocity control, a PID controller was developed:

$$u_{v,t} = u_{v,t-1} + K_{p,v}e_{v,t} + K_{I,v}\int_0^t e_{v,\tau}d\tau + K_{D,v}\dot{e}_{v,t} \ , \tag{7.3}$$

where the error is $e_{v,t} = r_{v,t} - v_t$, τ is the update interval (200 ms) and the gains ($K_{p,v}$, $K_{I,v}$ and $K_{D,v}$) were found manually. The current velocity, v_t, was derived from the filtered average of T_L and T_R. A PID controller for regulating the yaw rate $\dot{\theta}$ was also implemented in a similar fashion:

$$u_{\dot{\theta},t} = u_{\dot{\theta},t-1} + K_{p,\dot{\theta}}e_{\dot{\theta},t} + K_{I,\dot{\theta}}\int_0^t e_{\dot{\theta},\tau}d\tau + K_{D,\dot{\theta}}\dot{e}_{\dot{\theta},t} \ , \tag{7.4}$$

where $e_{\dot{\theta},t} = r_{\dot{\theta},t} - \dot{\theta}_t$, $\dot{\theta}_t$ is derived from a smoothed $\Delta\theta$ value, and the gains $K_{p,\dot{\theta}}$, $K_{I,\dot{\theta}}$, and $K_{D,\dot{\theta}}$ were found manually. Note that equations (7.3) and (7.4) would suffice if direct control of the motors was used. However, to provide a more comfortable ride for the user, the DX2 system heavily smooths the velocity and yaw rate. If a basic PID controller was used, this would result in either instability (due to high gains and phase lag) or poor transient response (low PID gains). Instead, the velocity and yaw rate controllers were modified slightly:

$$u'_{v,t} = \begin{cases} f_v(r_{v,t}) \ , & \text{if } r_{v,t} \neq r_{v,t-1} \text{ or } \dot{e}_{v,t} > \lambda_v \\ u_{v,t} \ , & \text{otherwise} \end{cases} \ , \tag{7.5}$$

$$u'_{\dot{\theta},t} = \begin{cases} f_{\dot{\theta}}(r_{\dot{\theta},t}) \ , & \text{if } r_{\dot{\theta},t} \neq r_{\dot{\theta},t-1} \text{ or } \dot{e}_{\dot{\theta},t} > \lambda_{\dot{\theta}} \\ u_{\dot{\theta},t} \ , & \text{otherwise} \end{cases} \ , \tag{7.6}$$

where $f_v(\cdot)$, $f_{\dot{\theta}}(\cdot)$ are linear heuristic functions found from experimentation and λ_v, $\lambda_{\dot{\theta}}$ are constants that decide when the transient period has passed. In other words, the embedded software's PID controllers only become active after the transient velocity / yaw rate period has passed, which initially is handled by heuristics.

7.2 Custom Player plugins

Throughout development, several custom Player plugins were created. Although it may have been easier to implement some of these plugins with the client program, every attempt was made to adhere to the philosophy of Player (and any project that uses the reusable software paradigm). That is, the drivers (and plugins) are generalised interfaces to hardware or algorithms, and the client program is project-specific that builds functionality upon the drivers. Moreover, to allow the possibility of migrating to a different robotic framework, the custom drivers were compiled as a library and later made into a plugin. This also allowed the plugin under development to be tested in a more friendly environment (e.g., in a client program). The general

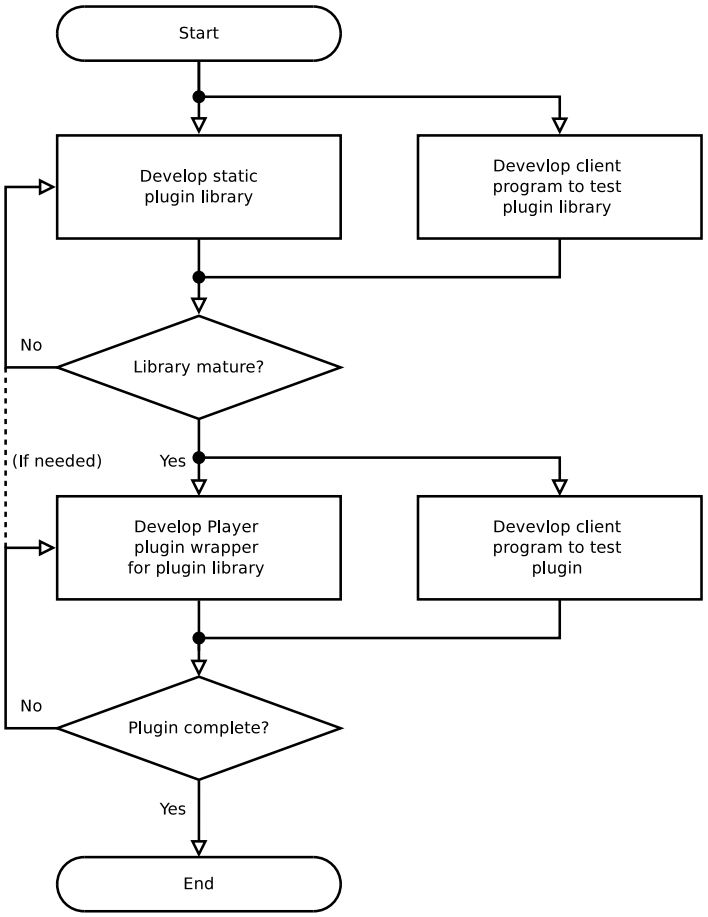


Figure 7.1: The general process I used in creating custom Player plugins.

process that was used when creating a plugin is shown in Figure 7.1. This subsection describes the custom plugins that were developed.

7.2.1 Wireless video client

Although the navigation system described in this thesis does not use computer vision to sense the environment (with the exception of the work described in Subsection 7.2.5), it was deemed useful to stream the small robot's onboard camera feed back to the development computer. A plugin was developed to interface to the video server that was running on the small robot (refer to Section 6.1). The robot had a Beagleboard on it that ran Angstrom (a Linux distribution). The video server, `spcaserv`, was compiled to run on the Beagleboard from open-source software and streamed images from the onboard USB camera to a client via sockets. The frames were transmitted as jpeg images and then decompressed into OpenCV's native IplImage structure. Player's `CameraProxy` data format is similar to an IplImage, thus to publish the frame to the Player server required copying the relevant data into the `player_camera_data_t` structure.

7.2.2 Camera undistortion

To account for lens distortion of the images captured from the video camera, a camera undistortion plugin was created. Since the computer vision processing library OpenCV was used, this was simply a matter of porting the provided OpenCV sample camera undistortion code to a Player plugin. Thus this plugin required the raw image from Player's `camera` interface and outputted the undistorted frame in another `camera` interface.

7.2.3 Robot interface

A light-weight general purpose bi-directional serial-based communication protocol was developed to interface Player to custom robots. Both the small robot (Section 6.1) and the electric wheelchair (Section 6.2) used this protocol, despite being instrumented with different sensors.

The protocol was loosely based on NMEA 0183 strings (the De facto standard in GPS modules) in that it uses: a UART physical interface, ASCII characters (although not as efficient as binary data, it is easier to debug), has a start character, fields are separated by commas, has a checksum, and is terminated by a new line character. Messages are added to a buffer that is later emptied by a mechanism depending on whether the software is running on the laptop or the embedded controller. Modularity and extensibility is achieved by separating the core protocol functions from implementation specific ones and also by using callback functions.

In order to accommodate both the small robot and the wheelchair, the plugin was designed to require a `opaque` interface. In the case of the wheelchair where a USB to serial port adapter is used to connect to the embedded controller, the `serialstream` driver was used to provide the `opaque` interface. For the small robot, the `tcpstream` driver was used instead, which accessed the robot's serial port wirelessly through the `ser2net` network proxy, again compiled to run on the router from open-source software.

Based on the parameters in the Player configuration file, the robot interface plugin provides different interfaces. On the small robot `power` (used for monitoring battery status), `sonar` and `imu` interfaces are created, whereas just the `joystick` interface is created when using the wheelchair. The plugin always provides a `position2d` interface, which allows control of the robot's motors and querying of its odometry. The plugin also requires dimensions of the robot, which are passed onto the `position2d` interface it creates. The robot's physical attributes are used by obstacle avoidance and path planning drivers.

7.2.4 XV-11 laser rangefinder

An innovative robotic vacuum cleaner made by Neato Robotics[1] became available for purchase in June 2010. This vacuum cleaner uses a laser range-bearing sensor and performs SLAM to efficiently to cover the entire floor surface with a single pass. The

[1] `http://www.neatorobotics.com/`

publication by Konolige et al. [115] describes the vacuum cleaner's pre-production laser sensor that according to the authors, cost less than $30 USD to produce [115]. Unfortunately, the individual laser rangefinder has yet to be sold by itself. For the purpose of evaluating this sensor and its applicability to this project, one of these vacuum cleaners was purchased.

The vacuum cleaner's laser rangefinder (referred here as the XV-11) achieves distance measurements via triangulation. The XV-11 uses a low power, eye-safe laser to project a dot onto the environment. The environment objects reflect this light back which is captured by an imaging device on the XV-11. Using the centroid of the reflected dot and the known geometry between the imager and the laser, the distance can be found. The laser and imager are made to rotate, thus allowing a scan to be made.

No official documents on interfacing to the XV-11 have been released. Fortunately, there has been sufficient interest within the research and hobby communities, and the interface has been reverse-engineered. A plugin was developed for interfacing the XV-11 to Player, using information found from the internet[1]. The XV-11 provides a full 360 ° range of measurements, with an angular resolution of 1 ° at a scan rate of 5 Hz. The device is also reported to be capable of measuring distances between 0.2 m and 6 m, with less than 0.03 m resolution.

7.2.5 Kinect to laser scanner

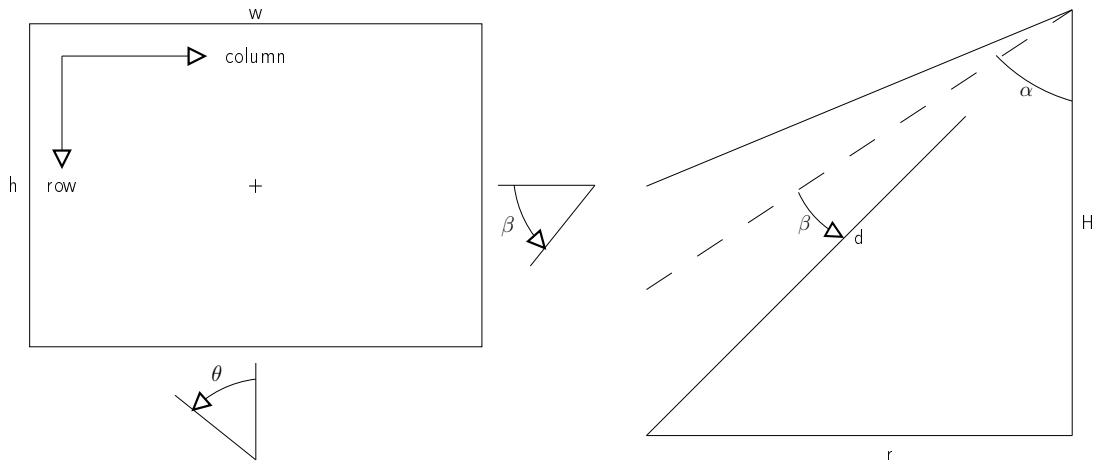
Originally released to the public in late November 2010, the Kinect sensor provides a colour and a depth (RGB-D) camera for the Xbox 360 gaming console. It also has an array of microphones, an accelerometer and a motor to tilt the device. The sensors communicate through USB and appear as different end points. The Kinect has generated significant interest within research and hobby groups, particularly due to its low cost (around $150 USD) and availability. It provides a viable alternative to stereo vision systems, which can cost over $2,000 USD. Despite being released for operation on a Xbox 360, open-source drivers have been developed to interface it to a computer (such as the `libfreenect`[2]).

The depth map is computed by hardware onboard the Kinect and made available in the form of an image via its USB interface. Depth maps are achieved by projecting a known pattern of infra-red (IR) light onto the scene, followed by processing the IR camera's video feed. The depth is estimated by finding how the pattern has been deformed by the scene. Thus the Kinect's depth sensing system is a type of structured light sensors [116]. The key specifications [116] pertaining to the Kinect's vision sensors are:

Colour camera — 640 x 480 RGB (8-bit) resolution, 30 frames per second (FPS), 57 ° horizontal and 43 ° vertical field of view (FOV).

Depth camera — 640 x 480 monochrome (11-bit) resolution, 0.4–7.0 m range, 30 FPS, 57 ° horizontal and 43 ° vertical FOV.

[1] `http://xv11hacking.wikispaces.com/LIDAR+Sensor`
[2] `https://github.com/OpenKinect/libfreenect`



(a) Angular displacements from the optical axis are encoded into pixel location.

(b) Mounting geometry of the Kinect.

Figure 7.2: Image and mounting geometry diagrams for emulating a laser scanner from a Kinect's depth map.

Using the `libfreenect` driver, a Player plugin was developed to emulate a laser rangefinder from the Kinect's depth map. Note that the resulting algorithms are dictated by the choice of mounting of the Kinect. One approach is to mount the Kinect in a similar fashion to a laser scanner (see Figure 6.6 (a)). Using only the depth map's middle row, a range-bearing scan can be easily derived. However, it would be difficult to mount a Kinect inconspicuously on a wheelchair's footrest. Instead, the Kinect was mounted on a pole behind the wheelchair's seat, as shown in Figure 6.6 (b). This arrangement not only makes installation easy but it also compensates for the Kinect's minimum sensing distance.

The pixel values (v) in the depth map are encoded as 11-bit values. Equation (7.7) was used to transform these values into actual distances [1], while (7.8) computes the distance to a particular angular displacement:

$$z(\theta,\beta) = \begin{cases} 0.1236\tan(\frac{v(\theta,\beta)}{2842.5} + 1.1863)\ , & \text{if } v(\theta,\beta) < 2047 \\ 0\ , & \text{otherwise} \end{cases}\ , \tag{7.7}$$

$$d(\theta,\beta) = \frac{z(\theta,\beta)}{\cos(\theta)\cdot\cos(\beta)}\ . \tag{7.8}$$

Assuming an ideal pinhole projection model, the horizontal (θ) and vertical (β) angular displacements from the optical axis can be calculated from column and row position of a pixel in the depth map, as shown in Figure 7.2 (a). Thus, $\theta = \arctan((1-2\cdot\text{row}/w)\cdot\tan(0.5\text{FOV}_x))$, and $\beta = \arctan((2\cdot\text{col}/h-1)\cdot\tan(0.5\text{FOV}_y))$. Figure 7.2 (b) shows the mounting geometry of the Kinect. Since the pose of the Kinect is known and assuming that the wheelchair drives on flat floor, the expected

[1] The transformation equation between raw depth values to metres was proposed by Stephane Magnenat on the OpenKinect Google group.

position of the floor can be calculated for each pixel:

$$d_{floor}(\theta, \beta) = \frac{H}{\cos(\alpha - \beta) \cdot \cos(\theta)} \ . \tag{7.9}$$

If an obstacle is in front of the wheelchair, the Kinect's depth map will return distances less than the expected distance to the floor for a corresponding pixel. Algorithm 3 shows the algorithm used to derive a laser scan to the closest frontier, from the Kinect's depth map.

Input : a 2-D depth map and a history of laser scans H
Output: a derived laser scan S

Set each element of S to maximum sensing range
foreach row r in depth map m **do**
    Calculate β from row position in map
    foreach element in current row r **do**
        Calculate θ from column position in map
        Compute distance d from pixel value and angular displacements
        Compute expected distance to the floor d_{floor}
        if d is less than d_{floor} **then**
            Project d onto floor: $r = d \cdot \sin(\alpha - \beta)$
            if r is less than $S(\theta)$ **then**
                $S(\theta) = r$
            end
        end
    end
end
if history of scans H **then**
    Perform filtering between scans
end

Algorithm 3: The algorithm used to derive a laser scan from the Kinect's depth map. Note that not all rows and columns are sampled to speed up computation.

7.2.6 GMapping wrapper

A wrapper was developed to interface the GMapping SLAM library[1] with Player. As discussed in Subsection 4.4.4, GMapping uses a grid-based map representation and requires a range-bearing laser scanner and odometry estimates. The ROS (see Subsection 3.1.3) GMapping wrapper package[2] was particularly useful as a reference to using the GMapping library. This subsection describes the developed GMapping plugin.

At startup, the GMapping plugin attempts to connect to the specified `laser` and `position2d` (raw odometry) interfaces. These interfaces are required and must be provided by other Player drivers/plugins. Following the successful connection to

[1] `http://www.openslam.org/gmapping.html`
[2] `http://www.ros.org/wiki/gmapping`

these interfaces, they are queried for their geometry information, e.g., the laser's pose relative to the robot's axes. This information, along with the GMapping specific configuration parameters (error values for the laser and odometry sensor models, initial map size and resolution, and computation-accuracy tradeoff settings), are used to configure GMapping's initialisation function. The wrapper also creates `map` and `position2d` (pose relative to the map) interfaces.

Each time a laser scan or a robot pose message is published by a driver on the Player server, it is passed onto the GMapping wrapper. This is manipulated to conform to GMapping's `RangeReading` (a laser scan with an associated pose) data type, and then passed onto GMapping's `processScan` function. The result returned by this function is used to determine if the GMapping plugin's map needs to be updated (to avoid unnecessary computation). There are several factors that determine when the map should be updated. They are set by GMapping's initial configuration parameters, including: the time since the last update and if the robot has travelled more than a linear (and or angular) distance since the robot's last update pose.

When a map is requested from the GMapping plugin, it has been designed to return the last map computed by a map update function[1]. This update function is called each time the result from GMapping's `processScan` function returns true and runs in its own thread. The map update function consists of the following steps:

1. Check to see if the map needs to be updated, and if so, proceed onto the next step. Otherwise, sleep the thread and check again later.

2. Initialise an instance of GMapping's `ScanMatcher` and `ScanMatcherMap` objects, using the parameters of the laser scanner (number of readings, field of view, angular resolution, and the relative laser scanner pose) and the output map (its dimensions and resolution) respectively.

3. Using the best particle (i.e., the highest weighted particle), traverse its trajectory tree. Provided that a reading exists at the current node, build up the scan matcher map using the node's pose and its associated laser scan.

4. Check to see if GMapping has resized the map. If it has, appropriately reallocate memory.

5. For each pixel in the scan matcher map, convert cell probabilities into discrete values that describe the cell as either unknown, free space, or occupied by an obstacle. This data now becomes the last map.

6. Provided that the map update thread is still alive (i.e., Player has not been shutdown), return to the first step.

[1]If the map's data is too big to fit in a message, it is returned in tiles. The sub-tiling requests and re-tiling processes are automatically done by the `map` client interface. By default, Player has a tile size limit of 640 x 640.

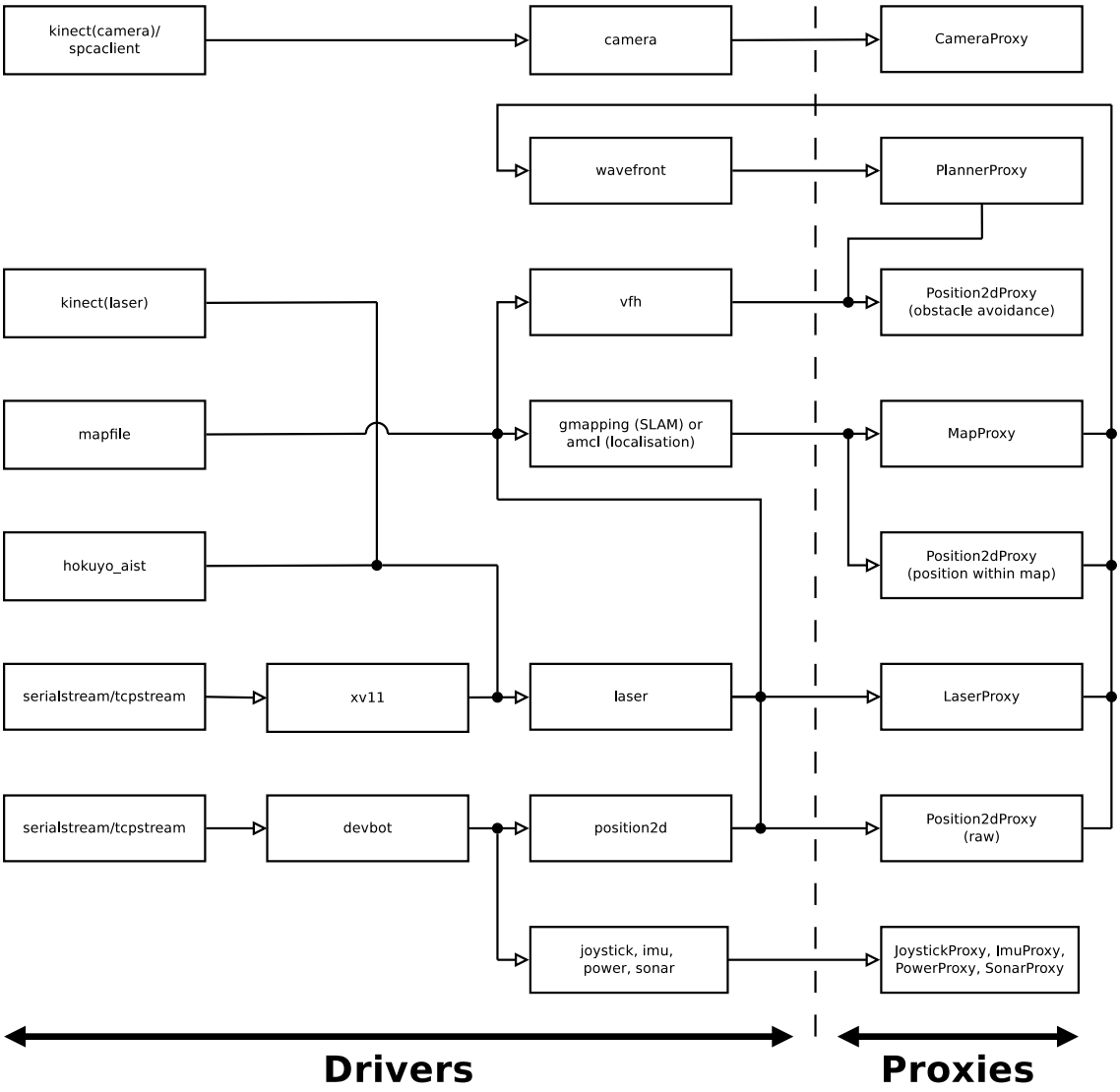


Figure 7.3: A high level block diagram showing the configuration of the Player server.

7.3 Player setup

This section outlines the setup of the Player server. Player is configured at run-time by passing it a text-based configuration file as a command line argument. The configuration file contains a list of drivers, where each driver entry specifies the interface(s) it provides, the interface(s) it requires, and if necessary, setup parameters. The essence of the configuration files used to setup Player for the small robot (see Section 6.1) and the wheelchair (see Section 7.3) shows all drivers and interfaces used between the small robot and the wheelchair setups.

As it can be seen from Figure 7.3, despite the differences between the small robot and the wheelchair, the structure of the Player setup remains mostly the same. This is also true in the case of the virtual Pioneer 2-DX robot in the Stage simulation environment. Other key points about Figure 7.3 include:

- Although three `position2d` interfaces are available to the various drivers and client program (see Section 7.4), it is up to each module to use them appropriately. For instance, the `position2d` provided by `gmapping` or `amcl` only provides odometry information relative to the map.
- There are up to three drivers that are capable of supplying laser data to the `LaserProxy`: a Hokuyo URG-04LX (`hokuyo_aist`), a XV-11 (`xv11`), and a virtual laser scanner from a Kinect camera's depth data. Only one laser scanner type is used at a time. Each of these drivers are configured to store information about its range limit, field of view, angular resolution, and its pose relative to the robot's centre of rotation. Thus the differences in laser type do not propagate through to the rest of the navigation system, as these parameters are taken into account by the immediate downstream drivers.
- The `wavefront` driver is used for path planning (a description of `wavefront` is given in Subsection 5.2.1 and in Section 5.3), and provides a `PlannerProxy`. When a specified destination pose is given, `wavefront` computes an efficient route to it, and stores way points in the `PlannerProxy`. The robot is autonomously navigated to the destination when `PlannerProxy` receives a start signal (in this case from the GUI, see Section 7.5), and sends velocity commands to the `Position2dProxy` provided by the obstacle avoidance driver (`vfh`).
- The mapping and localisation component of the navigation system is determined when starting up the Player server. If SLAM is to be performed, the `gmapping` plugin is used to provide a `MapProxy` and a `Position2dProxy` (giving the pose within the map). Alternatively, if the environment has been adequately mapped, `amcl` is used instead.

7.4 Player client

A client program that interfaces to the Player server was developed. It can be seen in Figure 7.4 that the client program is relatively simple and consists of a few modules. This is the case as the user interface (discussed in Section 7.5) has been completely decoupled from the client program and the bulk of the navigation algorithms are compiled as a Player driver or plugin.

An important module of the client program is the composite map to image generator. It pulls data from the `MapProxy` and converts it into an OpenCV image. This task is straight-forward (for each element in the 2-D grid map, convert it into an appropriately coloured pixel) and the image is subsequently displayed in the GUI. Depending on the selected display options in the GUI, the map image is also overlaid with other items, including: the robot's current pose, the robot's trajectory, the way points in the `PlannerProxy`, and the destination pose. A local map (a 4 m square area with the robot in the centre) is also generated. This is achieved by cropping the map image about the robot's pose and rotating it to align with the robot's co-ordinate frame. A laser scan is also overlaid on the local map image. A screen capture of the map image and the local map image displayed in the GUI is shown in Figure 7.5.

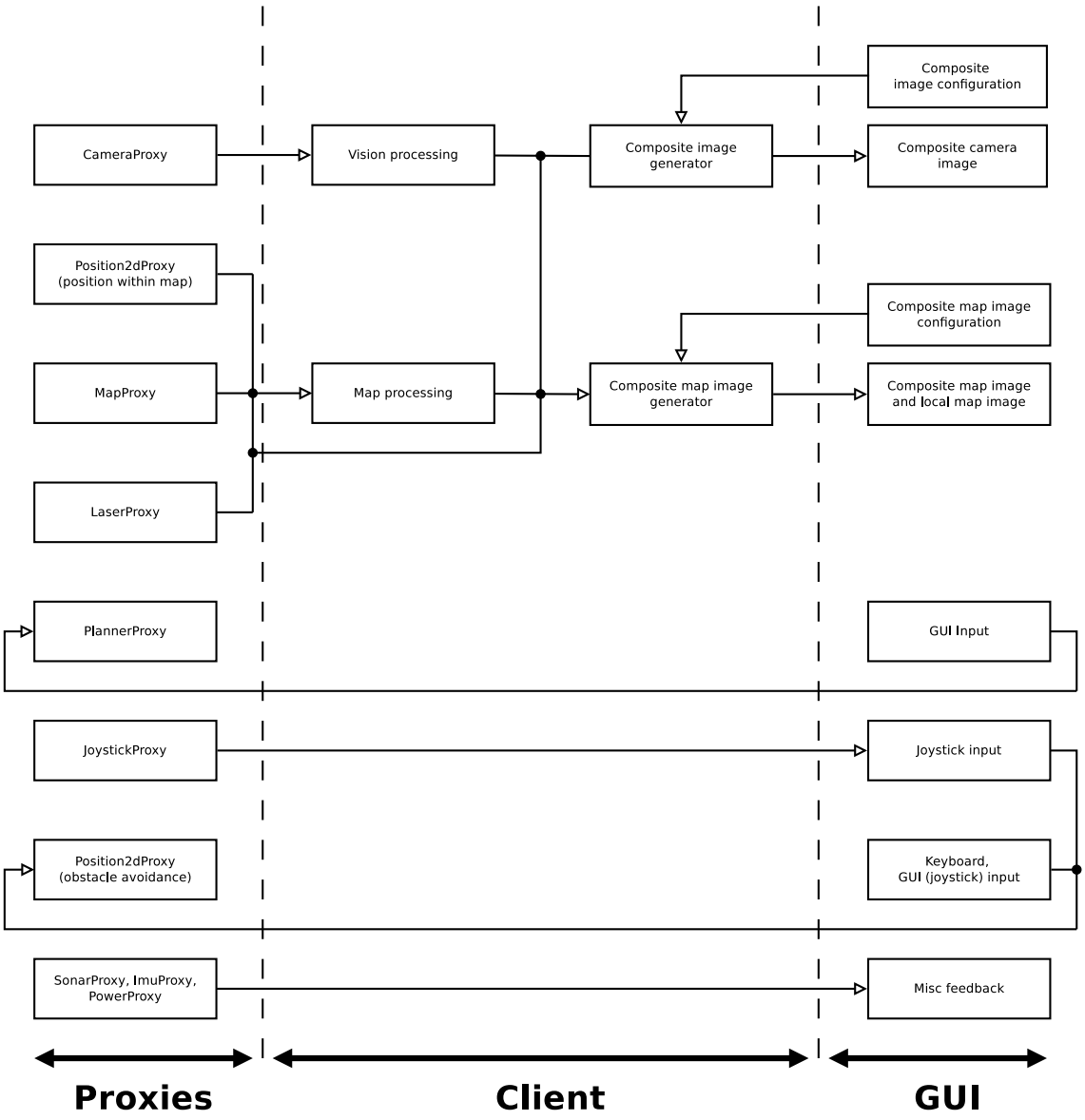


Figure 7.4: A high level block diagram showing the fundamental components of the client program.

Another aspect of the client program is vision processing. All this does is relay frames from the robot's onboard video camera to the GUI application and was implemented as a stub for future development towards this navigation system.

In order to handle the differences between instrumentation of the robots (the virtual Pioneer 2-DX, the small robot, and the wheelchair), the client program firstly queries the Player server for a list of available interfaces. All of the three robots provide two `position2d` and one `laser` interface. In addition to these, the small robot provides `camera`, `imu`, `sonar` and `power` interfaces. On the other hand, the wheelchair provides `camera` and `joystick` additional interfaces. The client program then subscribes to the available interfaces and appropriately sets up its own modules based on these subscriptions. Other main tasks that happen during startup of the client program include:

- The default timeout of the Player server is increased to 10 seconds. In the configuration of performing SLAM while allowing autonomous navigation, the map used by the `wavefront` driver needs to be refreshed each time a new destination pose is given. This subsequently requires re-computing C-space (a time consuming task), and would cause Player to shut down with the message default timeout (5 seconds).

- The Player server is queried several times with some dummy reads. Without doing so, it was found that some of the interfaces contained bad data.

7.5 User interface

A design decision was made to completely separate the client program from the user interface. This was motivated by the principle that good software design treats the user interface as a separate task to the underlying application and that Python is more suited than C/C++ to developing front-end applications. Although Player does come with a Python interface (automatically generated Python bindings to its `libplayerc/libplayerc++` libraries through SWIG[1]), I was more familiar with using Player under C++. Moreover, the client-gui separation provides another level of abstraction; the GUI application is only exposed to the functions essential to its operation. The client program (discussed in Section 7.4) was compiled as a shared library, and the GUI application interfaces to it through Python's `ctypes`[2] package.

Figure 7.5 shows a screen capture of the GUI. The GUI was developed using Glade[3], a user interface designer to work with the GTK+ toolkit under the GNOME desktop environment. Although a GTK+ based Python GUI can be developed using the PyGTK[4] library, Glade speeds up this process. The key features of the GUI include:

[1] `http://www.swig.org/`
[2] `http://python.net/crew/theller/ctypes/`
[3] `http://glade.gnome.org/`
[4] `http://www.pygtk.org/`

- Displays the current map, a local map, and if an onboard camera is available, shows the video stream. These tasks ran in a separate thread to the main GUI thread, so that they can be updated at a lower refresh rate to reserve system resources while not affecting the GUI's usability.
- The map display can have the following overlaid on it: the robot's current pose, the robot's trajectory, the path planner's way points, and the destination pose. Provisions have been made to allow the video stream to be also overlaid with items of interest.
- The GUI accommodates all three robotic setups (simulated robot, small robot, and the wheelchair) without requiring modifications. This is achieved by querying the client program for the available interfaces, then appropriately enabling (or disabling) features of the GUI.
- It has a terminal output (hidden by default) that displays vital text messages.
- Ability to save maps (via the file menu dropbox).
- Provides a variety of user input methods, including:
  - A joystick on the GUI that can be operated with a mouse click or with a touch screen interface on a tablet computer.
  - Joystick control from the DX2 system.
  - Keyboard.
  - Autonomous navigation towards a goal.

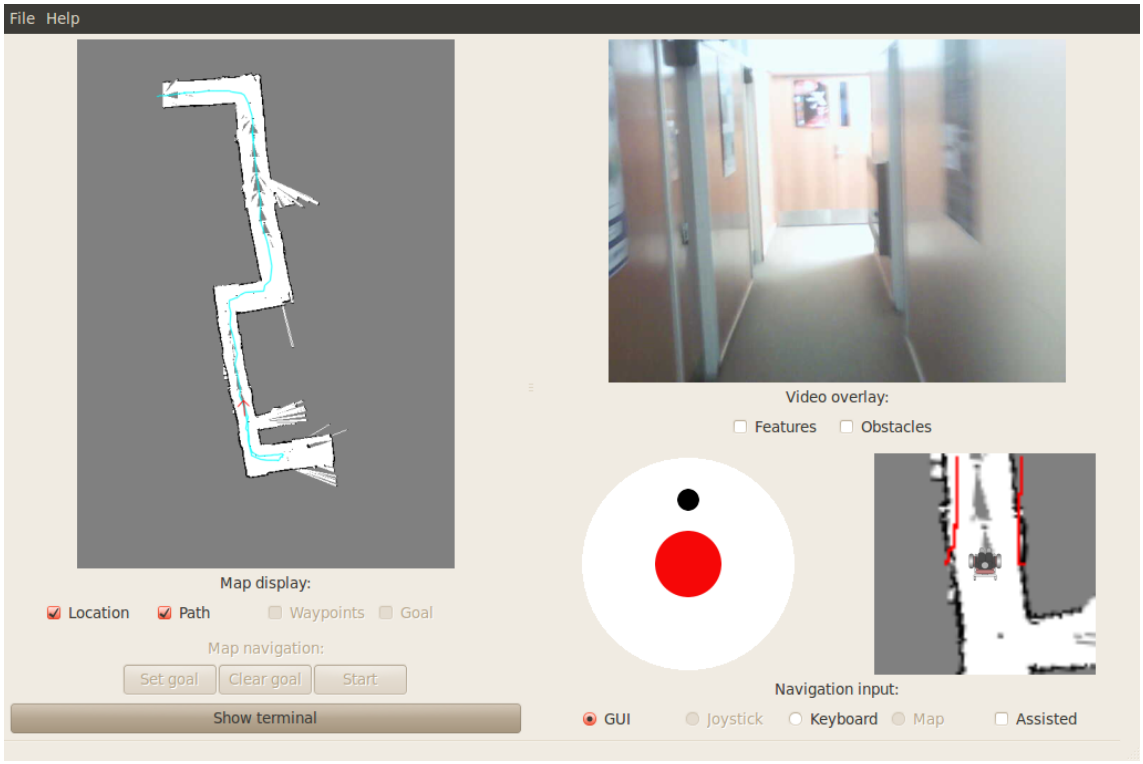


Figure 7.5: A screen capture of the GUI.

Chapter 8

Experiments

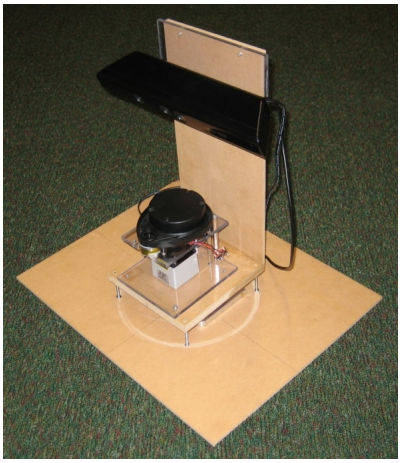
This chapter presents results from running the navigation system outlined in Section 7. The navigation system is evaluated on both simulated and physical robots, to allow both rapid analysis and show its real world applicability. Section 8.1 presents comparisons between three different laser scanners, which are later used in mapping an environment in Section 8.3. An analysis of the errors from the physical wheelchair's dead reckoning system is given in Section 8.2. Evaluation of the obstacle avoidance and path planning components on the navigation system are presented in Section 8.4.

Although research was conducted into localisation techniques given a map of the environment (see Subsection 5.2.3), results on such experiments have not been presented in this thesis. The was because Player's localisation driver (`amcl`) was found not suitable for this project. In experiments, it was found that `amcl` can converge to the correct robot pose. However, correct convergence would only occur if the initial guess pose was close to the true pose. Resolving this limitation is left as future work.

8.1 Laser scanner evaluation

A major cost of any robotic system is its sensors. Thus an underpinning consideration on the selection of sensors to instrument the wheelchair was their cost. As discussed in Subsection 6.2.2, the Pronto M51 testbed wheelchair was retro-fitted with three different laser scanners: a Hokuyo URG-04LX (about $1,300 USD), a XV-11 (salvaged from a robotics vacuum cleaner that cost $400 USD), and a virtual laser scanner derived from a Kinect (the Kinect costs around $150 USD). This section presents experiments that quantify the relative accuracy and precision of these three laser scanners, followed by a discussion of their relative merits.

Figure 8.1 shows the test jig and the test environment. The test jig consisted of a rigid frame that was attached to a rotating base. Each of the three laser scanners were attached to the frame and their scans were logged simultaneously using the `writelog` Player driver. The angular position of the rotating base was also recorded. A bare, full height, internal corner wall was used as the test environment, as the ground truth range-bearing measurements can be easily computed from a



(a) The laser test jig.

(b) The test environment.

Figure 8.1: The test jig and test environment for evaluating the laser scanners.

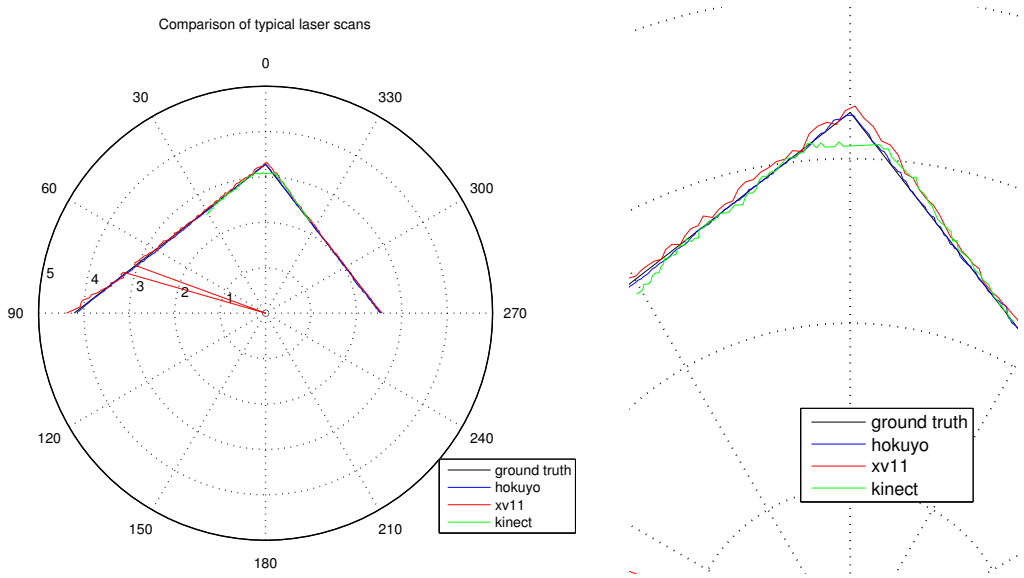
distance and an angle measurement. The test jig's base was clamped to a high table in the environment. Both static and dynamic (by rotating the sensors with respect to the static environment) tests were performed. Note that the scans from the URG-04LX and the XV-11 were reduced to that of the Kinect to make the results more comparable to one another.

8.1.1 Static tests

A typical static scan from each of the three laser scanners have been overlaid in Figure 8.2. As expected, the expensive Hokuyo scanner closely resembles the ground truth. The XV-11 rangefinder also returns impressive results, but its scan is prone to having missing sectors. This is despite a moderate level of filtering performed in the XV-11 plugin (see Subsection 7.2.4). The algorithm discussed in Subsection 7.2.5 that derived a laser scan from the Kinect's depth map has also demonstrated to closely resemble the ground truth. However, due to excessive smoothing of the Kinect's depth map, the derived laser scan often rounds corners, as shown in Figure 8.2 (b).

The static test of the laser scanners consisted of logging the scans over approximately one minute duration. The environment conditions were not altered during the test. For each measurement in a scan, the average absolute difference between the ground truth was computed. This process was repeated over all recorded scans. Figures 8.4 (a), 8.5 (a), and 8.6 (a) show the results that came from the static experiments.

As suggested by Figure 8.4 (a), the Hokuyo scanner tends to have an accuracy of 0.01 m, is quite precise and its errors are approximately Gaussian distributed. This is also suggested by work done by other research [117, 118]. On the other hand, the XV-11 scanner was found to have two modes of error distribution, as pictured in Figure 8.5 (a). This is likely to be attributed to missing sectors or readings in the scan. Although more filtering could alleviate this problem, it would lead to greater lag in the scans (which is a problem with its low scan rate). The experiments also



(a) Laser scans.

(b) Close up of laser scans.

Figure 8.2: Static scans from each of the three laser rangefinders. Due to excessive smoothing onboard the Kinect sensor, scans of an environment with sharp edges appear rounded.

suggest that the XV-11 most common mode of accuracy is around 0.045 m. This is roughly in accordance with its reported accuracy of 0.03 m [115].

The static test on the Kinect 'laser' scanner suggest that the accuracy is, in the worst case, 0.05 m. Also, as shown by Figure 8.6 (a), the errors generally appear to be flatly distributed. This either infers inadequacies in Algorithm 3 or that the errors are related to a linear variable. This linear variable could be the column position in the Kinect's depth map that a particular range measurement is derived from, or correlated with the environment lighting conditions. Further investigation was not conducted into this matter as it was never expected that the Kinect would be as accurate as the other laser scanners.

8.1.2 Dynamic tests

A dynamic test was carried out in order to estimate the practical working accuracy and precision of the laser scanners. Instead of having moving objects within the sensor's field of view (FOV), the sensors themselves were rotated about the static environment. The test jig's frame was rotated manually by hand. For many SLAM algorithms to work, the environment must be mostly static, and it often is, thus the dynamic tests should better indicate the laser scanner errors in practice.

As with the static tests, all laser scan data was logged simultaneously along with the angular displacement of the test jig. Figure 8.3 shows the angular displacement and velocity profile for the dynamic tests that were carried out. Due to the limited sensing ranges of the laser scanners, the FOV of the URG-04LX and the XV-11 were reduced to allow a wider range of angular displacements. They were reduced to that of the Kinect scanner, i.e., 57 °. Since each of the laser scanners has different update

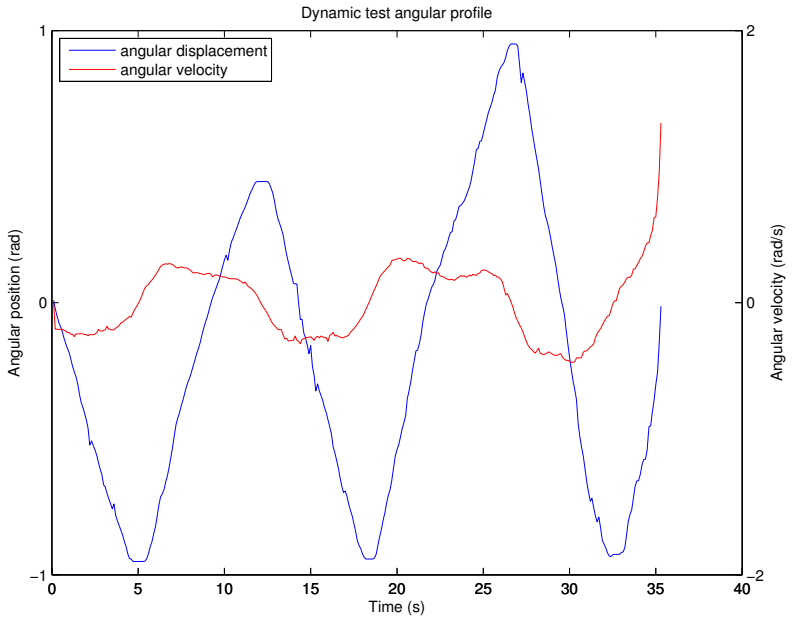


Figure 8.3: Angular velocity profiles of the dynamic tests for the laser rangefinders.

rates (URG-04LX: 10 Hz, XV-11: 5 Hz, Kinect: 30 Hz) and that they are not pose-stamped, each scan for each laser was linearly interpolated from pose information that was published by the Player server at 10 Hz.

Figures 8.4 (b), 8.5 (b), and 8.6 (b) show the results that came from the dynamic tests. As shown by the figures, the estimated accuracy and precision of all three lasers in the dynamic tests were poorer compared to the static tests. This is to be expected, as a dynamic environment is simply more challenging to measure. It can be seen in 8.4 (b) that the average measurement error for the URG-04LX seems to be generally less than 0.025 m. It was also found that the URG-04LX, in a dynamic environment, sometimes has missing sectors in its scan. These missing sectors result in large errors, hence the outliers in the histogram. Fortunately, these missing sectors do not occur very often (and the frequency of which are environment specific).

The dynamic tests also revealed that the measurement errors for both the XV-11 and the Kinect 'laser' get substantially worse; Figure 8.5 (b) suggests an error of about 0.05 m for the XV-11, while Figure 8.6 (b) suggests an error of 0.1 m for the Kinect derived laser. The source of the bulk of these errors is believed to be attributed to phase lag between the laser scan and the angular position. It was estimated from experimentation that an angular rate of greater than 2 rad/s leads to lag in the XV-11's scan, and similarly a rate of 1 rad/s for the Kinect 'laser'. The cause of this error is due to filtering (and for the Kinect, most of this is carried out in the sensor).

Although the dynamic tests revealed reduced accuracy for all of the rangefinders (particularly for the XV-11 and the Kinect derived laser scanner), it does not necessarily infer substantially worse performance when applied to SLAM. In practice, the sensors on the wheelchair undergo a combination of translational and rotational motion. Moreover, modern SLAM methods are probabilistic in that bad sensor data tends to be filtered out. This of course requires that there are sufficiently more

consistent scans than bad ones.

8.1.3 Final comparisons

In Subsections 8.1.1 and 8.1.2 the performance of the three laser scanners was evaluated through a series of simple tests. From these tests, it was unsurprising that the most expensive scanner, the URG-04LX, performed the best. However, the XV-11 and the Kinect derived scanner proved to be capable as well. This subsection provides a summary of the laser scanner comparison.

In the laser scanner tests, the FOV of the URG-04LX and the XV-11 were reduced to match that of the Kinect 'laser'. However, having a large FOV is highly beneficial for SLAM algorithms. In the case of GMapping, a larger FOV means that each scan captures more of the environment, and consequently provides more 'features' for its scan matcher to compute the change in pose between subsequent scans. On the other hand, the Kinect 'laser' has a key advantage over the other scanners in that by using Algorithm 3, the 2-D laser scan it returns is the distances to the closest frontier of obstacles. This is is highly beneficial as it will detect the presence of obstacles at all heights, as opposed to the distance to obstacles at the same level of the laser scanner. Other comparisons between the scanners are made in Table 8.1.

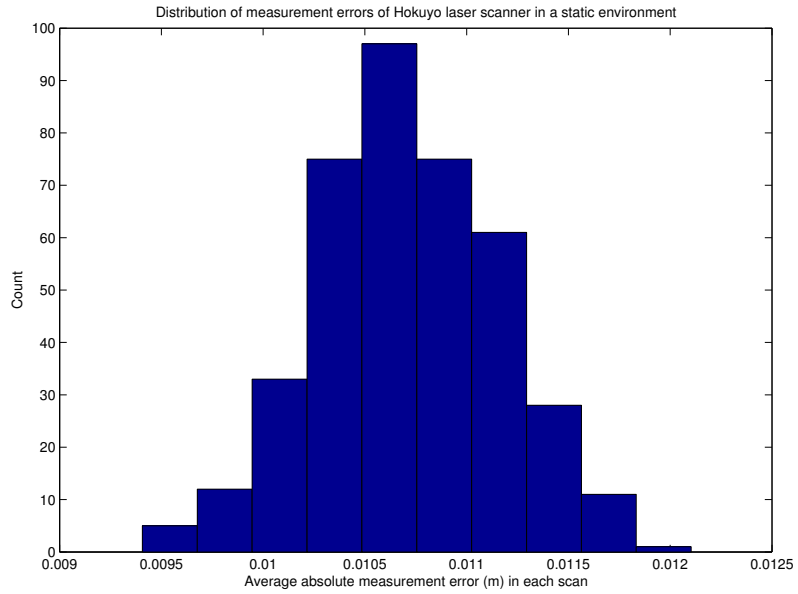
8.2 Odometry error analysis

Dead reckoning from odometry is the process of estimating the robot's current pose based on a previous pose. In the case of the wheelchair, and as with many other wheeled robotic platforms, the pulses from wheel encoders are periodically read and are used to update dead reckoning algorithms (see Subsection 7.1.2). However, the accuracy of the dead-reckoned pose with respect to the true pose reduces over time as errors accumulate without bound. Subsection 8.2.1 presents analysis on the wheelchair's dead reckoning errors, while Subsection 8.2.2 discusses the odometry model in the Stage simulation environment.

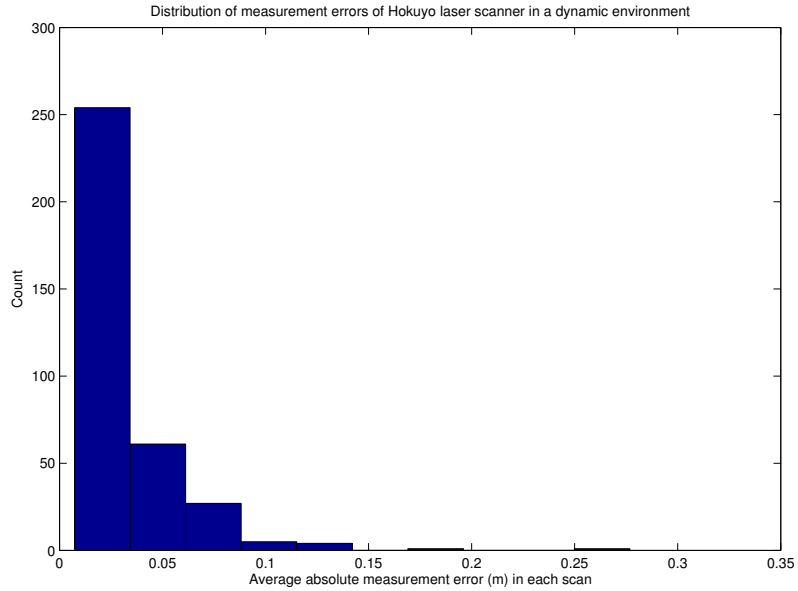
8.2.1 Wheelchair odometry

Odometry errors can be both systematic and non-systematic [119]. As discussed by Borenstein et al. [120], the main sources of systematic errors are: unequal wheel diameters, misalignment of the wheels, assumptions made about the robot's centre of rotation, and limited encoder resolution. Another potential source of systematic error in a digital system is from a finite machine epsilon and the particular rounding method used in the implementation's arithmetic. To identify systematic errors, straight and circular trajectory tests were performed. The test environment, shown in Figure 8.7, consisted of following an edge on the floor for 30.00 m and a circular track of radius 2.18 m made from masking tape.

The first step of estimating systematic errors was to accurately compute the wheel diameters. This was achieved by pushing the wheelchair along the 30.00 m

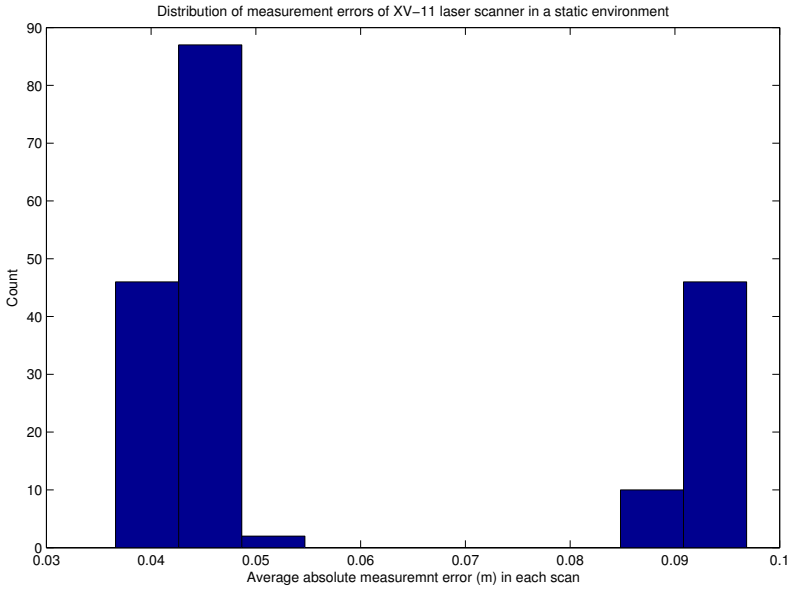


(a) Static test ($\mu_{error} = 0.0024\,m$, $\sigma_{error} = 0.0011\,m$, $\mu_{|error|} = 0.011\,m$, $\sigma_{|error|} = 0.00045\,m$).

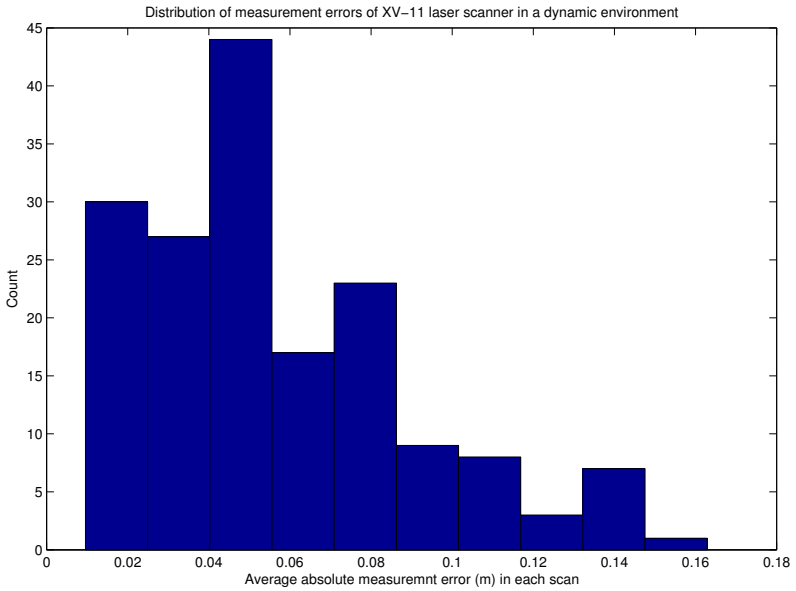


(b) Dynamic test ($\mu_{error} = 0.0071\,m$, $\sigma_{error} = 0.023\,m$, $\mu_{|error|} = 0.030\,m$, $\sigma_{|error|} = 0.028\,m$).

Figure 8.4: Evaluation of the Hokuyo URG-04LX laser scanner.

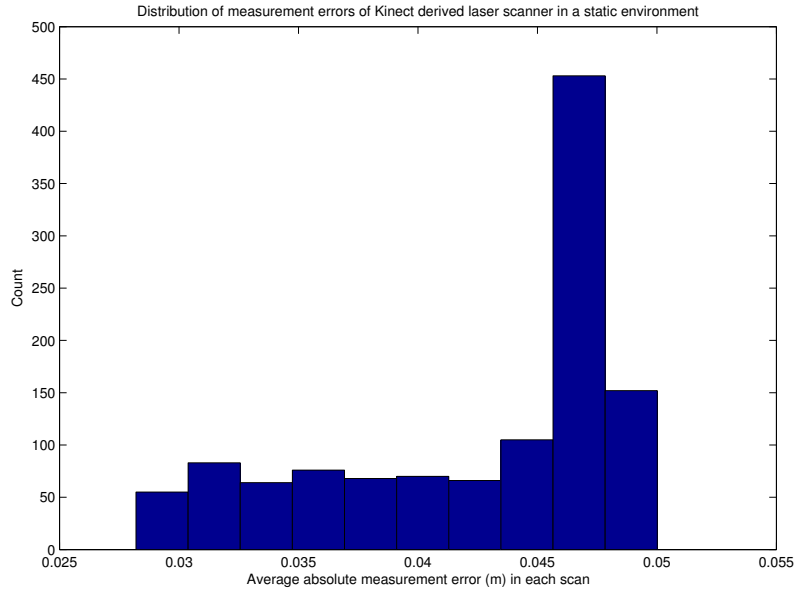


(a) Static test ($\mu_{error} = -0.017\,m$, $\sigma_{error} = 0.023\,m$, $\mu_{|error|} = 0.058\,m$, $\sigma_{|error|} = 0.022\,m$).

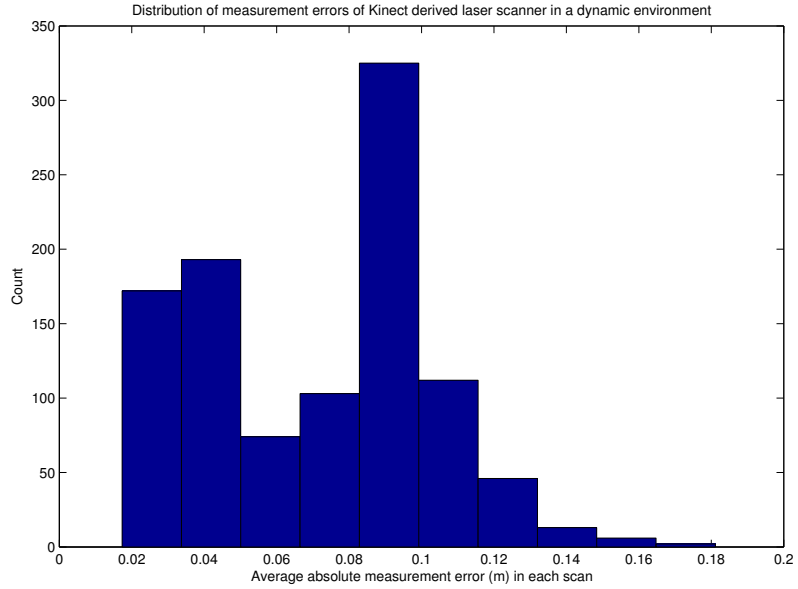


(b) Dynamic test ($\mu_{error} = 0.0051\,m$, $\sigma_{error} = 0.037\,m$, $\mu_{|error|} = 0.057\,m$, $\sigma_{|error|} = 0.033\,m$).

Figure 8.5: Evaluation of the XV-11 laser scanner.



(a) Static test ($\mu_{error} = 0.0080\,m$, $\sigma_{error} = 0.0014\,m$, $\mu_{|error|} = 0.043\,m$, $\sigma_{|error|} = 0.0061\,m$).



(b) Dynamic test ($\mu_{error} = 0.042\,m$, $\sigma_{error} = 0.058\,m$, $\mu_{|error|} = 0.15\,m$, $\sigma_{|error|} = 0.036\,m$).

Figure 8.6: Evaluation of the Kinect derived laser scanner.

Laser	Arguments for	Arguments against
URG-04LX	• Accurate and precise. • Large FOV. • Compact. • Low power requirements. • Low computational overhead.	• Relatively expensive. • Specialised part and must be imported. • Only measures ranges to obstacles in a 2-D plane.
XV-11	• Reasonably accurate and precise. • Full 360 ° FOV. • Compact. • Low power requirements. • Low computational overhead. • Reportedly cheap to produce [115].	• Low scan rate introduces lag between the scan and pose in dynamic environments. • Prone to have missing sectors in its scan. • Have to salvage it from a product (sensor is not sold by itself). • Requires hardware not commonly on modern computers to interface to it. • Only measures ranges to obstacles in a 2-D plane. • Difficult to source in many countries.
Kinect derived	• Cheap and readily available. • Contains other useful sensors. • Moderately low power requirements. • Allows tradeoff between computation and scan resolution. • Capable of computing the distance to the closest frontier of obstacles.	• High computational overhead and too demanding for use on current ARM embedded systems. • Uses up a substantial amount of a system's USB bandwidth. • Despite the 30 Hz depth map rate, it has a high amount of lag between the scan and pose in dynamic environments. • Lower accuracy and precision. • Limited field of view. • Heavily smoothed depth map gives loss of fidelity.

Table 8.1: A summary of the comparisons between the three laser scanners.

(a) Straight test track.

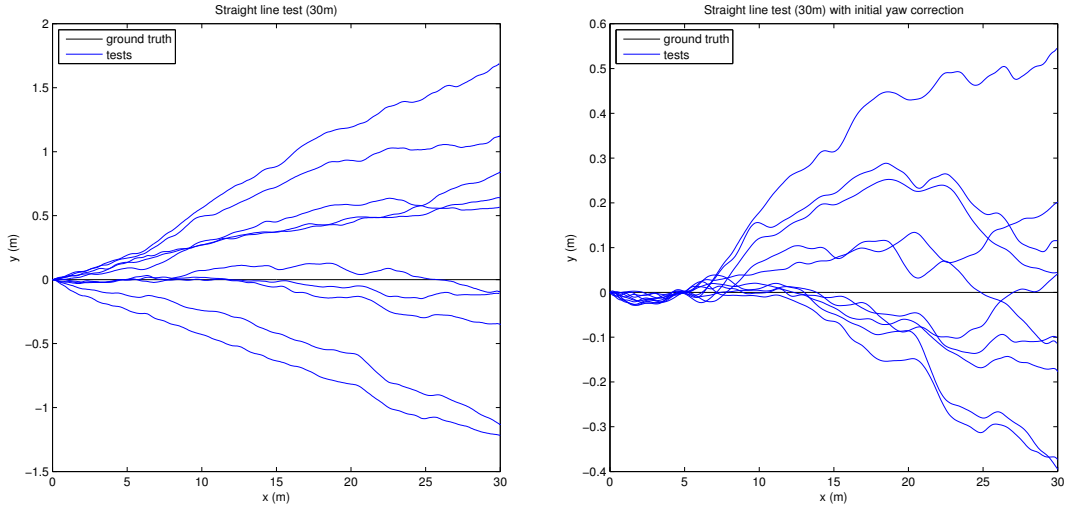
(b) Circle test track.

Figure 8.7: The test environment used for evaluating odometry errors.

straight line track and recording the raw encoder counts. This experiment was completed 10 times, and it was computed that the mean left and right diameters were 0.254 m (standard deviation of 0.013 m) and 0.257 m (standard deviation of 0.021 m) respectively. Using the calculated wheel diameters and a measured wheel separation value of 0.555 m, the straight line tests were then re-performed. Figure 8.8 (a) shows the odometry computed by (7.2) onboard the embedded controller. In each of the tests, the wheelchair was accurately aligned position-wise with the straight line track. However, it was difficult to the set the initial heading accurately. This explains the initial yaw bias observed in many of the tests shown in Figure 8.8 (a). The initial heading was later estimated, and Figure 8.8 (b) shows the corrected results. From Figure 8.8 (b), it was deduced that the systematic errors from the wheel diameter measurements have been accounted for, as the corrected results appear to have no systematic bias in them.

Another large potential source of systematic error could be from the measurement error of the separation of the wheels (D). To test this hypothesis, 10 tests were conducted on the circular track shown in Figure 8.7 (b). The wheelchair was pushed clockwise for one complete revolution around the circular track. The results from these tests are shown in Figure 8.9. From these experiments, it was deduced that the change in heading in each pose update has been slightly overestimated. Over a single loop, the overestimation in yaw is too small to meaningfully calculate the required correction factor to D.

Instead, a single test consisting of 10 loops was conducted, and the result of this is shown in Figure 8.10 (a). Over the 10 loops, the distance error was estimated to be 0.587 m (0.43%), thus it is a reasonable assumption that minimal systematic error in this test is from wheel diameter inaccuracies. In the test, it was found that the yaw was 61.8 ° too far, thus over 3,600 degrees D needs to be 1.017 times greater to compensate, giving a calculated new value of 0.565 m. The test was performed again using this value. It can be seen in Figure 8.10 (b) that the trajectory of the wheelchair is now closer to the ground truth. However, it is apparent that there are still other sources of systematic error present. This could be from:



(a) Straight line tests. (b) Corrected straight line tests.

Figure 8.8: Odometry results from conducting 10 straight line tests (over 30.00 m).

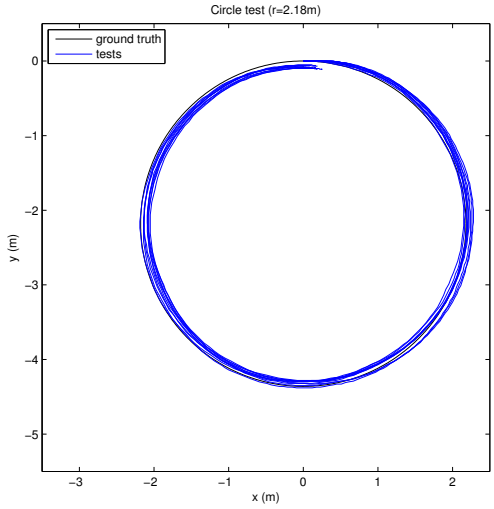
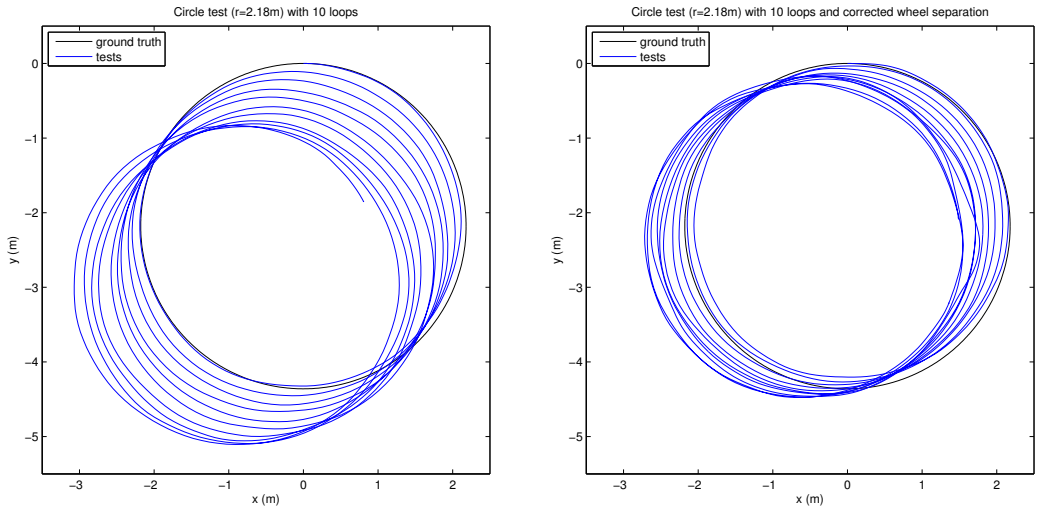


Figure 8.9: Odometry results from conducting 10 circle tests (radius of 2.18 m).



(a) Loop test with $D = 0.555\,\text{m}$. (b) Loop test with $D = 0.565\,\text{m}$.

Figure 8.10: Odometry results from conducting 10 loops (radius of 2.18 m).

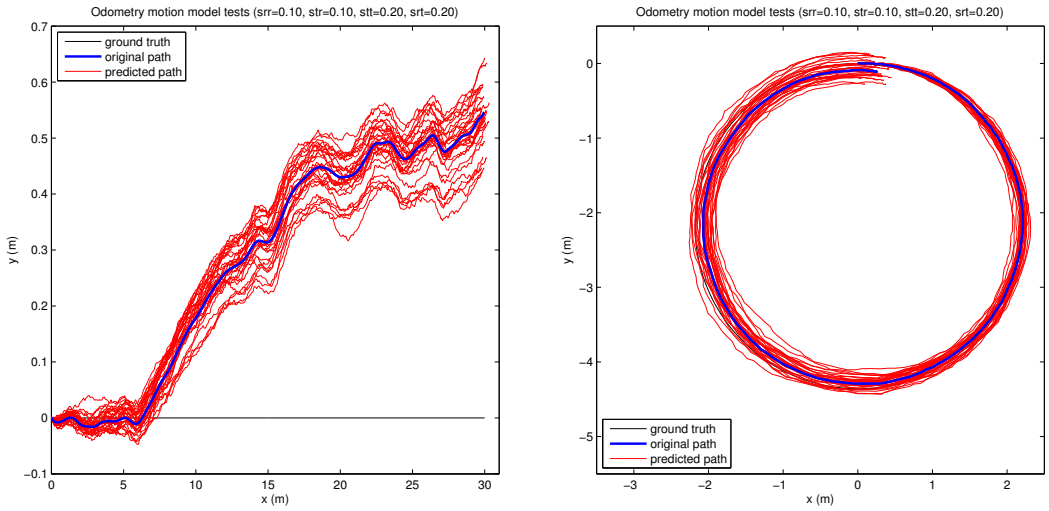
- Systematic rounding errors (which is typical when integrating).
- Deficiencies or over-simplifications in the dead reckoning equations (7.2).
- Systematic wheel slipping.

Unfortunately, all of these possible sources of systematic errors would require more complicated dead reckoning equations. For the purposes of this work, the current odometry equations have yielded satisfactory performance. Moreover, it is expected that in practice the user will travel mostly in straight lines and not perform excessive numbers of consecutive circles.

The work done by Kleeman [121] describes a motion model and an approach for quantifying random errors in robot odometry. The models described in [121] are validated in simulation, primarily as simulation allows noise-free results to be generated and the ground truth is readily attainable. Another approach in compensating for odometry uncertainty is to augment odometry and laser rangefinder readings to characterise systematic errors [119]. This principle of using landmarks (extracted from laser scans) to bound odometry-derived pose errors is used in SLAM. Thus many SLAM implementations handle the errors present in dead reckoning by defining an odometry motion model. The odometry model in GMapping assumes that the associated update pose errors are of a zero-mean Gaussian distribution that is described using four parameters: s_{rr}, s_{rt}, s_{tr}, and s_{tt}. Upon receiving a new pose s_t, GMapping's odometry model adds errors to the dead reckoned pose:

$$s'_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}'_{t-1} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} + \begin{bmatrix} S(s_{rr}|\Delta x| + s_{tr}|\Delta\theta| + s_{xy}|\Delta y|) \\ S(s_{rr}|\Delta y| + s_{tr}|\Delta\theta| + s_{xy}|\Delta x|) \\ S(s_{tt}|\Delta\theta| + s_{rt}\sqrt{\Delta x^2 + \Delta y^2}) \end{bmatrix}, \tag{8.1}$$

where s'_t is the pose with error, $S(\sigma)$ returns a random sample from a zero-centred normal distribution with variance σ^2, $\Delta s_t = s_t - s_{t-1}$, and $s_{xy} = 0.3 s_{rr}$. In other



(a) Straight line track predictions.

(b) Circle track predictions.

Figure 8.11: Odometry predictions using GMapping's motion model ($s_{rr} = 0.1$, $s_{rt} = 0.2$, $s_{tr} = 0.1$ and $s_{tt} = 0.2$).

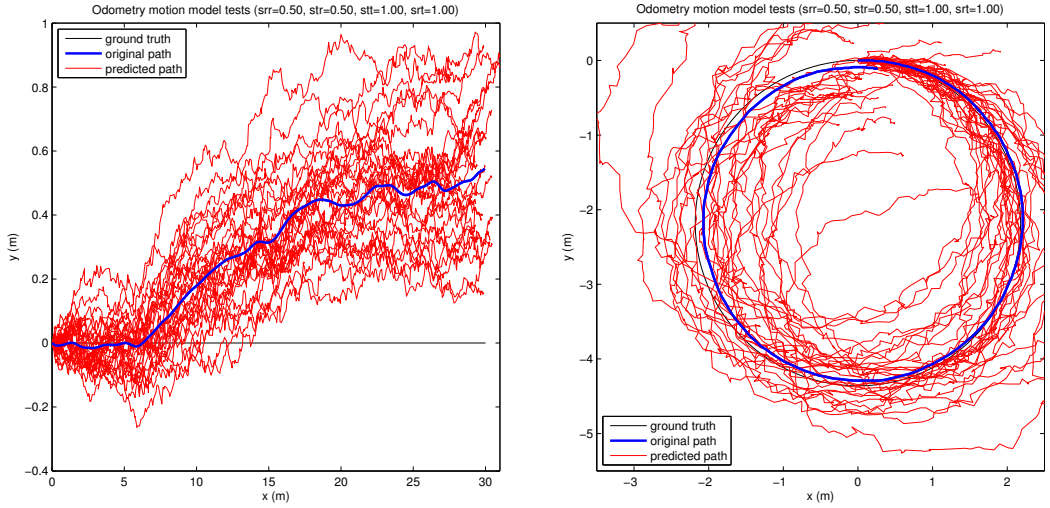
words, the four parameters represent: pure translational (s_{rr}), translational to rotational (s_{rt}), rotational to translational (s_{tr}), and pure rotational (s_{tt}) noise components.

In order to estimate the odometry model's noise parameters, analysis was conducted on actual odometry data from both a straight line test and a circle test (see figures 8.8 (b) and 8.9). The most deviant (with respect to the ground truth) test run from each of the two test types was used. It was reasoned that these runs represented experiments with the worst-case amount of random errors. Equation (8.1) was implemented in Matlab, and using the most deviant test run data, 30 predicted trajectories[1] were generated. Initially, the default GMapping odometry parameters from ROS [33] were used (i.e., $s_{rr} = 0.1$, $s_{rt} = 0.2$, $s_{tr} = 0.1$ and $s_{tt} = 0.2$). The results are shown in Figure 8.11.

As shown in Figure 8.11, the motion model produced trajectories that appear to be evenly distributed about the recorded path. This is as expected. However, particularly in the case of the straight line test (refer to Figure 8.11 (a)) the initial error parameters seem to underestimate the amount of noise present in the odometry. Figure 8.12 shows the odometry predictions when assuming significantly larger dead reckoning errors. Although the revised error parameters appear to better represent the uncertainty present in the straight line test, they yield poor performing predicted trajectories for the circle track test. Other combinations of error parameters also resulted in similar results. In summary, these results infer the following:

- The default GMapping error parameters in ROS ($s_{rr} = 0.1$, $s_{rt} = 0.2$, $s_{tr} = 0.1$ and $s_{tt} = 0.2$) are sufficient provided that the scan matching and particle resampling processes occur at least every 5 m when travelling in a straight line. Otherwise, the position estimate will drift significantly (in this case, up to $\pm$

[1]The default number of particles for GMapping is 30. Hence, during mapping nominally 30 possible trajectories will be maintained.



(a) Straight line track predictions.

(b) Circle track predictions.

Figure 8.12: Odometry predictions using GMapping's motion model ($s_{rr} = 0.5$, $s_{rt} = 1.0$, $s_{tr} = 0.5$ and $s_{tt} = 1.0$).

0.5 m over 30 m).

- The default error parameters are also sufficient for at least one revolution of a 2.18 m radius circle between particle re-sampling processes.
- Assuming a high number of dead reckoning errors leads to poor trajectory predictions, particularly in curved paths.

Note that if the odometry errors are underestimated, SLAM methods, in general diverge [60]. This is because, in the case of GMapping, insufficient error model parameters will yield a particle population that does not cover the whole space of possible robot poses. However, as shown in Figure 8.12 (b), if the error parameters are too high the particle population is too sparse. This inevitably leads to errors in the SLAM process, or requires maintaining a higher number of particles and thus more computation. Therefore, the motion model parameters need to be calibrated to a particular dead reckoning system and are an important consideration in tuning a SLAM system.

8.2.2 Simulation odometry

Prior to evaluating the navigation system in a virtual environment, it is useful to know how the errors in odometry have been simulated. Despite providing only a 2-D environment, Stage was used over Gazebo in simulation experiments. Reasons for this were discussed in Subsection 3.3.5. Equations (8.2) and (8.3) comprise Stage's motion model:

$$\Delta p = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix} = \begin{bmatrix} \dot{x}(1+\epsilon_x)\Delta t \\ \dot{y}(1+\epsilon_y)\Delta t \\ \dot{\theta}(1+\epsilon_\theta)\Delta t \end{bmatrix} , \tag{8.2}$$

$$s_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{t-1} + \begin{bmatrix} \Delta x \cos(\theta_{t-1} + \Delta\theta) + \Delta y \sin(\theta_{t-1} + \Delta\theta) \\ -\Delta y \cos(\theta_{t-1} + \Delta\theta) + \Delta x \sin(\theta_{t-1} + \Delta\theta) \\ \Delta\theta \end{bmatrix}, \quad (8.3)$$

where Δt is the update interval, while ϵ_x, ϵ_y and ϵ_θ are the dead reckoning error parameters. Note that in (8.2) and (8.3) Δx and Δy are with respect to the robot's local co-ordinate axes. From (8.2) it is evident that Stage's motion model is simplistic in that only systematic errors are represented. Thus the main modelled error is systematic drift in odometry, where the amount of drift added in each update is proportional to the change in pose. Rounding errors are also implicitly modelled. Also note that the change in pose is computed from velocities (as opposed to distance travelled in (7.2)). This is the case as ultimately, robots in Stage are velocity controlled.

The manner in which the dead reckoning model has been implemented in Stage is likely to have been a design decision. One explanation is so that the drift in odometry is repeatable for each simulation, provided that the conditions and user inputs remain the same. However, since (8.3) does not have any stochastic elements, it does not make sense to conduct experiments similar to the ones in Subsection 8.2.1 with the aim to 'tune' the ϵ error parameters to match what has been observed in the errors in the wheelchair's dead reckoning. Instead, sensible error parameters were used in the mapping experiments shown in Subsection 8.3.2.

8.3 Mapping

A key component of the navigation system introduced in this thesis is its mapping module. As outlined in Chapter 4, mapping an environment is a challenging task. Fortunately, several open-source SLAM libraries are available. GMapping [62] was identified to be the most suitable SLAM library, for reasons listed in Section 4.5. As discussed in Subsection 7.2.6, a wrapper was developed to integrate GMapping into the Player framework. The goal of this section is to discuss results obtained from mapping a real environment (Subsection 8.3.1), followed by comparisons with maps created with the Stage (See Subsection 3.3.5) simulator.

The environment used for the mapping tests was a office building on campus. It is envisaged that this area is likely to represent a typical operating environment. The ground truth map created from blueprints of this environment is shown in Figure 8.13.

8.3.1 Real environment

As discussed in Subsection 6.2.2, the Pronto M51 wheelchair was instrumented with three different laser scanners: a URG-04LX, a XV-11, and a virtual scanner derived from a XBox Kinect sensor. In the mapping experiments described in this subsection, only one rangefinder is used per experiment as input to the mapping module. The wheelchair was driven along the corridors of the test environment,

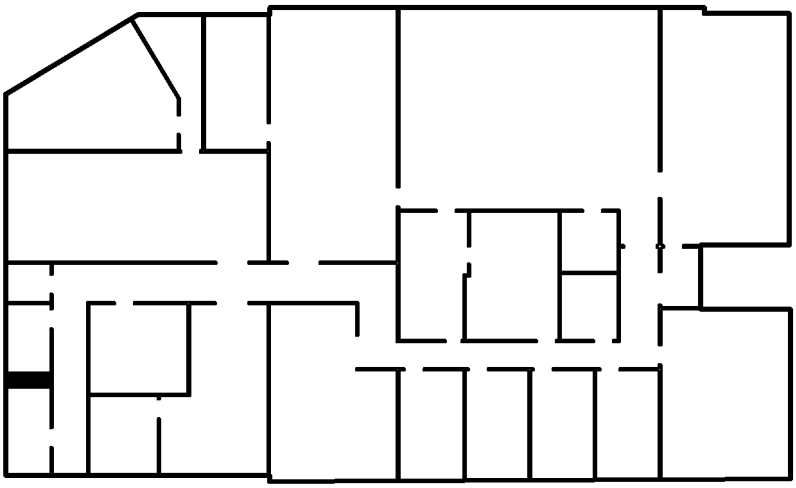


Figure 8.13: The ground truth map of the test environment. The dimension of the map is 36 m by 22 m.

as unfortunately most of the rooms were occupied (and no other suitable test area was available on the engineering campus). Mapping was achieved in real-time with the onboard laptop (HP 6730b with a T9300 2.5 GHz Core 2 processor and 4 GB of DDR2-800 RAM). Figure 8.14 shows the resulting map using the Hokuyo URG-04LX rangefinder.

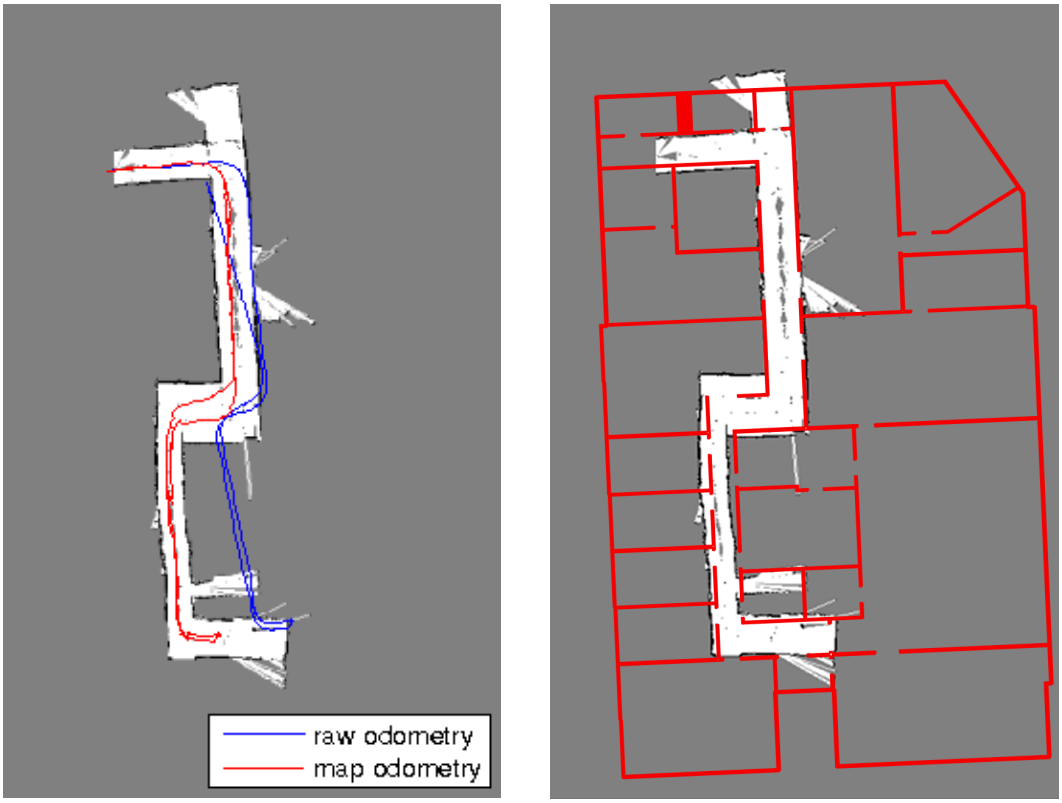
It can be seen from Figure 8.14 (b) that the reconstructed map resembles the ground truth map of the environment. Other observations made from Figure 8.14 include:

- The created map tends to exhibit unknown occupancy cells in the middle of the corridor. This is due to the interpretation of out of range readings, i.e., the `hokuyo_aist` driver reports zero for out of range readings. Although this is acceptable for the obstacle avoidance module, it will lead to problems with the path planning module as unknown cells are treated as obstacles.

- After turning around (at the bottom of the map) and travelling back along the corridor, the map has inconsistencies in its y-axis (aligned with the columns of the map image). Since this only happens in the y-axis — which is the axis in the experiment where the majority of the travelling is done — the error is likely to be associated from incorrect motion model parameters. Although the results found in Section 8.2 indicated that the default GMapping motion model parameters used in ROS were adequate, the tests that validated these were simplistic (only straight line and circle tests).

- The drift in dead reckoning becomes apparent in Figure 8.14 (a). Thus the importance of using techniques such as SLAM to correct for the unbounded errors associated with dead reckoning.

- There appears to be several laser scans with erroneous readings. It is suggested that these occur because of difficult surfaces in the environment on which the sensor can perform measurements.



(a) Odometry comparisons.

(b) Map comparison with floor plan.

Figure 8.14: Mapping an office corridor using the URG-04LX laser rangefinder. The robotic wheelchair started at the top left of the map, made its way down the corridor, turned around and returned to its start position. Note that the raw odometry was derived from wheel encoder data, while the map odometry was from the SLAM module's pose estimates.

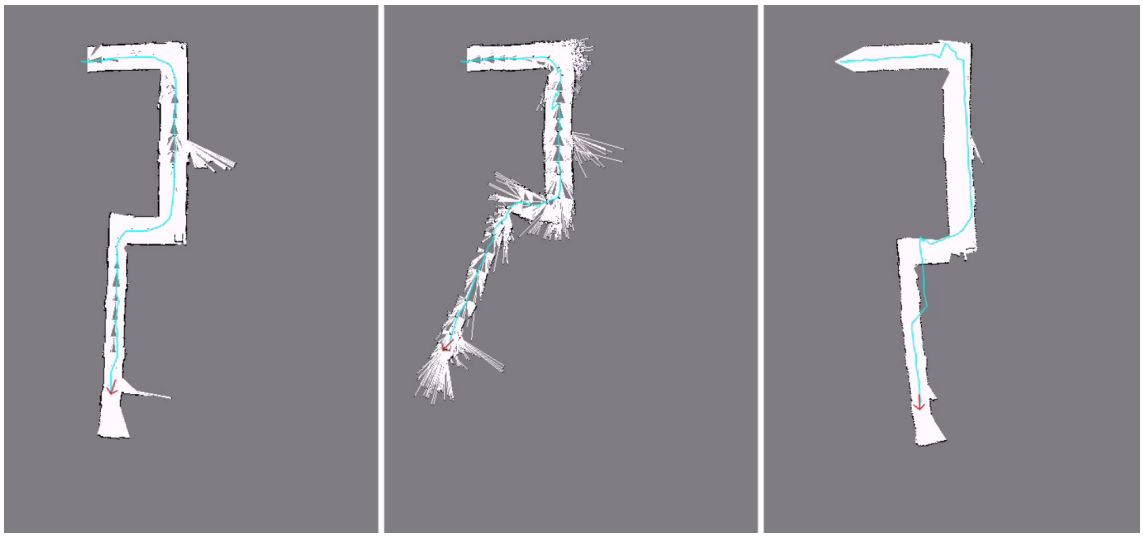


Figure 8.15: The maps created by the URG-04LX (left), XV-11 (middle), and Kinect derived (right) laser scanners. Note that the XV-11 was not working properly in this test.

Another test run along the corridor was performed, again logging the odometry poses and all three rangefinders scans to file (using Player's `writelog` driver). The data was played back through the navigation system three times, where each time a different rangefinder was used in the SLAM process. The resulting maps for each of the three scanners are shown in Figure 8.15.

The first observation is that the XV-11 resulted in GMapping creating a bad map of the environment. This was the case as during testing the XV-11 seemed to sporadically malfunction. Unfortunately, only one XV-11 was ordered and that there was not enough time left in the project to order another one. Otherwise, from the results obtained in Section 8.1 it was anticipated that a correctly working XV-11 scanner would yield a similar map to one made from a URG-04LX (provided that the wheelchair was not rotated abruptly). The cause of the XV-11's malfunction was not determined.

Another observation from Figure 8.15 is that the map made from Kinect 'laser' is quite good. Since the floor is always within view and range of the Kinect, the derived laser scan measurements will always have non-zero reading. Thus, unlike the other two scanners, the Kinect made map does not have unknown cell occupancies in the middle of the corridor. Also the Kinect generated map shows the distance to the closest frontier of obstacles. This can be observed in the middle right bends in the corridor in the map, where there was a desk in this corner. Note that an abrupt jump in the SLAM odometry reflects a change in index of the best particle in the mapping algorithm[1]. Note that the map from the Kinect appears smoother than that from the URG-04LX as the Kinect data has more smoothing (at the expense of dynamic response, see Subsection 8.1.2).

[1]The map trajectory shown in Figure 8.15 is from each best particle index at each given time, rather than the trajectory of the current best particle.

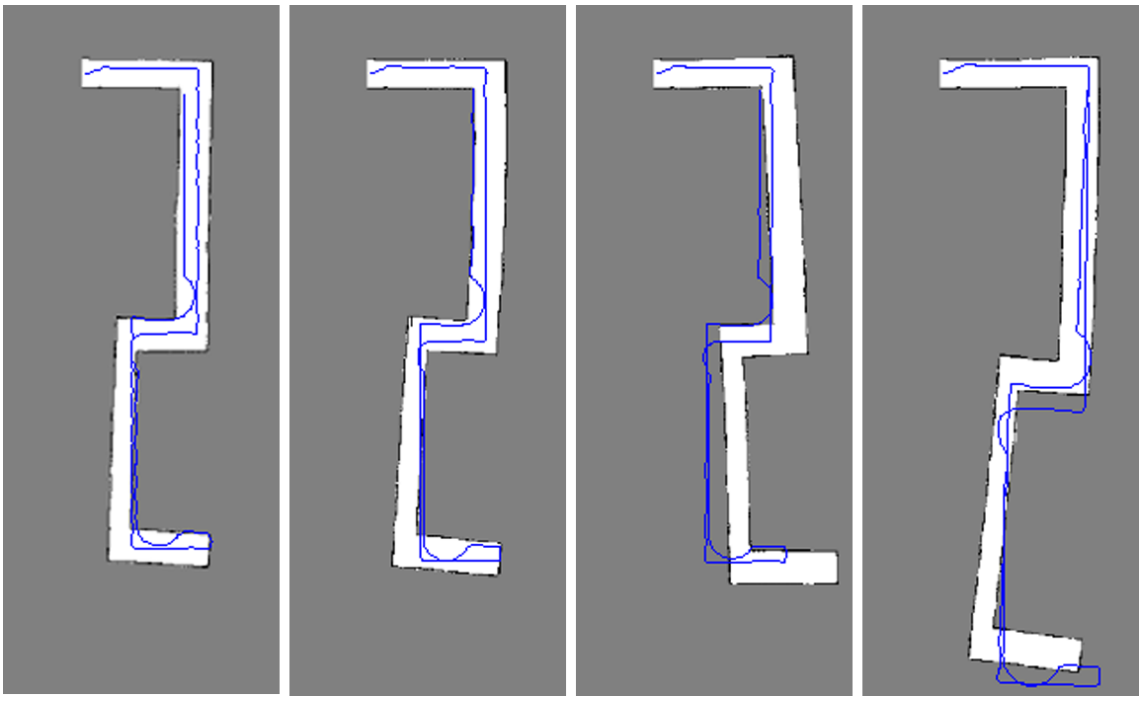


Figure 8.16: Mapping in a simulation environment with noiseless laser and varying levels of odometry error. Left to right: noiseless odometry, imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.05$), imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.1$), and imperfect odometry ($\epsilon_x = 0.3$, $\epsilon_y = 0.3$, and $\epsilon_\theta = 0.5$).

8.3.2 Simulation environment

A main benefit with designing this navigation system on top of Player is that, inherently, details of the robot have been abstracted from the top level application (i.e., the navigation system). That is, the navigation system does not need to be changed depending on whether it is controlling a physical or a simulated robot. The aim of this subsection is to explore the realness of mapping in Stage while using mostly its inbuilt sensor error models.

As discussed in Subsection 8.2.2, Stage is capable of simulating odometry drift. In order to compare the similarity of GMapping in simulation, the Stage environment was setup to resemble that the wheelchair was in (see Subsection 8.3.1). Four mapping experiments were carried out with different motion model parameters. In these experiments, a noiseless laser scanner with a 4.0 m maximum range, 1 ° angular resolution and 180 ° FOV was simulated. The results are shown in Figure 8.16. Note that to make all of the experiments comparable the velocity commands to the robot were exactly the same. This was achieved by commanding the robot via a simple deterministic wall-following algorithm. The robot started in the top left of the map, and followed its way around the left wall until it revisited the first corner of the corridor. Also note that the simulation results presented here are representative of typical cases.

Interestingly, as implied from Figure 8.16 increasing the error parameters of Stage's motion model yields larger maps. It is postulated that this is due to the

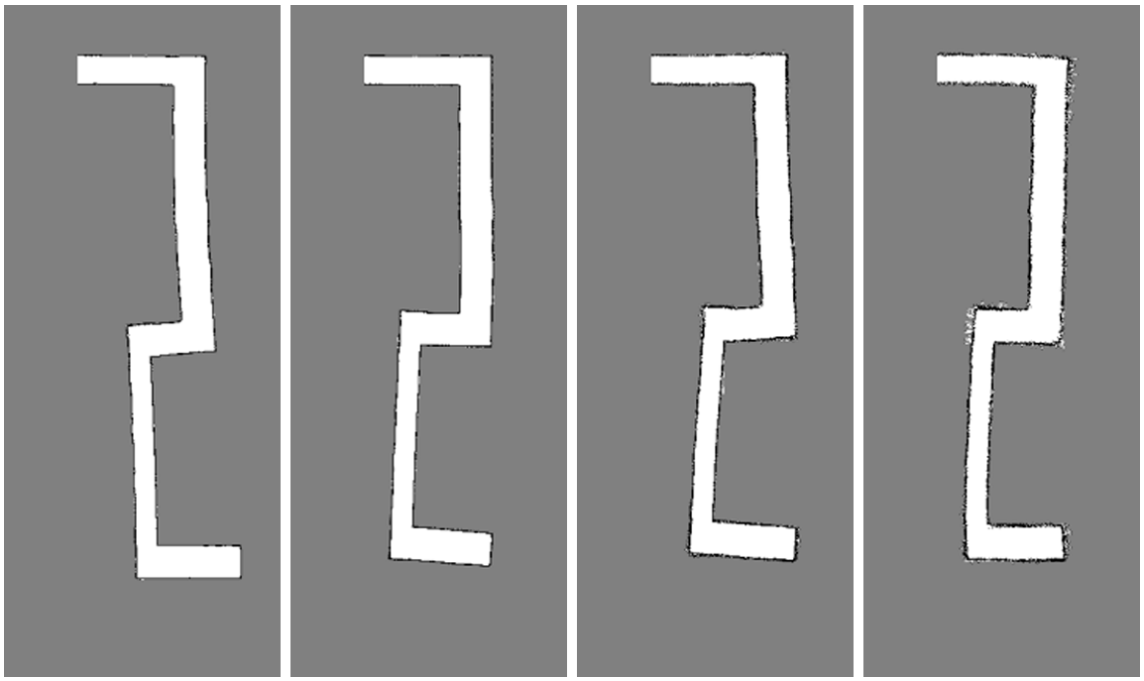


Figure 8.17: Mapping in a simulation environment with noiseless odometry and varying levels of additive Gaussian noise to laser scans. Left to right: noiseless, 1 %, 5 %, and 10 %.

range limits of the virtual laser scanner and the nature of the environment. In the test environment, large periods are spent navigating a straight and featureless corridor. Since most of the consecutive scans are similar, scan matching fails to estimate the change in pose and thus GMapping places more emphasis on odometry data. This problem is also likely to occur in a large open area where all static objects are out of the scanner's sensing range. Fortunately, it is less likely that walls in a real environment are as featureless as ones in a simple simulation environment.

Another aspect of any physical system that should be present in simulation tests is sensor noise. For 2-D laser scanners, it is common to make the assumption that the measurement error is Gaussian noise [84, 117, 118]. Thus for each simulated laser scan, Gaussian noise is added to each of the scan's range readings:

$$r_i' = r_i(1 + \epsilon x) \ , \tag{8.4}$$

where ϵ is the level of additive noise and x is a random variable with uniform distribution over the interval $[-1, 1]$. Using the logged data collected in each of the four experiments displayed in Figure 8.16, varying levels of white noise was added to the laser scans. Adding error to the readings just before it is processed by GMapping ensures that the robot's trajectory between the experiments remains the same. Figure 8.17 shows the resulting maps from various levels of additive noise to the laser scans and noiseless odometry. Meanwhile, figures 8.18 and 8.19 show the resulting maps with noisy laser data and imperfect odometry.

From figures 8.17, 8.18 and 8.19 it is apparent that GMapping still gives consistent maps despite adding up to 10 % of ratiometric white noise to the laser scans.

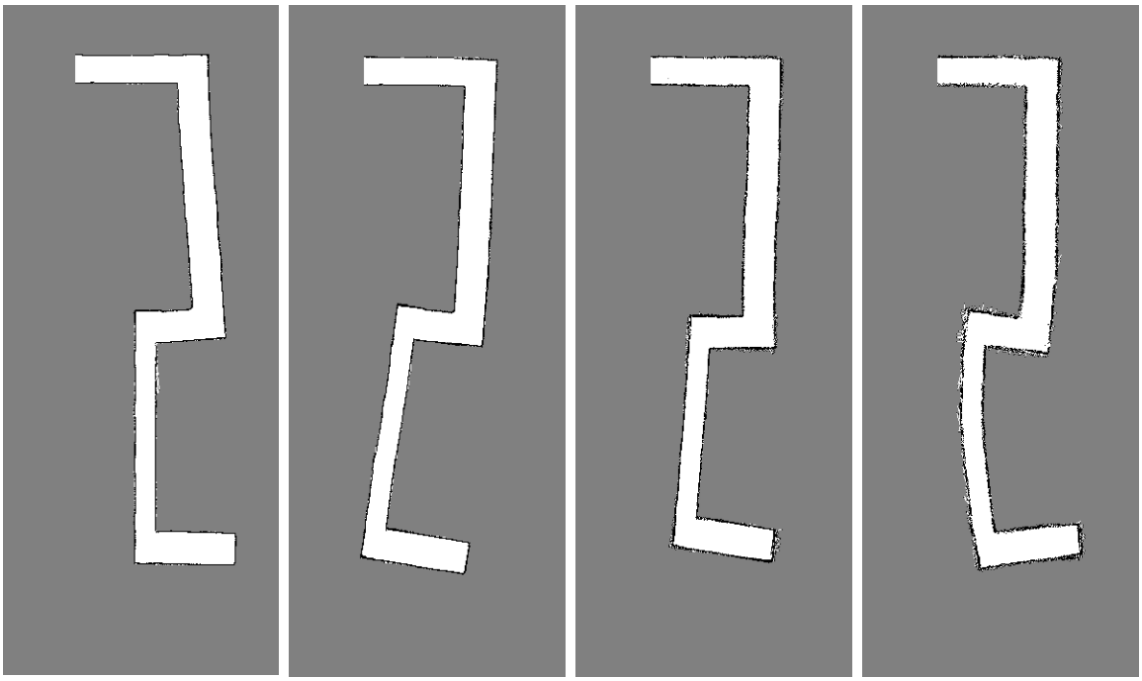


Figure 8.18: Mapping in a simulation environment with imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.05$) and varying levels of additive Gaussian noise to laser scans, left to right: noiseless, 1 %, 5 %, and 10 %.

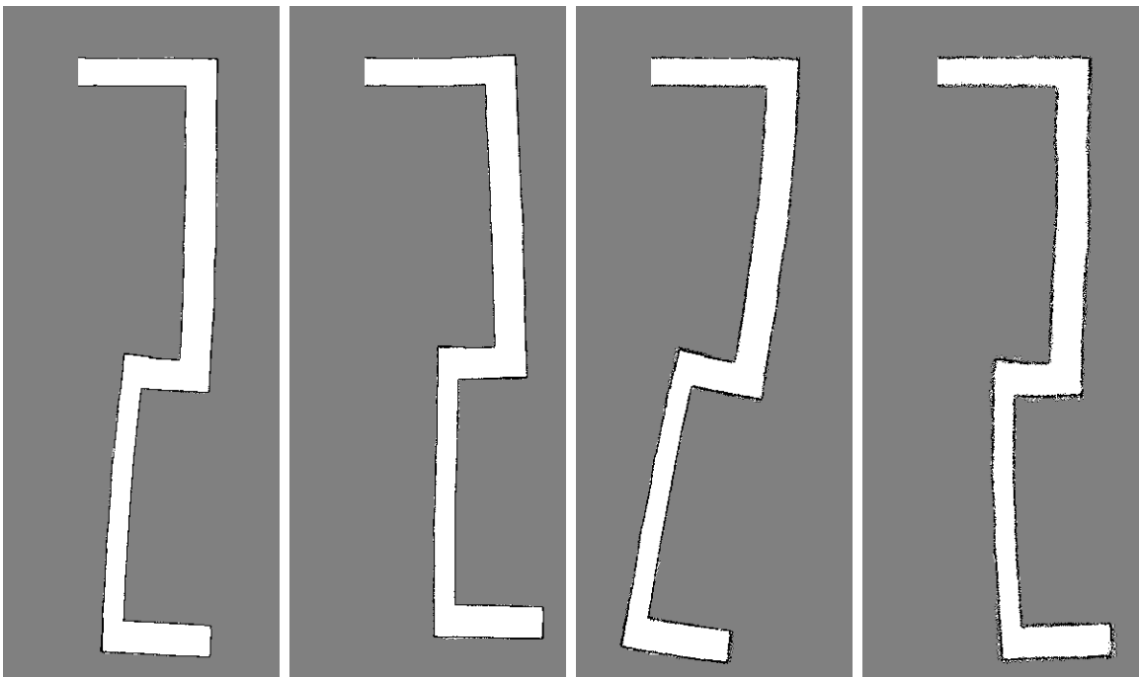


Figure 8.19: Mapping in a simulation environment with imperfect odometry ($\epsilon_x = 0.3$, $\epsilon_y = 0.3$, and $\epsilon_\theta = 0.5$) and varying levels of additive Gaussian noise to laser scans. Left to right: noiseless, 1 %, 5 %, and 10 %.

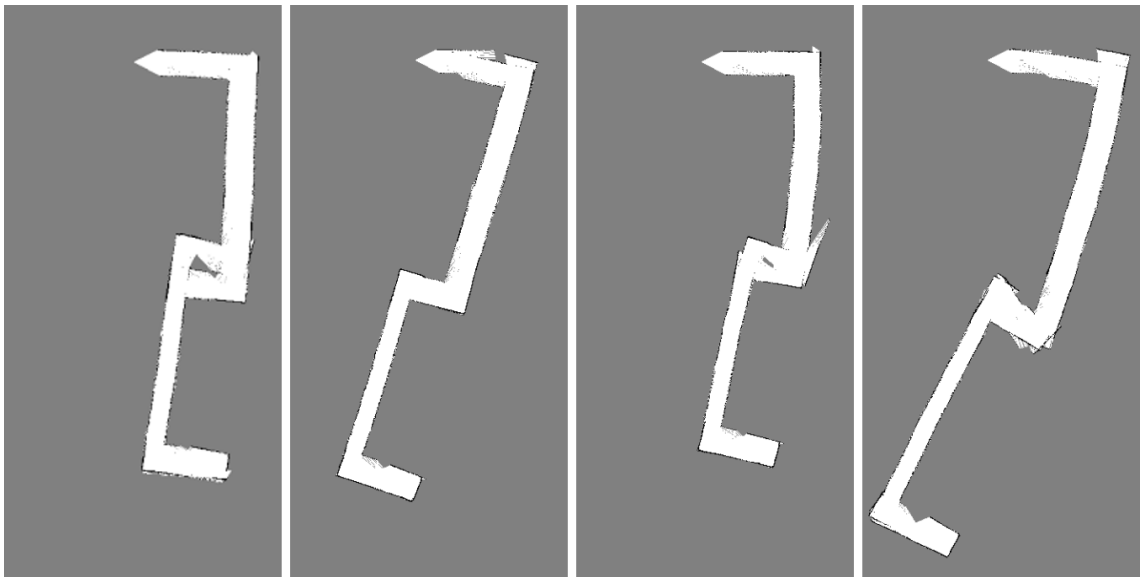


Figure 8.20: Mapping in a simulation environment with a noiseless 57° FOV laser and varying levels of odometry error. Left to right: noiseless odometry, imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.05$), imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.1$), and imperfect odometry ($\epsilon_x = 0.3$, $\epsilon_y = 0.3$, and $\epsilon_\theta = 0.5$).

However, it is believed that this is partly due to the lack of random error in the simulated odometry. In other research, GMapping was found to suffer from "a lack of accuracy when (laser measurement) noise is increased to 10 %" [84]. Note that in their experiments, the odometry was affected by zero-mean Gaussian distributed noise. Thus as future work, it is recommended to change Stage's motion model to better reflect errors associated with real odometry[1].

Another mapping test that was carried out in the simulation environment was to evaluate the influence of reducing a rangefinder's FOV. To simulate the Kinect 'laser', the laser scans were reduced to a field of view of 57° prior to being input to GMapping. Figure 8.20 shows the resulting maps with varying levels of odometry error and a noiseless 57° FOV laser, while Figure 8.21 includes 5 % additive ratiometric noise to the laser data.

It can be seen from Figure 8.20 that reducing the rangefinder's FOV has degraded the quality of the resulting map. The reduction of FOV and the lower quality maps produced by GMapping is likely to be the result of scan matching failure(s). This is most noticeable at the corners where each of the scans captures a smaller view of the environment. Thus the scan matching process has less 'features' to estimate the change in pose between the scans and so the accuracy of the change in pose reduces, leading to map inconsistencies. It is also interesting to observe that, with a reduced rangefinder FOV, GMapping tends to create more consistent maps when the laser rangefinder has measurement error. This somewhat counter-intuitive result can be explained as GMapping's probabilistic odometry and sensor models assume that the readings are corrupted by noise, and thus make predictions accordingly.

[1]Balaguer et al. [84] also notes that adding Gaussian noise to odometry is 'highly suboptimal' in that it does not reflect the incremental nature of dead reckoning.

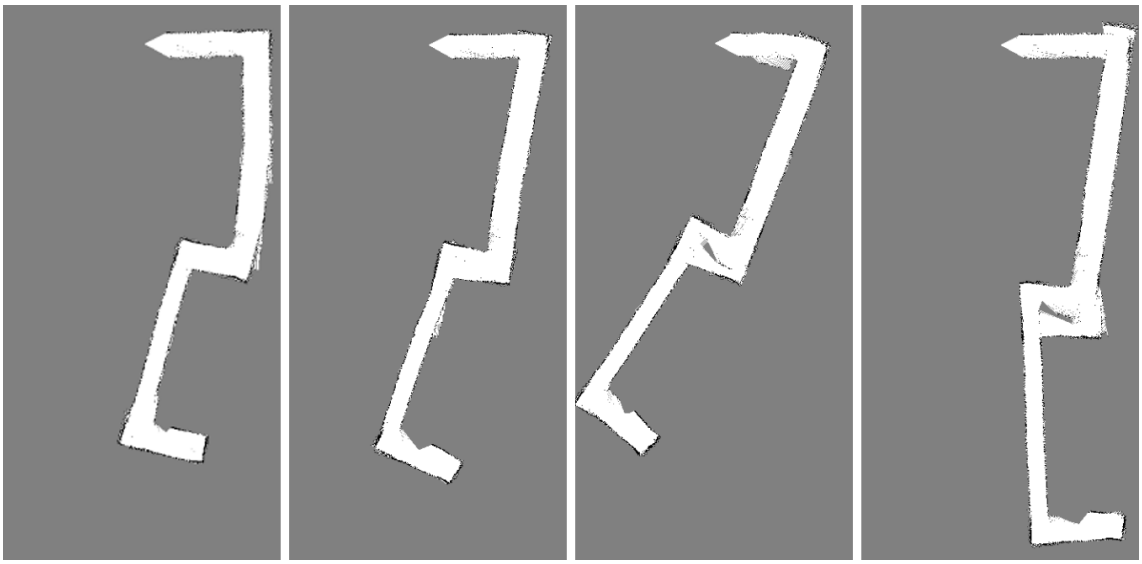


Figure 8.21: Mapping in a simulation environment with a 5 % additive noise 57 ° FOV laser and varying levels of odometry error. Left to right: noiseless odometry, imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.05$), imperfect odometry ($\epsilon_x = 0.03$, $\epsilon_y = 0.03$, and $\epsilon_\theta = 0.1$), and imperfect odometry ($\epsilon_x = 0.3$, $\epsilon_y = 0.3$, and $\epsilon_\theta = 0.5$).

8.4 Navigation evaluation

A crucial component of any autonomous/semi-autonomous robotic system is its ability to navigate its way through an environment. Generally, navigation is decomposed into local navigation (obstacle avoidance) and global navigation (path planning) problems. This section describes the obstacle avoidance and path planning modules used in the system. Due to the lack of a suitable test environment, the navigation experiments were carried out in simulation. Thus future work is required to test the obstacle avoidance and path planning routines on the wheelchair, once a suitable test area[1] becomes available.

Path planning is used for travelling between two points in an environment. This provides a high level of autonomous operation, useful for a smart wheelchair designed to handle the most severe cases of cogitative impaired users. To find a route to a destination, the module firstly requires a map of the environment. In this case, the map is provided by the SLAM module (see Section 8.3). Currently, the `wavefront` driver (described in Subsection 5.2.1) that came with Player is used for path planning.

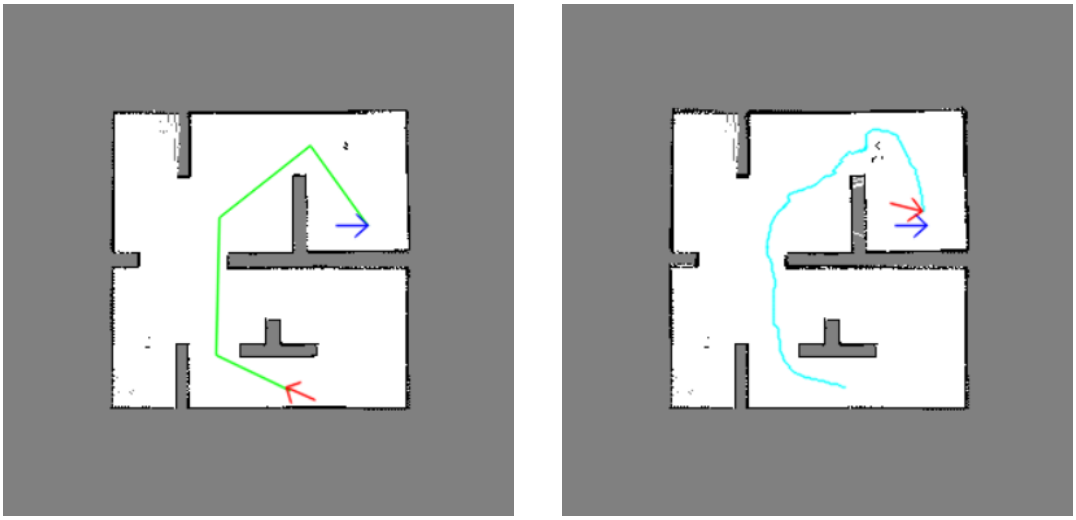
Once the environment had been adequately mapped, the user was able to select a goal within the map for the path planner find a route to. Provided that there was a viable path, `wavefront`, as expected, produced a set of waypoints that gave the shortest distance to the goal, as seen in figures 8.22 (a) and 8.23 (a). The robot was then directed to autonomously navigation towards the goal, where the actual

[1] A suitable test environment for preliminary navigation experiments would be a large open area that has partitioned by temporary walls.

path taken for the two paths is shown in figures 8.22 (b) and 8.23 (b) respectively. Note that in the current configuration, SLAM is performed while navigating. This is the case as revisiting places in the environment allows for loop closure which tends to increase the quality of the map. However, this requires the configuration space (C-space) used by the path planner to be updated each time a new path to a goal is requested. In simulation, this process was found to take in the order of 10 s on a modern laptop. This is not seen as an immediate issue: it is not expected that the user will request paths to goals often. Also the path planner runs in its own thread, and thus does not block other navigation tasks during extensive computation.

The obstacle avoidance algorithm used in this navigation system is VFH (see Subsection 5.1.2), a standard driver within the Player framework. Note that `vfh` provides obstacle avoidance if position control is used. This makes Player's `vfh` suitable for goal-orientated local navigation. However, if velocity control is used `vfh` simply passes on these commands to the underlying `position2d` interface. Thus to give the system obstacle avoidance when using velocity control, a routine was developed that reduced the speed of the robot as it approached obstacles and stopped the robot when it was close to a collision.

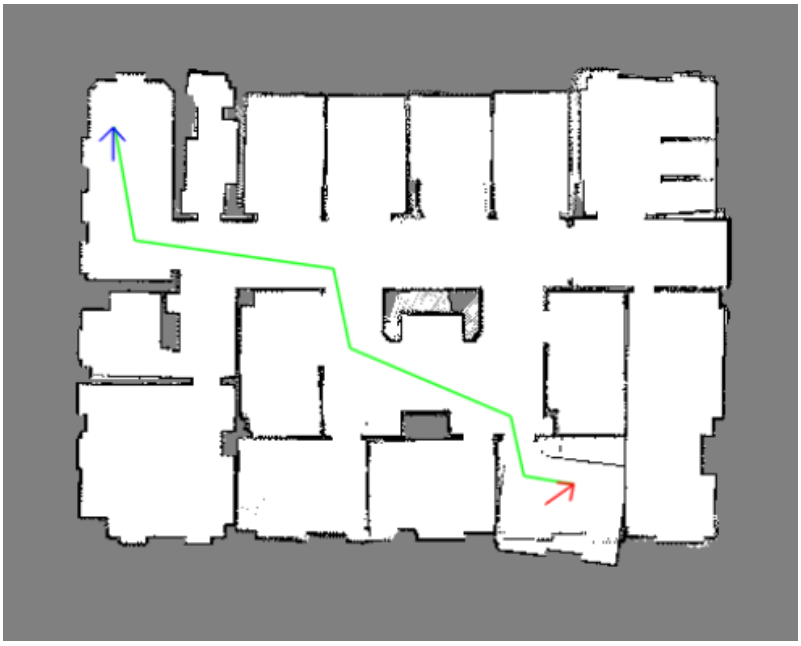
In order to evaluate Player's `vfh` driver, a simple environment was created in Stage. Since the functionality of `vfh` is only active when using position control, the obstacle avoidance driver was evaluated in conjunction with the path planning driver. As shown in Figure 8.22 (a), the path planner has successfully computed an efficient path to the goal. Before the robot autonomously navigates to the goal, a dynamic obstacle (another robot) was intentionally placed in the way of the path found to the goal. Figure 8.22 (b) demonstrates that `vfh` has indeed navigated around this obstacle. However, we also found that, in simulation, `vfh` struggled to pass through narrow passages (such as doorways), particularly when approaching from an acute angle. In such cases, the resulting behaviour from VFH was to make several attempts at aligning itself such that it could pass through the centre of the passageway. Future work is needed to overcome this issue, such as adding doorway recognition to the navigation system, then alter the waypoints to align the wheelchair correctly on its approach.



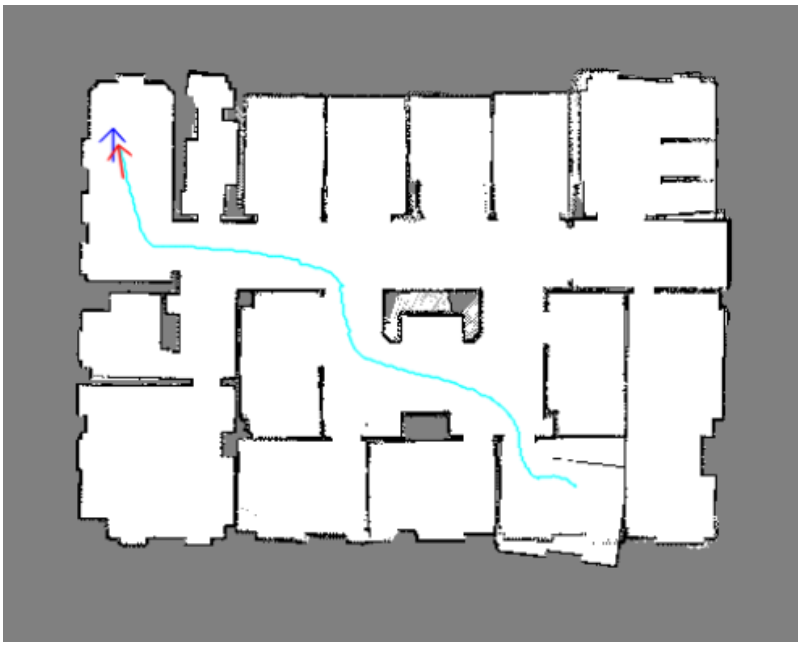
(a) Waypoints to goal.

(b) Actual path taken to goal.

Figure 8.22: Autonomous navigation with obstacle avoidance in a simulation environment. The red arrow is the current pose while the blue one is the goal pose.



(a) Waypoints to goal.



(b) Actual path taken to goal.

Figure 8.23: Autonomous navigation in a simulation environment. The red arrow is the current pose while the blue one is the goal pose.

Chapter 9

Conclusion

This thesis describes the development of a semi-autonomous wheelchair navigation system. The system has been demonstrated on a simulated robot in a virtual environment and also on an instrumented wheelchair in an office environment. The key features of the developed system include:

- Detection of static and dynamic obstacles to provide user-assisted navigation.
- Creation of a 2-D grid map of the environment and localisation of itself within the map. The map building and localisation process uses SLAM incrementally and in real-time.
- Ability to compute an efficient path to a user specified destination within the map. If a viable path to the destination is found by the path planner, the wheelchair is capable of autonomously navigating towards it while avoiding obstacles.
- Does not require structuring the environment.
- Provides a variety of methods to control the system, including: via a GUI, joystick, keypad, and autonomous navigation.
- Designed in a modular fashion that allows changes to a module in isolation. For instance, a new SLAM or path planning algorithm may be added to the system without propagating implementation-specific details to the other system components.

Another highlight of this system is that it utilises several popular and modern open-source software packages. This is important for the project's future development as it inherits the well thought out design philosophy used in these frameworks. The main framework that is used in this navigation system is Player, and thus the system implements a client-server model with multiple layers of abstraction. Since information is transmitted via sockets, it allows a variety of configurations of hardware deployment, including: onboard, distributed, and central computation.

In summary, the system introduced in this thesis provides a platform for the client, Dynamic Controls, on which to base their future wheelchair navigation research. Due to the systems engineering approach taken here, a project with a large

scope has been able to be delivered in a relatively short time frame. It also demonstrates the capability of the wealth of free open source software that is currently available.

9.1 Future work

A main limitation of this project was to develop a navigation system for indoor use only. This was done primarily to limit the scope of the project. Many of the sensors, techniques and algorithms used in this project have taken advantage of this, and are thus not suitable for outdoor use. For instance, the Kinect sensor is known not to work well outside as IR from sunlight tends to swamp its projector's IR pattern. On the other hand, the challenge of localisation is substantially easier as a GPS signal is often available outdoors in urban environments.

Another sensor-related issue that requires future work is sensing on various surfaces. The laser scanners used in experimentation have been shown to be accurate and precise. Unfortunately, they do not work well on all surfaces; none of the rangefinders evaluated here can find distances on transparent surfaces, e.g., glass or plastic panes. It is my recommendation that the wheelchair be instrumented with sonar rangefinders as well, and that any highly critical module — namely the the obstacle avoidance module — takes this information into account.

Due to the nature of smart wheelchairs (or any other medical device), rigorous testing must be carried before the system is commercialised. Unfortunately, no suitable indoor environment was available at the time for fully testing the prototype robotic wheelchair. Thus, this system requires further testing in the form of user trials, ideally in a hospital ward or an elderly care centre. Note that this thesis focused on a proof of concept rather creation of a product that could be immediately commercialised.

The system would also benefit from a more sophisticated dead reckoning module. It was found that in the tests the encoders gave accurate distance estimates but tended to yield inaccurate heading values. The heading estimate could be improved by fusing odometry information with a rate gyro or data from an IMU. Although odometry is corrected in the SLAM process, a better dead reckoning system would permit mapping with less particles (and thus lower computation demands). Improved dead reckoning would also allow the SLAM process to rely more on odometry when scan matching fails, i.e., mapping a large environment where all obstacles are out of the scanner's range. A more advanced model should also take into consideration other physical variables of pertaining to the wheelchair, such as uneven weight distribution on the wheels and varying tyre pressures (for chairs with pneumatic wheels).

Another task to be completed is to test the obstacle avoidance and path planning features of the system on the physical wheelchair. These components were not tested in a real workplace as a suitable test environment was not available at the time. However, it is my belief that provided that the map adequately represents the area and that the rangefinders have adequate filtering and are appropriate for the environmental conditions, the path planning and obstacle avoidance modules should

yield comparable performance to that obtained in a simulation environment.

It would also be useful to be able to add to a partially complete map of the environment. It is envisaged that this would require implementation of a sub-mapping technique on top of GMapping, and also the ability to set the current map of GMapping by cleverly writing to its internal variables and data structures. A feature map of 'GeoTagged' areas/objects of interest associated to the SLAM generated map would also be useful to the system, as it would allow autonomous navigation to places (as opposed to autonomous navigation to a destination in the map).

Experimentation into 3-D SLAM could also be conducted. Although 3-D SLAM is bound to be more computationally expensive and the extra dimension redundant in the case of a wheelchair, the extra dimension should make the data association task more robust (as landmarks are more likely to be unique, and thus reduces the change of map inconsistencies). Finally, future research that would be highly beneficial to the system would include user intention prediction. Information could be extracted from camera frames, laser scans, joystick history, etc, and interpreted via a Partially observable Markov decision process (POMDP) to provide assistance to specific navigation tasks. The work by Taha et al. [122] describes such work.

References

[1] R. C. Simpson. Smart wheelchairs: A literature review. *Journal of Rehabilitation Research and Development*, 42(4):423–438, 2005.

[2] L. Fehr, W. Langbein, and S. Skaar. Adequacy of power wheelchair control interfaces for persons with severe disabilities: a clinical survey. *Journal of Rehabilitation Research and Development*, 37(3):353–360, 2000.

[3] E. Trefler, S. G. Fitzgerald, D. A. Hobson, T. Bursick, and R. Joseph. Outcomes of wheelchair systems intervention with residents of long-term care facilities. *Assistive Technologies*, 16(1):18–27, 2004.

[4] R. C. Simpson. How many people would benefit from a smart wheelchair? *Journal of Rehabilitation Research and Development*, 45(1):53–72, 2008.

[5] R. C. Simpson, E. LoPresti, S. Hayashi, I. Nourbakhsh, and D. Miller. The smart wheelchair component system. *Journal of Rehabilitation Research and Development*, 41(3B):429–442, 2004.

[6] S. P. Levine, D. A. Bell, L. A. Jaros, R. C. Simpson, Y. Koren, and J. Borenstein. The NavChair assistive wheelchair navigation system. In *IEEE Transactions on Rehabilitation Engineering*, volume 7, pages 443–451, 1999.

[7] H. A. Yanco. Wheelesley: A robotic wheelchair system: Indoor navigation and user interface. *Assistive Technology and Artificial Intelligence, Applications in Robotics, User Interfaces and Natural Language Processing*, pages 256–268, 1998.

[8] Massachussets Institute of Technology. The MIT intelligent wheelchair project: Developing a voice-commandable robotic wheelchair. `rvsn.csail.mit.edu/wheelchair/`.

[9] G. Bourhis and M. Sahnoun. Assisted control mode for a smart wheelchair. In *IEEE International Conference on Rehabilitation Robotics*, pages 158–163, 2007.

[10] S. Vicente Diaz, C. Amaya Rodriguez, F. Diaz Del Rio, A. Civit Balcells, and D. Cagias Muniz. TetraNauta: a intelligent wheelchair controller for users with very severe mobility restrictions. In *IEEE International Conference on Control Applications*, volume 2, pages 778–783, 2002.

[11] H. Seki, S. Kobayashi, Y. Kamiya, M. Hikizu, and H. Nomura. Autonomous/semi-autonomous navigation system of a wheelchair by active

ultrasonic beacons. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1366–1371, 2000.

[12] W. Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Vienna University of Technology, Vienna, Austria, 2002.

[13] J. Crowley and Y. Demazeau. Principles and techniques for sensor data fusion. *Signal Processing*, 32(1-2):5–27, 1993.

[14] B. Kuipers. Building and evaluating an intelligent wheelchair, 2006.

[15] A. Murarka, J. Modayil, and B. Kuipers. Building local safety maps for a wheelchair robot using vision and lasers. In *The 3rd Canadian Conference on Computer and Robot Vision*, pages 25–33, 2006.

[16] Y. Kuno, N. Shimada, and Y. Shirai. Development of intelligent wheelchair system with face- and gaze-based interface. In *IEEE International Workshop on Robot and Human Interactive Communication*, pages 262–267, 2001.

[17] D. M. Brienza and J. Angelo. A force feedback joystick and control and control algorithm for wheelchair obstacle avoidance. In *Disability and Rehabilitation*, volume 18, pages 123–129, 1996.

[18] T. Felzer and R. Nordman. Alternative wheelchair control. In *IEEE International Symposium on Research on Assistive Technologies*, pages 67–74, 2007.

[19] N. I. Katevas, N. M. Sgouros, S. G. Tzafestas, G. Papakonstantinou, P. Beattie, J. M. Bishop, P. Tsanakas, and D. Koutsouris. The autonous mobile robot SENARIO: a sensor aided intelligent navigation system for powered wheelchairs. In *IEEE Robotics and Automation Magazine*, volume 4, pages 60–70, 1997.

[20] R. Simpson and S. Levine. Voice control of a powered wheelchair. In *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, volume 10, pages 122–125, 2002.

[21] F. Doshi and N. Roy. Efficient model learning for dialog management. In *IEEE International Conference on Human-robot interaction*, pages 65–72, 2007.

[22] Y. Kuno, N. Shimada, and Y. Shirai. Look where you're going [robotic wheelchair]. *IEEE Robotics and Automation Magazine*, 10(1):26–34, 2003.

[23] B. E. Swartz and E. S. Goldensohn. Timeline of the history of eeg and associated fields. *Electroencephalography and Clinical Neurophysiology*, 106(2): 173–176, 1998.

[24] RIKEN. Real-time control of wheelchairs with brain waves. 2009. `http://www.riken.jp/engn/r-world/info/release/press/2009/090629/index.html`.

[25] S. Hemachandra, T. Kollar, N. Roy, and S. Teller. Following and interpreting narrated guided tours. In *IEEE International Conference on Robotics and Automation*, pages 2574–2579, 2011.

[26] G. Bugmann, P. Robinson, and K. L. Koay. Stable encoding of robot paths using normalised radial basis networks: application to an autonomous wheelchair. In *International Journal of Vehicle Autonomous Systems*, volume 4, pages 239–249, 2006.

[27] H. N. Chow, Y. Xu, and S. K. Tso. Learning human navigational skill for smart wheelchair. In *International conference on Intelligent Robotics and Systems*, volume 1, pages 996–1001, 2002.

[28] R. Ryan, J. Battat, B. Charrow, J. Ledlie, D. Curtis, and J. Hicks. Organic indoor location discovery. In *The International Conference on Mobile Systems, Applications, and Services*, 2010.

[29] S. Dawson, B. Wellman, and M. Anderson. Using simulation to predict multi-robot performance on coverage tasks. In *IEEE International Conference on Intelligent Robotics and Systems*, pages 202–208, 2010.

[30] L. Meeden. Bridging the gap between simulations and reality with improved models of sensor noise. In *Proceedings of the Third Annual Genetic Programming Conference*, pages 824–831, 1998.

[31] S. Carpin, M. Lewis, J. Wang, S. Balarkirsky, and C. Scrapper. USARSim: a robot simulator for research and education. In *IEEE International Conference on Robotics and Automation*, pages 1400–1405, 2007.

[32] B. Gerkey, R. Vaughan, A. Howard, and N. Koenig. The Player Project, December 2010. `http://playerstage.sourceforge.net/`.

[33] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. IEEE International Conference on Robotics and Automation, 2009.

[34] J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2), 2007.

[35] Robocup rescue virtual robots competition. `http://www.robocuprescue.org/wiki/index.php?title=Virtualrobots`.

[36] M. Somby. Updated review of robotics software platforms, July 2008. `http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Updated-review-of-robotics-software-platforms/`.

[37] PSU Robotics. Player/Stage drivers. `http://psurobotics.org/wiki/index.php?title=Player/Stage_Drivers`.

[38] H. Durrante-Whyte and T. Bailey. Simultaneous localization and mapping: Part I. *IEEE Robotics and Automation Magazine*, 13(2):99–110, 2006.

[39] Elmar A. Ruckert. Simultaneous localisation and mapping for mobile robots with recent sensor technologies. Master's thesis, Graz University of Technology, Styria, Austria, 2009.

[40] T. Botterill. *Visual Navigation for Mobile Robots using the Bag-of-Words Algorithm*. PhD thesis, University of Canterbury, Christchurch, New Zealand, 2010.

[41] S. Thrun. Robotic mapping: A survey. In *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002.

[42] H.J.S Feder, J.J Leonard, and C.M. Smith. Adaptive mobile robot navigation and mapping. In *International Journal of Robotics Research*, volume 18, pages 650–668, 1999.

[43] T. Bailey. *Mobile Robot Localisation and Mapping in Extensive Outdoor Environments*. PhD thesis, The University of Sydney, Sydney, Australia, 2002.

[44] M.A. Fischler and R.C. Bolles. Random sample consensus: A paradigm for model tting with applications to image analysis and automated cartography. In *Communications of the ACM*, volume 24, pages 381–395, 1981.

[45] J. Neira and J. D. Tardos. Data association in stochastic mapping using the joint compatibility test. In *IEEE Transactions on Robotics and Automation*, volume 17, pages 890–897, 2001.

[46] J. Wang, G. Hu, S. Huang, and G. Dissanayake. 3D landmarks extraction from a range imager data for SLAM. In *Australasian Conference on Robotics and Automation*, 2009.

[47] J. Neira, J. Tardos, and J. Castellanos. Linear time vehicle relocation in SLAM. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 427–433, 2003.

[48] C. Le. A review of submapping SLAM techniques. Master's thesis, University of Giron, Catalonia, Spain, 2006.

[49] K. Granstrom, J. Callmer, F. Ramos, and J. Nieto. Learning to detect loop closure from range data. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 15–22, 2009.

[50] B. Williams, M. Cummins, J. Neira, P. Newman, I. Reid, and J. Tardos. A comparison of loop closing techniques in monocular SLAM. In *Robotics and Autonomous Systems*, volume 57, pages 1188–1197, 2009.

[51] S. Williams. *Efficient Solutions to Autonomous Mapping and Navigation Problems*. PhD thesis, The University of Sydney, Sydney, Australia, 2001.

[52] A. Davison, I. Reid, N. Molton, and O. Stasse. MonoSLAM: Real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007.

[53] J. Guivant, E.M. Nebot, and H.F. Durrant-Whyte. Simultaneous localization and map building using natural features in outdoor environments. In *International. Conference on Intelligent Autonomous Systems*, volume 1, pages 581–588, 2000.

[54] R. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME - Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[55] Z. Chen. Bayesian filtering: From kalman filters to particle filters, and beyond. *Adaptive Systems Lab, McMaster University. Hamilton, ON, Canada*, 18(7): 1–69, 2003.

[56] V. Tresp and R. Hofmann. Nonlinear time-series prediction with missing and noisy data. *Neural Computation*, 10(1):731–747, 1998.

[57] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Pres, Cambridge, MA, 2005.

[58] N. Gordon, D. Salmond, and A. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *IEE Proceedings F Radar and Signal Processing*, volume 140, pages 107–113, 1993.

[59] C. Stachniss, G. Grisetti, D. Hahnel, and W. Burgard. Improved rao-blackwellized mapping by adaptive sampling and active loop-closure. In *IEEE International Conference on Robotics and Automation*, pages 2432–2437, 2005.

[60] Enrique Fernandex Perdomo. Test and evaluation of the FastSLAM algorithm. Master's thesis, University of Las Palmas de Gran Canaria, Spain, 2009.

[61] N. Kwak, I. Kim, H. Lee, and B. Lee. Analysis of resampling process for the particle depletion problem in FastSLAM. In *IEEE International Symposium on Robot and Human interactive Communication*, pages 200–205, 2007.

[62] G. Grisetti, C. Stachniss, and W. Burgard. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. 23:34–46, 2007.

[63] M. Montermerlo. *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem With Unknown Data Association*. PhD thesis, Carnegie Mellon University, USA, 2003.

[64] Kevin Murphy. Bayesian map learning in dynamic environments. In *In Advances in Neural Information Processing Systems*, pages 1015–1021. MIT Press, 2000.

[65] A. Doucet, N. de Freitas, K. Murphy, and S. Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *16th Conference on Uncertainty in Artificial Intelligence*, pages 176–183, 2000.

[66] A. Eliazar. *DP-SLAM*. PhD thesis, Duke University, USA, 2003.

[67] A. Elfes. Using occupany grids for mobile robot perception and navigation. In *IEEE Computer*, volume 22, pages 46–57, 1989.

[68] A. Schultz, W. Adams, B. Yamauchi, and M. Jones. Unifying exploration, localization, navigation and planning through a common representation. In *IEEE International Conference on Robotics and Automation*, pages 2651–2658, 1999.

[69] J. Leonard and H. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. In *IEEE Transactions on Robotics and Automation*, volume 7, pages 376–382, 1991.

[70] A. Diosi, G. Taylor, and L. Kleeman. Interactive SLAM using laser and advanced sonar. In *IEEE International Conference on Robotics and Automation*, pages 1103–1108, 2005.

[71] I. Althoer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Journal of Discrete and Computational Geometry*, 9(1): 81–100, 1993.

[72] S. Thrun. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[73] M. Bosse, P. Newman, J. Leonard, M. Soika, W. Feiten, and S. Teller. An Atlas framework for scalable mapping. In *IEEE Conference on Robotics and Automation*, volume 2, pages 1899–1906, 2003.

[74] Sebastian Carreno. Msc. thesis a study on the Atlas framework for metric SLAM. Master's thesis, Heriot-Watt University, Edinburgh, 2009.

[75] J. Leonard and P. Newman. Consistent, convergent and constant-time SLAM. In *IEEE Joint Conference on Artificial Intelligence*, pages 1143–1150, 2003.

[76] J. Gutmann and K. Konolige. Incremental mapping of large cyclic environments. In *IEEE Conference on Computational Intelligence in Robotics and Automation*, pages 318–325, 1999.

[77] R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. 5(4):56–68, 1987.

[78] R. Brooks. Symbolic error analysis and robot planning. 1(4):29–68, 1982.

[79] T. Bailey, J. Nieto, J. Guivant, M. Stevens, and E. Nebot. Consistency of the EKF-SLAM algorithm. In *IEEE Conference on Intelligent Robots and Systems*, pages 3562–3568, 2006.

[80] J. Sasiadek, A. Monjazeb, and D. Necsulescu. Navigation of an autonomous mobile robot using EKF-SLAM and FastSLAM. In *Mediterranean Conference on Control and Automation*, pages 517–522, 2008.

[81] T. Bailey, J. Nieto, and E. Nebot. Consistency of the FastSLAM algorithm. In *IEEE Conference on Robotics and Automation*, pages 424–429, 2006.

[82] B. Siciliano and O. Khatib. *Springer handbook of robotics*. Springer, 2008.

[83] A. Eliazar and R. Parr. DP-SLAM 2.0. In *IEEE Conference on Robotics and Automation*, volume 2, pages 1314–1320, 2004.

[84] B. Balaguer, S. Carpin, and S. Balakirsky. Towards quantitative comparisons of robot algorithms: Experiences with SLAM in simulation and real world systems. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2007.

[85] O. Khatib. Real-time obstacle avoidance for robot manipulators and mobile robotics. *The International Journal of Robotics and Research*, 5(1):90–98, 1986.

[86] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *IEEE Transactions on Robotics and Automation*, volume 2, pages 1398–1404, 1991.

[87] E. Rimon and D.E. Koditschek. Exact robot navigation using artificial potential functions. In *IEEE Transactions on Robotics and Automation*, volume 8, pages 501–518, 1991.

[88] S. Petti. *Safe Navigation within Dynamic Environments: A Partial Motion Planning Approach*. PhD thesis, Mines ParisTech, France, 2007.

[89] J. Borenstein and Y. Koren. Histogram in-motion mapping for mobile robot obstacle avoidance. In *IEEE Transactions on Robotics and Automation*, volume 17, pages 535–539, 1991.

[90] I. Ulrich and J. Borenstein. VFH+: reliable obstacle avoidance for fast mobile robots. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1572–1577, 1998.

[91] I. Ulrich and J. Borenstein. VFH*: local obstacle avoidance with look-ahead verification. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 2505–2511, 2000.

[92] D. Fox, W. Burgard, and S. Thrun. Controlling synchro-drive robots with the dynamic window approach to collision avoidance. In *IEEE International Conference on Intelligent Robotics and Systems*, volume 3, pages 1280–1287, 1996.

[93] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 341–346, 1999.

[94] K. Rebai, O. Azouaoui, and N. Ouadah. Bi-steerable robot navigation using a modified dynamic window approach. In *International Conference on Advanced Robotics*, pages 1–6, 2009.

[95] K. Macek, I. Petrovic, and E. Ivanjko. An approach to motion planning of indoor mobile robots. In *IEEE International Conference on Industrial Technology*, volume 2, pages 969–973, 2003.

[96] J. Minguez and L. Montano. Nearness diagram navigation (ND): a new real time collision avoidance approach. In *IEEE International Conference on Intelligent Robots and Systems*, volume 3, pages 2094–2100, 2000.

[97] J. Minguez. *Robot Shape, Kinematics, and Dynamics in Sensor-Based Motion Planning*. PhD thesis, University of Zaragoza, Spain, 2002.

[98] J.W. Durham and F. Bullo. Smooth nearness-diagram navigation. In *IEEE International Conference on Intelligent Robots and Systems*, pages 690–695, 2008.

[99] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume 4, pages 100–107, 1968.

[100] L. Kiss, A.R. Varkonyi-Koczy, and P. Baranyi. Autonomous navigation in a known dynamic environment. In *IEEE International Conference on Fuzzy Systems*, pages 266–271, 2003.

[101] R. Galve. Laser-based obstacle avoidance for mobile robotics. Master's thesis, Technical University of Denmark, 2008.

[102] R. A. Jarvis. Collision-free trajectory planning using the distance transform. In *Mechanical Engineering Transactions of the Institute of Engineers*, volume 3, pages 187–191, 1985.

[103] E. Parra-Gonzalez and J. Ramirez-Torres. *Object Path Planner for the Box Pushing Problem.* Multi-Robot Systems, Trends and Development. InTech, 2011.

[104] S. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[105] E. Fabrizi and A. Saffiotti. Extracting topology-based maps from gridmaps. In *IEEE International Conference on Robotics and Automation*, pages 2972–2978, 2000.

[106] B. Gao, D. Xu, F. Zhang, and Y. Yao. Constructing visibility graph and planning optimal path for inspection of 2D workspace. In *IEEE International Conference on Intelligent Computing and Intelligent Systems*, volume 1, pages 693–698, 2009.

[107] T. Kwon and J. Song. Real-time building of a thinning-based topological map. *Intelligent Service Robotics*, 1(3):211–220, 2008.

[108] S. Thrun. Learning maps for indoor mobile robot navigation. *Artificial Intelligence*, 1997.

[109] S. Engelson and D. McDermott. Error correction in mobile robot map learning. In *IEEE International Conference on Robotics and Automation*, pages 2555–2560, 1992.

[110] S. Thrun, W. Burgard, and D. Fox. A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping. In *IEEE Conference on Robotics and Automation*, volume 1, pages 321–328, 2000.

[111] D. Fox. Adapting the sample size in particle filters through KLD-sampling. 22(12):985–1003, 2003.

[112] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1322–1328, 1999.

[113] L. Ronghua and H. Bingrong. Coevolution based adaptive monte carlo localization (CEAMCL). volume 1, pages 183–190, 2004.

[114] A. Rentschler, R. Cooper, S. Fitzgerald, M. Boninger, S. Guo, W. Ammer, M. Vitek, and D. Algood. Evaluation of selected electric-powered wheelchairs using the ANSI/RESNA standards. *Archives of Physical Medicine and Rehabilitation*, 85(4):611–619, 2004.

[115] K. Konolige, J. Augenbraun, N. Donaldson, C. Fiebig, and P. Shah. A low-cost laser distance sensor. In *IEEE International Conference on Robotics and Automation*, pages 3002–3008, 2008.

[116] J. Stowers, M. Hayes, and A. Bainbridge-Smith. Altitude control of a quadrotor helicopter using depth map from Microsoft Kinect sensor. In *IEEE International Conference on Mechatronics*, pages 358–362, 2011.

[117] Y. Okubo and J. Borenstein. Characterization of the Hokuyo URG-04LX laser rangefinder for mobile robot obstacle negotiation. In *SPIE – The International Society for Optical Engineering*, 2009.

[118] C. Park, D. Kim, B. You, and S. Oh. Characterization of the Hokuyo UBG-04LX-F01 2D laser rangefinder. In *IEEE International Symposium on Robot and Human Interactive Communication*, pages 385–390, 2010.

[119] A. Martinelli and R. Siegwart. Estimating the odometry error of a mobile robot during navigation. In *European Conference on Mobile Robotics*, pages 1–6, 2003.

[120] J. Borenstein, H. R. Everett, and L. Feng. *Where am I? Sensors and Methods for Mobile Robot Positioning*. University of Michigan, 1996.

[121] L. Kleeman. Odometry error covariance estimation for two wheel robot vehicles. Technical report, Department of Electrical and Computer Systems Engineering, Monash University, 1995.

[122] T. Taha, J.V. Miro, and G. Dissanayake. Wheelchair driver assistance and intention prediction using POMDPs. In *Intelligent Sensors, Sensor Networks and Information*, pages 449–454, 2007.