NTNU – Trondheim
Norwegian University of
Science and Technology

Navigation with Simultaneous Localization and Mapping

For Indoor Mobile Robot

Mikael Berg

Master of Science in Engineering Cybernetics
Submission date: June 2013
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Keywords: Simultaneous Localization and Mapping, Mobile Robots, Particle Filter, Kalman Filter, Path Planning, Guidance, Web Interface

Abstract

This thesis describes the development of an integrated solution for simultaneous localization and mapping (SLAM), path planning and path following for a four-wheel indoor robot. Running on the robot's on-board computer, the solution estimates maps and location of the robot in real-time.

The maps built are occupancy grid maps, which are suitable for path planning and for human inspection. The implemented path planning algorithm is successful in generating paths which avoid obstacles and can be followed accurately by automatic control using a line-of-sight approach.

Experiments are presented which demonstrate the software's ability to accurately build a map of many environments with no prior assumptions or preparations. Providing foundations for localization and navigation, the system represents both a proof of concept and a platform on which others can build further.

Sammendrag

Rapporten beskriver utviklingen av en integrert løsning for samtidig lokalisering og kartlegging (SLAM), ruteplanlegging og rutefølging for en innendørs robot med fire hjul. Løsningen kjører på robotens egen datamaskin, og estimerer kart og robotens plassering i sanntid.

Kartene består av celler i et rutenett, og er passende for ruteplanlegging og for menneskelig inspeksjon. Den implementerte ruteplanleggingsalgoritmen kan generere ruter som unngår hindringer og som kan følges nøyaktig av automatisk kontroll som bruker en line-of-sight-tilnærming.

Eksperimenter presenteres som demonstrerer programvarens evner til å bygge nøyaktige kart over miljø, med uten forhåndsantakelser eller preparasjoner. Systemet gir grunnlag for lokalisering og navigasjon, og representerer et “proof of concept” og en plattform for videre utvikling.

Problem Description

The Student should:

1. Decide on a set of sensors suitable for performing SLAM[1] on the robot in question.
   (a) The sensors should be suitable for the generation of maps.
   (b) The sensors should to some degree give possibilities of autonomous operation for the robot, such as avoiding and detect collissions.
2. Do a literature search and choose one or more SLAM strategies or algorithms suitable for on-line[2] mapping and navigation of a indoor flat environment. The algorithm should also be suitable for doing such calculations off-line. The algorithm should be able to maintain maps of size suitable for office buildings or similar buildings.
3. Develop the following solutions, given an on-board computer of high performance:
   (a) *Data gathering:* Implement a solution for passively gathering sensor data. The solution should be able to do this while the robot is controlled by some other means, and the data should be suitable for performing off-line SLAM.
   (b) *Manual control:* Data gathering while the robot is controlled manually by some solution the Student has developed.
   (c) *On-line SLAM with manual control:* Data gathering, map generation and estimation of position in *real-time*[3], where the robot is controlled by some solution the Student has developed.

[1]SLAM stands for *Simultaneous Localization and Mapping*, and refers to a process which generates maps of an environment which a robot has navigated, while simultaneously deciding the positions which the robot has traversed.

[2]On-line SLAM refers to a SLAM-algorithm which is capable of performing SLAM while the robot is driving, and could potentially be used for navigation. The opposite is off-line, which performs SLAM from sensor logs and similar.

[3]Real-time here means calculations are fast enough to facilitate decisions or navigation while driving.

4. The following solutions should also be appreciated if they are developed:
   (a) *On-line SLAM with automatic exploration:* Gathering of data, map generation and automatic navigation with the purpose of exploring the environment where the robot started, with the goal of generating a map.
   (b) *On-line SLAM with abstracted navigation:* Given an already available map, the user can through a user interface decide on a destination. The robot should plan a path and navigate to this destination. If the route is blocked, a new path should be generated if possible, and and the new path should be navigated.
5. Develop an interface for controlling the different modes in the previous points, inspect and/or download sensor and map data, display estimated position and other parameters. The interface should be available via other units than the robot itself, through a wireless network.

Limitations:

1. It is not required to avoid all kinds of collisions or erroneous navigation, also not in automatically navigating modes. The robot is followed by a person with access to an emergency stop mechanism. Cliffs, which exist with downward facing staircases, and collisions with tables or other low objects are situations which the solutions need not handle.

Summary and Conclusions

The start point of this thesis was the problem description, developed in cooperation with the Supervisor. Plans were presented for robotic hardware, which was to be built by another student [6]. Goals for the thesis were defined for facilitating autonomous operation of the robot, which included choosing and mounting a suitable set of sensors, implementing simultaneous localization and mapping (SLAM), path planning and guidance, along with a human interface.

Some previous work was considered from other students at the university. This includes work on SLAM for small LEGO robots in constructed mazes, and work on localization of a small competition robot on predefined maps. The former was most relevant, but proved unsuitable for this thesis because of a poor match of system architecture and application. The quality of the LEGO robot's maps were seen as not promising for larger environments. Importantly, they were *landmark* based maps, but this project was assumed to work better with *grid maps*, which are more suitable for path planning.

After research on what sensor types would be most suitable for the project, two sensor types were mounted on the robot hardware: a low-end LIDAR unit and two encoders attached to freely rotating wheels, one on each side of the robot. The LIDAR makes distance measurements up to a maximum range of 5.6 meters. The encoders provide odometric measurements which can facilitate *dead reckoning.* The sensors were chosen because of their popularity in literature, ease of use and flexibility for further work.

A decision was made to implement a system from scratch in the programming language *Go*, which had proven successful for the competition robot previously mentioned.

The field of SLAM has been active in later years, and many algorithms exist distributed over several approaches. Much research was done in order to find a suitable algorithm to cover the points of the problem description and the visions of the project for the future. Most of the literature is

divided by the kinds of maps the approaches produce, prominently *landmark* based maps and *grid maps*. A decision was made to implement a grid map algorithm because they better facilitate path planning.

Some grid map algorithms maintain several maps in parallel through a *particle filter*, a multi-hypothesis representation. A downside with the approach is the large consumption of RAM. An approach was considered which lessens the RAM use of related maps, but the structure was seen as complicated and existing implementations were hard to follow. Although multi-hypothesis mapping has some advantages, especially in situations where the trajectory of the robot closes a loop in the environment, a decision was made to focus on single-hypothesis SLAM.

A minimal algorithm known as *TinySLAM* was first implemented for the project. However, because of poor initial results, further work on the algorithm was discontinued in favor of *Hector SLAM*. The abstracted functioning of the two algorithms is similar: they are both based on matching subsequent range scans to each other.

The Hector SLAM algorithm performs scan matching by a Gauss-Newton based algorithm inspired by work in computer vision. Bilinear filtering of the map and computation of approximated gradients allows scan matching to be performed in a computationally efficient manner. In order to reduce the problem of local minimas in the scan matching optimization scheme, maps are simultaneously produced at different resolutions, forming a structure similar to image pyramids.

Odometric measurements was incorporated in the SLAM algorithm through an extended Kalman filter. The filter considers both odometry and scan matching position estimates in order to provide an estimated position for the scan matching procedure.

A software was implemented around the SLAM algorithm, controlling sensors, motor control, path planning and higher-level control. The software provides a web interface with an API. The web interface is implemented with a server side built in the software, and serves HTML web pages and static files. The API provides data reflecting the state of the program and data such as estimated position of the robot and maps.

The sensor module can log sensor data and play them back in real-time. This represents a form of simulation, allowing testing to be performed at a desktop computer without robot hardware. Repeatability is another advantage. The rest of the software is agnostic to whether sensor data comes from a log or currently running sensors.

A map storage module provides means of storing obtained maps for re-use.

Path planning was implemented using the well-known A* algorithm, based on simplified instantaneous maps from the SLAM module. Measures were taken in order to avoid planning paths through known obstacles or too close to them. The algorithm is capable of planning paths through areas of unknown state, but punishes this behavior according to a configurable parameter. Paths produced by the A* algorithm are inherently non-smooth and can contain zig-zag patterns, but the effect is lessened through path smoothing by an algorithm with configurable parameters.

The motor module provides means of controlling the robot manually, through the web interface. It also provides guidance, making the robot follow planned paths. A collision detection module is used to stop the robot when obstacles are detected in a sector in front of the robot, whenever the robot is automatically following paths. When such stops occur, the robot backs in order to escape the obstacle, before planning a new path to the same goal location based on the currently available map. The scheme can be used for automatically mapping an environment.

Several experiments were conducted in order to demonstrate abilities and short-comings with the implementation and sensors. The SLAM-algorithm can run on-line and in real-time, which is used for guidance and can provide assistance if the robot is controlled manually, by this software or via other software. Results were shown demonstrating the accuracy of the odometry and its impact on the accuracies of maps when combined with LIDAR data and the SLAM-algorithm.

The software was shown to accurately localize the robot and update maps accordingly when using a previously obtained map or, equivalently, when traversing areas of a map which has previously been mapped. The path planning and guidance abilities of the software is able to generate and follow efficient paths.

In unknown environments, path planning and guidance can be used in order to automatically explore and build maps. By selecting a goal position which the robot cannot reach, the software will exhaustively explore the environment until it can be certain that the goal position is unreachable.

The SLAM process was demonstrated to have problems in areas where the LIDAR measurements and the current scan matching algorithm cannot alone produce accurate position estimates.

Closing loops of the environment is a problem recognized to be hard in literature. Experiments show that the current software-hardware combination is able to close small loops. Larger loops cannot reliably be closed, shown by an experiment with a loop of 125 meters. Multi-hypothesis mapping might help in such situations. Alternatively, a LIDAR units with longer range could significantly enhance the performance.

The robot lacks sensory equipment to be autonomous in all environments. Downward staircases, curbs and high doorsteps are among obstacles which the robot cannot see. However, the robot shows autonomous abilities in previously mapped areas.

Contents

Preface

Mobile robots are becoming more popular, both in industrial and domestic situations. Cleaning floors, transporting persons and even people – the robots both relieve us for monotonous tasks and increase our safety, exemplified with the *Neato* range of robotic vacuum cleaners and the *Google Autonomous Car* project. The latter has completed over 500 000 km without accidents on unprepared, real roads.

Automation and remote operation can make industry more effective, and the oil platforms are good examples. If more operations could be done remotely, from land, fewer people would have to work off-shore, far away from home and exposed to risks of weather and accidents.

A central need for mobile robots and their autonomy is their ability to know their environment, where they are and how they can move from one place to another. Having a map, being able to localize within the map, being able to plan paths and traverse them are abilities which can serve as foundations for many applications.

Obtaining a map is thus a central question. The most labor efficient and carefree solution is to let the robot replicate human behavior, making maps as we go. The field known as SLAM, simultaneous localization and mapping, studies how this can be achieved.

This thesis came to being on these thoughts. A prototype mobile robot was to be built, with a robotic arm capable of manipulating its environment, sensors, motors and software helping with navigation.

An operator controlling the prototype would have several means of control. Web cameras could be used for assistance when operating the arm remotely.

By automatically generating maps and showing the robot's position in these maps, remote operators would gain more certainty in their navigation. Perhaps the operator could be completely freed of the monotonous task of navigation altogether? This became the vision of this thesis.

Answering the questions demanded research in the field of SLAM. Recent popularity has led to many approaches to the problem, each with their own advantages and limitations. The thesis describes an implementation of a robotic software from scratch, able to serve as a foundation the mobility part of the prototype.

The work was hard but rewarding. Being given the time and opportunity to implement such a large system represented a project of large freedoms, with room for creativity. The amount of freedom also posed challenges of defining scope and goals, particularly since the platform opens up so many ideas for development of features: many ideas had to be set aside for further work.

Although not explicitly mentioned in the problem description, a large concern became making sure the work provides later students, projects and theses based on this system with a good “cornerstone”. The author wanted the system to be well documented and easy to understand and develop further. Grid map based SLAM is to the author’s knowledge a novel field for NTNU, which could be a point of expansion.

When conducting experiments in public areas, many students asked about the project, what abilities the robot had, how the SLAM process worked and were interested in working on the project themselves, for their pre-project or master thesis.

This shows the project has large interest and potential among students. Should the institute choose to develop this project further, many great accomplishments can be made.

Declaration of Text Independence

For his master thesis pre-project, the author worked with localization for the NTNU-Eurobot team’s robot and code base. The project has minor similarities with the work of this thesis.

The pre-project included use of a LIDAR unit, which was also relevant for this thesis. However, the advantage of the previous work is limited, as drivers are provided from the manufacturer.

The work of the pre-project made use of *particle filters* for localization, which bears some similarities with the *Rao-Blackwellized* particle filters on which some SLAM algorithms are based. However, the inner workings are largely different. Most prominently, the pre-project used static maps and did not perform SLAM.

Material from the pre-project report is only used in this thesis where cited.

Distribution of Workload

Much of the work in relation to this thesis was done in research. SLAM is a field of continuous research, and many algorithms and approaches exist. A high number of articles of the field is published in later years; a search for “simultaneous localization and mapping” of Google Scholar over 5000 results just from the last five months. A few textbooks exist describing individual algorithms in detail, but few if any books give an overview of the field. *Probabilistic Robotics* [59] covers many landmark based approaches, but the field has moved past much of its scope. A recent book by *Fernández-Madrigal et al.* [22] shows promise from its cover text, but was not available to the author during thesis work. Developing such an overview was therefore time consuming.

Much time also went to implementation of the system. Although efforts were made in order to reuse available code, the system was built from the ground up. Efforts were made in order to allow the system to be modular and expandable, further drawing on time.

The robot hardware was not ready to be tested before the mid-April, complicating initial testing of the software. Some LIDAR sensor logs were acquired by mounting the LIDAR to a trolley and connecting it to a laptop computer running the software.

Acknowledgments

I would like to thank Professor Tor Onshus, my supervisor, who trusted me to take on this problem and offered insights and project guiding during the thesis work. Thanks also to Mr. Stefan Kohlbrecher and his team at Technische Universität Darmstadt, who developed the Hector SLAM algorithm under an open license, and who responded to my bug report. Thanks to the developers of *Go*, who helped me get my code to compile, and who quickly and helpfully responded to my bug reports.

Thanks to Petter Aspunvik, who built the robot hardware which made this thesis possible. Through good cooperation, he went to great lengths to facilitate the needs of my work, and helped set up experiments.

Thanks to my fellow students at the office Anders Jordtveit Mørk, Torstein Thode Kristoffersen, Håvard Håkon Raaen, Henrik Emil Wold and Tone Ljones, with which I had many technical discussions driving the development further, and who generally made many long days not just survivable, but also humorous.

Mikael Berg
Trondheim, June 2013.

Chapter 1

Introduction

This chapter will outline the project and the motivations for the problem description on which this work is based. It introduces the hardware which was available during work with this thesis. We also outline some related projects at NTNU, and give a short introduction to the SLAM problem, which is central to this report.

1.1 About the Project

This master thesis is part of a project at the Department for Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). The vision of the project is to prototype a robot suitable for automatic or semi-automatic maintenance and repair in industrial and other situations. One possible application is use on oil platforms in the North Sea, where it's desirable to reduce use of on-site staff and have higher degree of automation. This involves the control of a robotic arm mounted on the robot as well as navigation of the robot.

The robot's arm can be controlled manually from a computer with a joystick. This setup can also be used to control the robot's wheels through motors, allowing the operator to see where the robot is going using cameras mounted on the robot. The cameras are also used for remote operation of the robotic arm.

The project was started in the last months of 2012, with a student planning the robot's hardware setup. Although he would focus on manual control of the robot and control of the robot arm, care was taken to ensure compatibility with automatic control schemes.

1.2 Motivation for Mapping and Guidance

Automatic mapping and navigation can be advantageous for the NTNU project and similar robots, for several reasons.

If the robot has a long way to drive in order to complete a task, it is beneficial to have the robot navigate to the destination autonomously. This frees time, and allows the operator to work more efficiently. One can also imagine schemes where the robot could follow a timed schedule, ordering it to report at different places at given points in time, allowing several operators to share the robot more efficiently.

In several applications, there can also be a significant latency between the manual commands given by a remote operator and the movements of the robot. If the robot is controlled over the Internet, latencies can be high, potentially making it cumbersome to control the robot accurately manually.

Even with manual navigation, it is convenient to have the robot construct maps of the environment and display its position in these maps, for assistance. Manual mapping would be very time consuming and we would risk having to update the maps when the environment changes, i.e. when furniture is moved, hallways are closed due to maintenance work or even if doors are closed.

With the need for real-time map generation, localization must be performed and can be seen both as a byproduct and a prerequisite, as we shall later see. Localization by itself is also handy for manual navigation, and a requirement for automatic navigation.

A robot which generates its own maps, is able to plan suitable paths and follow them to a goal location was seen as highly desirable for the project.

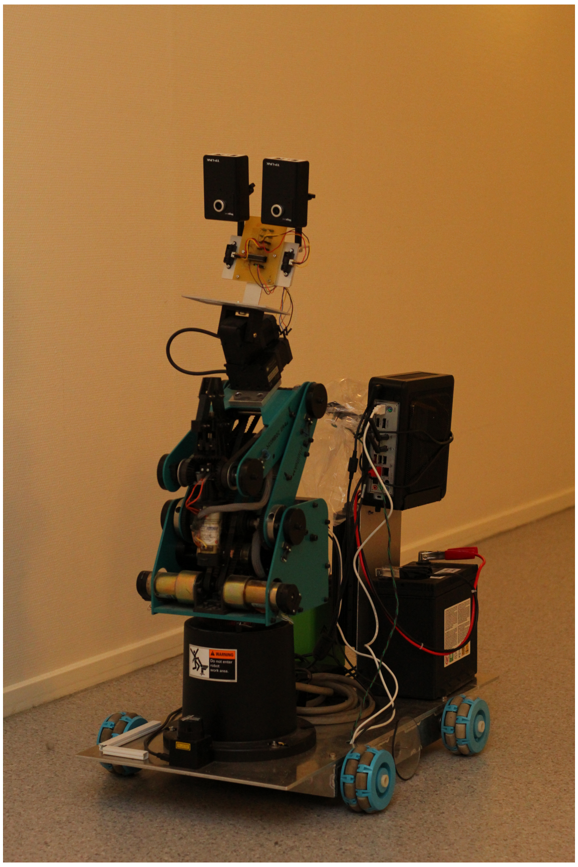


Figure 1.1: The full robot hardware, as used during the experiments of this thesis.

1.3 Robot Hardware

The robot's hardware was planned, built and tested during the first half of 2013, simultaneously with the writing of this thesis and the development of the software described in it. A brief description of the hardware will be given here, while a more comprehensive one is given in [6].

The robot base consisted of a steel chassis, providing a plate of approximately 80×35 cm on which equipment could be placed. A picture as per the end of the work of this thesis can be seen in Figure 1.1.

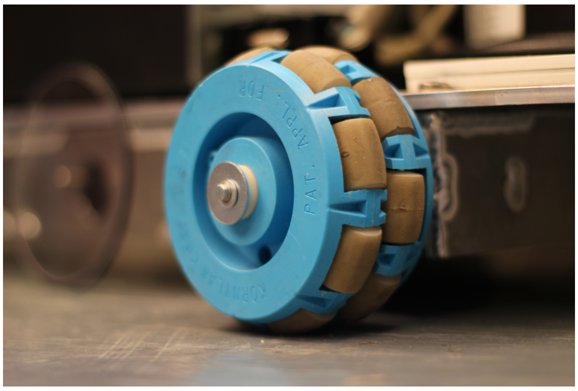


Figure 1.2: One of the omniwheels mounted on the robot. The "rollers" can roll perpendicularly to the rolling direction of the wheel.

1.3.1 Wheels and Driving Capabilities

Four motors were attached to the chassis, one for each of four wheels. The wheels were mounted in a rectangle.

The wheels were *omni-directional*, as shown in Figure 1.2. Having smaller wheels perpendicular to the rolling direction allowed them to slide laterally. This eliminated the need for a complicated car-like *Ackermann*-type steering mechanism. Instead, turning was achieved by letting the left hand side wheels rotate with a different velocity from the right hand side wheels. This gives a rotating motion around the approximate geographical center of the wheels, when the wheels are mounted in a rectangle.

When the two wheels on each of the sides always rotate as a pair, i.e. with the same velocity, this setup gives equations of motion approximately equal to that of a *differential wheeled* robot. These equations can be found in [50]. The approach is also called *skid steer drive*. During the months of work, a motor controller card for controlling the motors from a PC was added.

The motor controller card, which the on-board computer communicates with, was set up such that the left pair and the right pair of wheels were always given the same control signals.

1.3.2 Equipment

On the 0.3 m^2 robot base plate, the following equipment is fitted or shuld be fitted at a later time:

- A robotic arm and a controller box.
- Two web cameras.
- A 12 V car battery connected to a inverter, offering 240 V AC power.
- A lithium battery providing power for the motors.
- A computer equipped with a powerful *i5* CPU from *Intel Corporation* and 8 GB RAM, running *Windows 7*.

Additionally, a range sensor and odometric sensors were added, as described in Chapter 3.

1.4 The SLAM Problem

Simultaneous localization and mapping, or *SLAM*, is a process very central to this thesis. The name refers to the problem of creating and updating a map, while also maintaining the location of a robot in that map. In other words, *to learn a map of the robot's environments* using the robot's sensors.

For autonomous mobile robots, learning maps is often essential. Being able to automatically navigate in an environment is dependent on having a map, and manually creating this map is often a hard and labor intensive effort. Maintaining can prove costly enough to render the robot unusable.

Equipping the robot with sensors and software enabling it to solve this task by itself can be of great importance to the success of the robot system.

Autonomous mobile robots also need to *localize* them selves in their environment. Some sensor arrays could provide a full state estimate, such as an overhead camera combined with computer vision software. This solution is used primarily when the environment restricted to a small surface, such as in the *Micro Robot World Cup Soccer Tournament* (MiroSot) [63]. In such applications, the full robot position can be computed directly.

However, when the environment grows, or the environment should not be changed in order for the robot to obtain a position estimate, such sensor arrays become infeasible. For a robot exploring unprepared indoor environments, its location most often has to be computed from several sensor scans, and is dependent on a map.

Importantly, the problem of SLAM – learning a map while simultaneously estimating the robot's position in that map – consist of two mutually dependent subproblems. If a complete and accurate map existed, simpler algorithms such as *Monte Carlo Localization* could have been utilized for generating position estimates [59, 8]. Likewise, if a complete history of accurate positions existed for the robot, map learning would be reduced to writing sensor data to a map representation [45]. It's a *chicken and egg problem* [59].

For this reason, the problem is recognized to be hard, and it requires a search for a solution in a high-dimensional space [27] of possible locations and maps.

1.4.1 Applications

SLAM has many applications in modern robotics, and features frequently in many kinds of mobile robots where other forms of position estimation, such as GPS, is not available or not accurate enough. Examples include indoor robots, subsea, underground and even on the surfaces of other planets [45].

SLAM can be used to facilitate autonomous navigation or for the purpose of providing maps itself. An example on the latter can be found in urban search and rescue (USAR) situations [37], where unmanned vehicles, drones or hand-held devices gather data and perform SLAM to provide maps of areas such as collapsed buildings or mines. Some automated systems can even plot the locations of trapped victims in these maps, providing useful help for rescue teams.

For indoor mobile robots, SLAM can provide a basis for automated navigation. The process can relieve operators of having to construct and maintain maps.

While few consumer products performing SLAM are available on the market, one example is *Neato Robotics'* autonomous vacuum cleaners [1]. Their vacuum cleaners perform SLAM algorithms in order to plan trajectories cleaning the whole floor.

Other well-known projects include *Stanley*, an automated car based on a *Volkswagen Touareg*, which won the *DARPA Grand Challenge* in 2005. The vehicle performs SLAM and determines drivable areas [60].

1.5 Previous Work by NTNU Students

Related work has been done NTNU students in recent years. Two of these projects are described briefly below.

1.5.1 LEGO Robots

Students at the Department of Engineering Cybernetics at NTNU have written several project reports and master theses around SLAM in *LEGO* robots [52, 31, 56, 57, 39, 40, 47, 30, 62, 61, 29]. The robots have been small, and mainly built from LEGO parts. The main body of SLAM related software has been implemented using *MATLAB* (e.g. [29]), while sensor data has come from IR-sensors, ultra sonic sensors and web cameras, as well as wheel encoders for odometry. The LEGO robots have implemented SLAM capabilities for mapping and navigation in an unknown maze. An overview of the reports is presented in [62].

The SLAM software was running on off-board computers, with sensor data and control signals being sent over a wireless connection between robot and computer.

The most recent master thesis, [29], utilized two LEGO robots. One had a sensor array consisting of a rotating set of IR sensors, while the other was equipped with a wireless web camera. The thesis reports “increased performance”, but does not document the accuracy of the maps, and has no examples of map output.

[62] reports frequent errors in the IR distance sensors and increased accuracy in the position estimates. The project report includes several map output examples for evaluation.

[62] also contains a short explaination of the methods used for SLAM in the LEGO robot systems. A MATLAB tool called *CAS Robot Navigation*

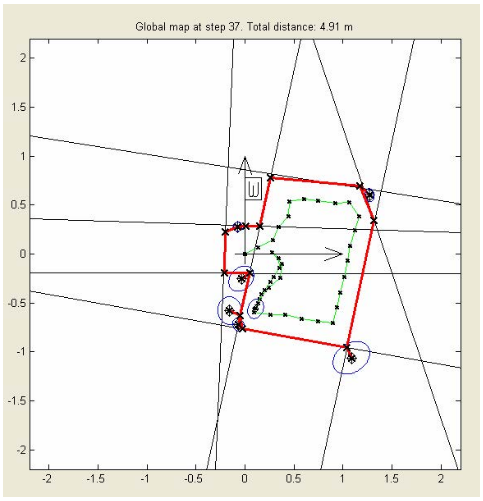


Figure 1.3: Example map output form [62]. The environment is rectangular. The SLAM algorithm introduces an invalid line segment cutting one corner. Figure courtesy of [62].

Toolbox is used. The system is *landmark-based*, and produces *beacons* and *line segments*, which together make up a map. The map and robot position is maintained in an extended Kalman filter [5].

Of the main problems with the SLAM algorithm used for the LEGO robots, [62] reports its inability to represent arbitrary environments. The beacons and line segments are unable to represent curves. There is no mention of how the system would have performed in larger environments. In addition, the system struggles with erroneous measurements, constructing beacons and line segments in the map that do not actually exist, which in turn can limit the navigation algorithms.

1.5.2 Eurobot-NTNU

NTNU has a team named *Eurobot-NTNU*, which competes in the annual competition *Eurobot*, usually held in France. The team builds robots and accompanying software for the purpose, which includes means of localizing the robot for navigation. The competition is held on game boards of 2×3 meters, mainly with known obstacles.

[8] describes a system for localizing the robot on the competition game board using *Monte Carlo localization* strategies and a laser range scanner (LIDAR). The system is able to localize the robot on the board using a tablet PC with limited performance.

In a later report by other students, this scheme was found to perform better than previous localization technologies for the robot setup [41]. It performed well, showing a low variance on the estimates on consecutive experiments, and was recommended for future use by the *Eurobot-NTNU* team.

The software for the Eurobot-NTNU team's robot includes a set of modules which can be used for other projects. Some of the modules are an *extended Kalman filter* module, a *web interface* module, a *path planning* module and a *guidance* module for following planned paths [50]. The modules are all written in the language *Go*.

Localization schemes for use in the Eurobot competition can rely on a static map, as the game table has known measurements to within reasonable accuracy. The work of building a map can be done manually, which eliminates the need for a SLAM algorithm. This seriously simplifies the software and computational needs, as the map without much error can be seen as being correct, accounting all errors to measurement errors and position estimate errors.

However, many parts of this system are related to SLAM problems. The use of a laser range scanner and some of the techniques used for localization, Kalman filters and particle filters, are common also for SLAM purposes.

1.6 Thesis Outline

This thesis will present the planning, implementation and testing of a software developed for performing simultaneous localization and mapping (SLAM) as well as path planning and guidance for a indoor mobile robot. In Chapter 2, background material on the SLAM problem is presented, as basis for some of the later decisions regarding the implementation. Chapter 3 discusses some of the alternatives for sensors the software could use for achieving its goals, and includes the decisions which were made with regards to sensors. Chapter 4 details the implementation of the system and decisions made with regards to SLAM, guidance and other problems, before experimental results and documentation is presented in Chapter 5. A discussion is presented in 6, while thoughts for future work is left for Chapter 7.

Chapter 2

Simultaneous Localization and Mapping

A short introduction to the SLAM problem was given in Chapter 1. This chapter gives a more in depth look into some aspects of the SLAM problem, how different techniques can help and what characterizes different solutions.

The implementation of the specific system for this thesis is left for Chapter 4.

2.1 Map Representations

Several map representations are recognized for SLAM purposes, and most of them can be put in one of two categories: *landmark-based maps* and *occupancy grid maps* [59]. Many algorithms are only able to build one of these map representations, a fact which has consequences for software implementations – much code cannot be shared between SLAM implementations of different kind. Exceptions do however exist, such as the SLAM algorithm proposed in [48], which uses an *hybrid SLAM* method which combines grid-based and graph-based approaches.

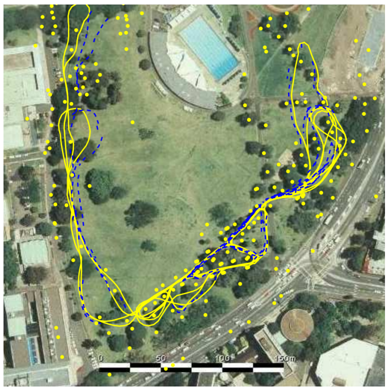


Figure 2.1: A typical run of the FastSLAM algorithm using landmark-based maps, presented in [43]. The line represents the robot's estimated path, while the dots represent the positions of the landmarks. Figure courtesy of [43].

2.1.1 Landmark-based SLAM

Landmark-based maps are based on *landmarks*, which are features in the environment. Landmarks can be corners, line segments or points. An example is presented in [43], where trees are used as landmarks, detected by searching for local minima in laser measurements (see Figure 2.1).

The landmarks are seen as distributed in a continuous space. In other words, each landmark is associated with a position in space, possibly with some rotation or other properties. The landmarks are assumed to be relatively sparse and unambiguous [18].

One of the aspects which makes landmark-based SLAM hard, is the problem of associating observed landmarks with the correct entry in the map, as illustrated in Figure 2.2. If an error is made in the data association process, the SLAM process can suffer catastrophic consequences. If the correspondence between observations and landmarks in the map was known, the problem would be much simpler.

A real advantage with landmark-based maps is the compactness of the representation. A list of landmarks takes up very little space, and is cheap to hold in a computer's memory. Formerly popular SLAM implementations grow quadratically in the number of landmarks, both in terms of memory and processing time, but more recent implementations can limit the growth

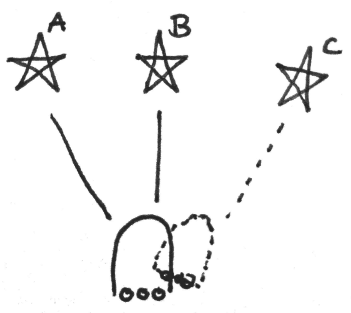


Figure 2.2: The data association problem of landmark-based SLAM. The robot senses two landmarks, but two different interpretations are possible. The sensed landmarks can either be landmark A and B, or landmark B and C. When the robot chooses one of the possible associations, this has implications for its estimated location.

to near linear [45].

Being able to represent maps with small amounts of data, landmark-based maps are especially well suited for constructing several map versions at a time, allowing for multi-hypothesis tracking.

2.1.2 Occupancy Grid Maps

An alternative to the landmark mindset is occupancy grid maps, or *metric maps*. While the landmark-based SLAM processes single out recognizable landmarks, occupancy grid maps can be seen as regarding *everything* as landmarks. The individual sensor measurements are assumed to be individually not very distinctive, but dense [18].

Grid maps discretize the environment into a grid – for regular maps a grid of two dimensions. They can have a variety of different ways of representing a cell, everything from a simple binary bit to tree structures [18]. A common representation is a number, for example an estimated *log-odds* for the cell to be occupied. Whatever the internal representation, the main objective is to determine for each cell, if it is occupied or free.

Each cell in the grid map has a predefined location and in most cases a predefined size. The collection of all these cells form a map, much like

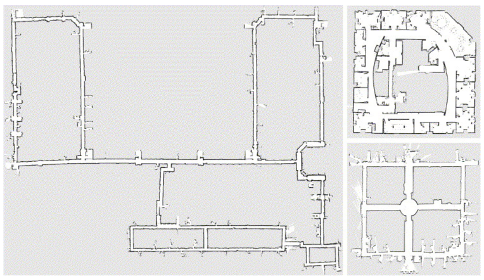


Figure 2.3: Example of output from a grid map based SLAM algorithm. The output closely resembles floor plans, and are easy to interpret visually. Image courtesy of [27].

pixels in a computer image. The simplest representation is to form a grid structure upon initialization, and keep this throughout the process.

Two of the main advantages of the grid map mindset are ease of visualization and the level of detail.

While landmark-based maps can be abstract, grid based maps are very concrete and closely resembles regular maps such as floor plans, as seen in Figure 2.3. This makes the grid maps easy to interpret by humans and makes automatic path planning simpler [18].

Grid maps can also be created of almost arbitrary resolution, allowing maps with high levels of detail. Given the assumption that every cell in the map is independent on the other cells, the map can represent arbitrary environments, which is a clear advantage.

A commonly cited disadvantage of grid maps [48, 18] is the huge amounts of block memory they can require. Maintaining a large number of detailed grid maps requires significant amounts of memory, even for modern computers. As a result, many algorithms [54, 37] do not necessarily utilize more than one grid map.

2.2 Sensors for SLAM

The choice of sensors for performing SLAM is large, and different types of sensors are used in different contexts. Autonomous underwater vehicles can use sonars, while unmanned aerial vehicles can use radar systems, infrared

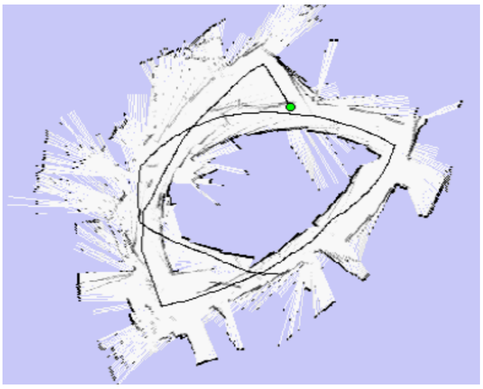


Figure 2.4: Mapping based on raw odometry. The map is highly inconsistent and is hard to interpret. Image courtesy of Sebastian Thrun, `robots.stanford.edu`., screenshot of video demonstration *Raw Data of Intel Research Lab.*

cameras or other means of sensing. The focus of this thesis is on autonomous indoor wheeled robots.

2.2.1 Range Sensors

While it is not strictly required, most autonomous mobile robots have some form of range sensor. A range sensor can tell the distance to the nearest object in a given direction or sector. Ultrasonic, IR and laser based systems (LIDAR) are common, while vision systems based on digital cameras represent another alternative. An outline of how a SLAM algorithm could work based on touch is mentioned in [11], as *bug algorithms*.

The range sensors form the method of detecting obstacles, facilitating the generation of a map and location tracking of the robot relative to them, whether the map be landmark or grid map based. The range sensors also play a role in correction of odometric errors where odometric sensors are present.

2.2.2 Odometric Sensors

In order to efficiently and correctly cope with more situations, many robot designs and SLAM approaches also include odometric sensors, that is, sensors which measure the distance traveled.

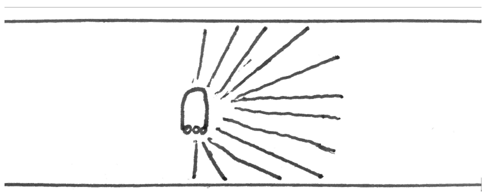


Figure 2.5: Robot driving in a long corridor using a range sensor with limited maximal range. Without odometry, the robot is unable to track its position in the direction of the corridor.

In fact, a naïve SLAM algorithm could be solely based on odometry and *dead reckoning*, writing range sensor readings to the map based on the position deduced by odometric measurements. Such mapping with raw odometry most often gives inconsistent results, since errors accumulate and are never corrected, as illustrated in Figure 2.4.

However, some situations do require odometric input. Consider for instance a long corridor, with straight walls, as sketched in Figure 2.5. A range sensor would only be able to sense the robot's lateral displacement in the corridor, since there is no way to discriminate between different positions along the corridor. Odometric input can provide information about the movements.

Even so, a case can be made that the importance of odometry has declined. The main reason for this, is that odometric readings are often very imprecise compared to LIDAR readings.

While the individual distance measurements of a LIDAR might have higher uncertainty, there are usually so many of them, allowing for a more precise estimate to be deduced through *scan matching*, comparing several scans and inferring their relative positions. For instance, [26] reports that the probability distribution $p(z|x)$ is much more peaked than $p(x|x', u)$, meaning the probability of obtaining the reading z given a position x is less uncertain than the probability of the position x given the previous position x' and an odometric reading u.

Odometry could thus be thought of more as an hint, directing a scan matching procedure to a region of valid scan matches, and some SLAM algorithms can produce accurate results without odometry at all [37].

2.3 Scan Matching

Scan matching is a concept frequently used in SLAM algorithms. For some algorithms, scan matching is the most central aspect. Combining range sensor measurements form one LIDAR revolution, hereafter called *scans*, can be used to estimate the movement of the robot between these scans.

LIDAR scans or measurements originating from other range sensors are represented in a coordinate system fixed to the sensor unit. When the robot and LIDAR move, the coordinate system in which the LIDAR scans are given is moved relative to world and map coordinates. Scan matching is the procedure of aligning different LIDAR scans to a world or map coordinate system, based solely on the scans themselves, or with the help of other inputs. In this view, scan matching aligns LIDAR scans to each other, or to a previously obtained or static map [37].

Scan matching is related to matching of geometric primitives, which is an important problem in computer vision, required for example for object tracking and object recognition [9].

Several algorithms are used for scan matching, for example *iterative closest point* (ICP), *polar scan matching* (PSM) and *normal distribution transform* (NDT).

Work on scan matching based localization started with ICP [37] in 1992 [9], which is a general approach for registering 3D point clouds. ICP iteratively matches points in one set to the closest points in other [65]. The ICP algorithm takes as input some motion estimate between the two scans, before iteratively using a least-squares estimation to reduce the average distance between the matched points. Using heuristics based on maximum tolerable distances between matches and orientation consistency, ICP seeks to eliminate false matches. ICP's point correspondence search is however expensive [37].

Polar scan matching assumes the scans are given in polar coordinates, like LIDAR scans are. PSM takes advantage of this coordinate system and does not need to perform an expensive search for correspondences [15]. A preprocessing step is required, where the algorithm seeks to remove erroneous measurements and group measurements of the same object. The algorithm then alternately estimates translation and rotation until convergence or a maximum number of iterations is reached.

Normal distribution transform scan matching models the probability of locally measuring a point at some position by a collection of normal distributions [9]. When matching a second set of scan endpoints and some geometric transform between them, a measure is defined by mapping all points according to the transform, before evaluating the corresponding local normal distributions. The sum of these points is maximized through a Newton's method optimization. NDT is successfully applied to the SLAM problem by [9], showing it can be used for mid-size indoor environments.

Two additional scan matching procedures is presented in sections 2.8.2 and 2.8.1. The first one using map gradients and a Gauss-Newton optimization approach, the other using a Monte Carlo optimization.

2.4 Particle Filters and SLAM

Particle filters represent a scheme for estimating state probability distributions, much like Kalman filters. However, where Kalman filters represent probability distributions using multivariate Gaussians, Particle filters represent distributions using *particles.* The particles can be seen as samples of an underlying, unknown, probability distribution [45].

In a particle filter, areas of high probability will contain more particles than areas of low probability. This is achieved by giving each particle a *weight* based on how they coincide with measurements. A particle which has a high probability of observing a measurement will be given a higher score.

In a *resampling* step, a new set of particles is drawn from the previous iteration based on the particle weights. Each particle is drawn, with replacement, by a probability proportional to its weight.

Given enough particles, this representation can approximate arbitrarily complex and multi-modal probability distributions [45]. The latter is especially advantageous in ambiguous situations, like those which can arise when globally localizing a robot in a known environment using range sensors. A localization approach called *Monte Carlo localization* (MCL) describes this particle filter approach to localization.

Particle filters can use probabilistic models to propagate particles. Such models can be parametrized, incorporating pseudo random numbers to distribute particles. This ensures particles which are alike after resampling can differ after the propagation step.

See for instance [59] for a more in-depth description of particle filters, or [8] for a discussion and implementation of MCL.

One of the main drawbacks of particle filters is their inability to scale in the number of dimensions. The number of particles needed for estimation grows exponentially in the number of dimension of the estimation – they suffer from *the curse of dimensionality.*

[16] introduces *Rao-Blackwellized Particle Filters (RBPFs)*, exploiting the structure of the application to increase the efficiency of the particle filter. They marginalize out some of the substructure in order to reduce the size of the space over which we need to sample.

In the context of SLAM, a naïve particle filter implementation would see each particle as a position together with a map, and would sample over *all* possible maps and positions. The map is in this sense highly dimensional, which is easiest to see for grid maps, where each grid cell represents a dimension – unknown, occupied or free. This leads to an extremely inefficient computation.

An important effect cited in [45] is the following: error in the robot's path correlates errors in the map. Moreover, the correlations between elements of the map *only* arise through robot pose uncertainty. For landmark-based maps, this means landmark positions could be estimated independently, while for grid maps, uncertainty in cell classifications would only arise through range sensor error.

Through the technique of *Rao-Blackwellization*, named so because it is related to the Rao-Blackwell formula [16], particle filters can factorize the SLAM posterior $p(\boldsymbol{x}_{1:t}, m|\boldsymbol{z}_{1:t})$ into a path posterior and map posteriors as

$$p(\boldsymbol{x}_{1:t}, m|\boldsymbol{z}_{1:t}) = p(\boldsymbol{x}_{1:t}|\boldsymbol{z}_{1:t}) \cdot p(m|\boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}). \tag{2.1}$$

The first factor on the right hand side only represents pose estimation, and the second factor can be computed efficiently since the poses of the robot are known when estimating the map. Different formulations and notations of the equation exists, based on assumptions for example of landmarks [45, 26].

Rao-Blackwellized particle filters for SLAM utilize this factorization by for each particle, regarding the path as *known*. The landmarks or the grid map can thus be computed from a known path. For landmarks, this means estimating landmark positions individually, with no explicit cross-correlation.

Each particle in the filter can represent slightly different paths – samples of a distribution of paths. The particles thus have different maps. Measurements are used to weight the particles, and resampling favors particles (and maps) which are probable.

2.5 Loop Closure and Map Inconsistencies

[28] cites *loop closure* as a particularly hard aspect of SLAM. It arises when a robot traverses some cycle in the environment, and the SLAM algorithm has to recognize the fact that the observed portion has been observed before and which part of the constructed map it represents. In other words, the robot has to associate currently observed data with previously observed data.

Some SLAM algorithms handles loop closures explicitly [28], by correcting backwards in time. Others have built-in mechanisms which do not explicitly consider loop closures, but which are capable of correcting maps in such situations [26, 45, 17]. Some SLAM algorithms does not treat loop closures in any special way [37, 54] and must be accurate enough for the maps not to be inconsistent.

Map inconsistencies can arise when the SLAM algorithm fails to match new observations with old. This leads to several problems. An underlying problem is the fact that the same physical location is represented at two or more different locations in the map, which causes large problems for path planning. Some of these inconsistencies can be observed as *double walls*, where the same wall or object is represented twice at slightly different locations in the map. A typical example is shown in Figure 2.6.

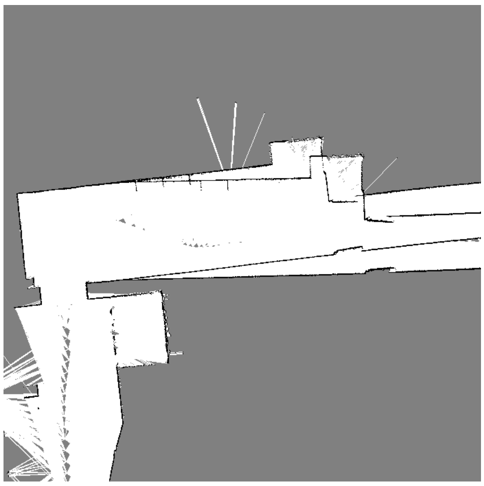


Figure 2.6: After traversing a large loop of the environment, the robot meets a previously observed area, but fails to associate the currently observed data with the previously mapped area. Here, the result is double walls, in this case both translated and rotated slightly. The map is thus inconsistent.

2.6 Single Extended Kalman Filter Based SLAM

Introduced in an article from 1986 [53], the single extended Kalman filter approach to SLAM was dominant for years [45]. The approach uses a high dimensional EKF to provide a posterior over features in the map and robot pose. Typically, map features are implemented as landmarks, with a representation of two states in the EKF for each landmark, plus 3 for the robot pose, that is, $2N + 3$ dimensions for N landmarks.

One problem with this scheme surfaces when maps grow large. The complexity of the EKF representation grows quadratically, both in computational time and memory. This is a consequence of the Gaussian representation. Note that the correlations between all pairs of state variables are maintained, so any sensor observation could affect all state variables.

When sensor readings are considered, landmark associations between sensor data and already observed landmarks are typically done in a maximum likelihood way. Once the association has been made, a wrong data association can never be undone.

This is the approach taken of the *CAS Robot Navigation Toolbox*, which was used for the LEGO robot described in Chapter 4.

2.7 Rao-Blackwellized Particle Filter Based SLAM

Some SLAM algorithms are best characterized by their use of Rao-Blackwellized particle filters, as described in Section 2.4. The family of algorithms include both landmark and grid map based algorithms.

FastSLAM, originally presented in [45], is a Rao-Blackwellized particle filter based approach where each particle in the filter maintains a full posterior distribution over robot poses and maps. The original FastSLAM algorithm is landmark-based.

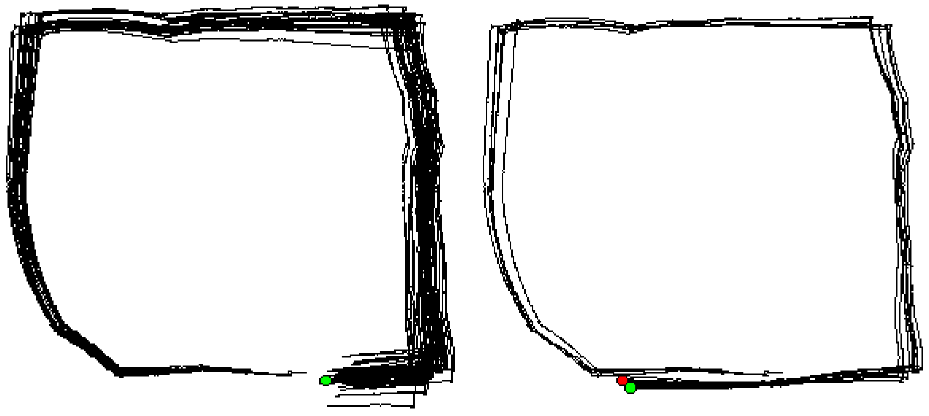
The algorithm breaks up the single, large error covariance matrix of the single Kalman filter approach presented in the Section 2.6 into one extended Kalman filter per landmark [45].

The full algorithm considers several sets of these landmark filters. A number of particles are maintained at a time, where each particle contains one Kalman filter for each landmark, in addition to a Kalman filter for the current position.

Data associations are done on a per-particle basis, which gives rise to one of the advantages with FastSLAM. Particles with the correct data association will tend to receive higher particle weights, and will survive in the filter. In this way, FastSLAM tracks several data association hypotheses. Note that the per-particle data association means that particles can have different numbers of landmarks.

The Rao-Blackwellized particle filter based SLAM algorithms do not necessarily explicitly cover loop closing scenarios. However, a loop closing is a typical point where some particles will be more consistent with the sensor measurements than other, triggering re-samplings of the particles. Particles more consistent with closing the loop will tend to “survive”, and more maps will from then on be based on these more consistent particles. This scenario is illustrated in figure 2.7, which visualizes the particle cloud shortly before and after loop closure.

Although the FastSLAM algorithm is based on landmarks, modifications can be done in order to use grid maps. [28, 26] shows such a scheme, where



(a) Before closing loop **(b)** After closing loop

Figure 2.7: FastSLAM closing a loop of the environment. Upon closing the loop, some particles are more consistent than other, and receive a larger weight. When re-sampling, the lower-weight particles are eliminated. Work from then on is focused on the particles, or hypotheses, more consistent with observations. Figure courtesy of [28].

a Rao-Blackwellized particle filter is used together with scan matching in order to produce grid maps.

Each particle of the filter maintains an estimate of the robot pose. Additionally, the particles maintain individual maps instead of a series of Kalman filters for landmarks. The maps are occupancy grid maps.

In order to lower the computational needs, series of LIDAR measurements can be transformed into odometry measurements using scan matching, while the remaining LIDAR measurements are used for mapping. Various scan matching approaches can be used [26].

Like the landmark-based FastSLAM algorithm, this occupancy grid map version also performs multi-hypothesis tracking through its particle filter approach. Having several hypothesis of the robot's pose to choose from can help with closing loops in the environment.

A drawback with occupancy grid mapping in general is the large amounts of memory required for representing maps. This problem is amplified when particle filters are assigning an individual map to each particle.

[17] presents *DP-SLAM*, which builds further on Rao-Blackwellized particle filter occupancy grid map SLAM. By reversing the map data structure, DP-SLAM provides a more efficient data structure for representing several related maps.

DP-SLAM introduces the concept of *particle ancestry trees*, where each particle contains a pointer to its parent. Additionally, each particle contains a list of the has a list of the grid squares which it has updated.

DP-SLAM's, or more specifically, *DP-mapping*'s, solution to the problem of maintaining large quantities of related maps is to associate particles with maps, rather than vice versa. The maps are constructed on a single grid, with each cell maintaining a balance tree of references to the particles which have made changes to that cell. The combined structure can be seen as a tree representation of where the paths of different particles diverged, and and can facilitate exploitation of the redundancies between the maps [18]. According to [18], the method can maintain hundreds of maps in real time.

2.8 Scan Matching Based SLAM

Some SLAM algorithms do not necessarily make use of particle filters, and instead rely solely on scan matching. Two specific implementations are described below.

2.8.1 TinySLAM

TinySLAM was developed by Steux and El Hamzaoui [54] with the goal of developing a simple SLAM algorithm not exceeding 200 lines of C code. It is based on the availability of odometry and a LIDAR sensor. The algorithm is small and easily understandable, based on particle filters. The algorithm produces grid maps.

The algorithm's functioning comes down to two main parts: *distance calculation* and *map updating.* It has a routine for calculating the "distance" from a LIDAR scan to a map, given the pose where the scan was recorded, which only considers the end points of the distance readings. An observation

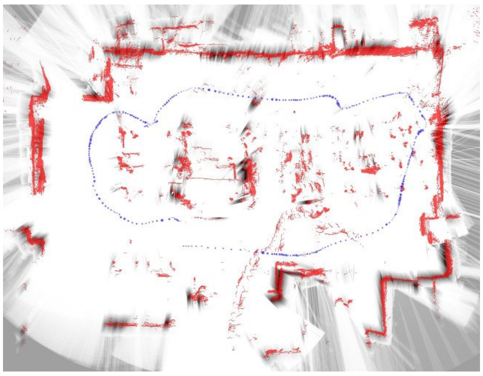


Figure 2.8: Example output from the TinySLAM algorithm, a map of the CAOR laboratory in Mines ParisTech, as presented by the authors of the TinySLAM paper. Image courtesy of [54].

can be made that the functioning is not unlike the scan-based sensor model developed by Thrun and collegues and presented in [59]. Essentially, the scheme is based on the value of each grid cell indicating the probability of a laser beam to stop at that cell. Given the origin of the laser beam and its length, one can easily calculate the end cell position, and through a simple lookup determine the á priori probability that a beam should end there. The probability is based on the cell's *distance* from the nearest obstacle. For the *localization* problem, this *likelihood field* is typically pre-computed such that each grid cell value indicates this distance [59, 8].

TinySLAM uses odometric data to estimate where the robot was when each incoming LIDAR reading was taken, given a mathematical model of the robot's movements. This initializes a scan matching procedure in the distance calculation part of the algorithm. The scan matching procedure is a *Monte-Carlo search* for the best fitting origin of the LIDAR scan, which uses the distance calculation routine to rate randomly distributed poses in the neighbourhood of the estimated position. After a number of iterations are done, the pose considered which had the lowest distance to the map is chosen.

After this pose is chosen, the algorithm goes on to write the LIDAR scan to the map. It does so by drawing a line for each beam, with a *hole* around the end point of the beam, where the value gradualy declines, reaching a minimum in the end point and then inclines back to the initial value before the line drawing is completed. Over time, the this will allow the map to approximate a likelihood field as described above.

The scan matching is done by sampling randomly from a distribution. This means different intances of the algorithm might not produce the exact same map, even given the exact same input data. The algorithm could be extended to perform in a Rao-Blackwellized particle filter manner, where each particle has its own map.

TinySLAM is released as open source software under MIT License available from `openslam.org`, written in C. The implementation uses a range of different `struct` type definitions in order to hold data in an intuitive manner. The code is well modularized and easy to understand when read together with [54].

TinySLAM performs a fixed number of iterations per LIDAR scan for it's scan matching optimization. In general, a higher number of iterations give a more optimal result, limited only by the computer's resources.

2.8.2 Hector SLAM

Kohlbrecher, von Stryk, Meyer and Klingauf presents a SLAM approach in [37], which has an implementation known as *Hector SLAM* written by the authors and described in [35]. While the article presents a use case where an inertial measurement unit is combined with a LIDAR for estimating full 3D motion (6 degrees of freedom), the implementation does not depend on the intertial sensors. For the rest of this thesis, both the underlying principle described in [37] and the implementation will be referred to as *Hector SLAM*.

The motivations behind the development of Hector SLAM include a need for estimating full 3D motion, which is needed in applications like *Urban Search and Rescue*. In USAR scenarios typically include unstructured environments where the unit can experience roll and pitch motion. Additionally, there was a need for an algorithm compatible with low-weight units with limited computational resources. The authors had a goal of enabling sufficiently accurate localization while keeping computational requirements low.

Hector SLAM assumes distance measurements comes from some accurate, high-resolution and high-frequency range measurement unit like a LIDAR.

While the algorithm is designed to support full 3D motion estimation, the SLAM algorithm performs 2D mapping, making it suitable also under the assumption that the environment is planar.

Hector SLAM works with grid maps, and its underlying principle is scan matching. When the algorithm starts, the first LIDAR scan is written to the map. Subsequent LIDAR scans are matched with the map in order to estimate some rigid transformation between them, that is, the relative displacement in translation and rotation. LIDAR scans are written to the map depending on criterions of a minimum displacement in translation or rotation relative to the location of the previous map writing.

The discussion of Hector SLAM proceeds by presenting a summary of its inner workings. For more details, the reader is referred to [37] or the source code at [36], on which this summary is based.

The scan matching procedure is inspired by work in computer vision, originally presented in [38]. The algorithm uses a Gauss-Newton approach for optimization, and formulates the optimized rigid transformation as

$$\boldsymbol{\xi}^* = \underset{\boldsymbol{\xi}}{\operatorname{argmin}} \sum_{i=1}^{n} [1 - M(\boldsymbol{S}_i(\boldsymbol{\xi}))]^2, \tag{2.2}$$

where $\boldsymbol{\xi} = (p_x, p_y, \psi)^\top$ and $\boldsymbol{S}_i(\boldsymbol{\xi})$ denotes the world coordinates of the end-point of measurement i. The occupancy value $M(P_m)$ of some coordinate P_m is computed using a bilinear filtering scheme in order to provide sub-grid cell accuracy.

[37] presents a derivation of a Gauss-Newton equation $\Delta\boldsymbol{\xi}$, minimizing Equation 2.2 as

$$\Delta\boldsymbol{\xi} = \boldsymbol{H}^{-1} \sum_{i=1}^{n} \left[\nabla M(\boldsymbol{S}_i(\boldsymbol{\xi})) \tfrac{\delta \boldsymbol{S}_i(\boldsymbol{\xi})}{\delta \boldsymbol{\xi}}\right]^\top \left[1 - M(\boldsymbol{S}_i(\boldsymbol{\xi}))\right] \tag{2.3}$$

with Hessian matrix

$$\boldsymbol{H} = \left[\nabla M(\boldsymbol{S}_i(\boldsymbol{\xi})) \tfrac{\delta \boldsymbol{S}_i(\boldsymbol{\xi})}{\delta \boldsymbol{\xi}}\right]^\top \left[\nabla M(\boldsymbol{S}_i(\boldsymbol{\xi})) \tfrac{\delta \boldsymbol{S}_i(\boldsymbol{\xi})}{\delta \boldsymbol{\xi}}\right]. \tag{2.4}$$

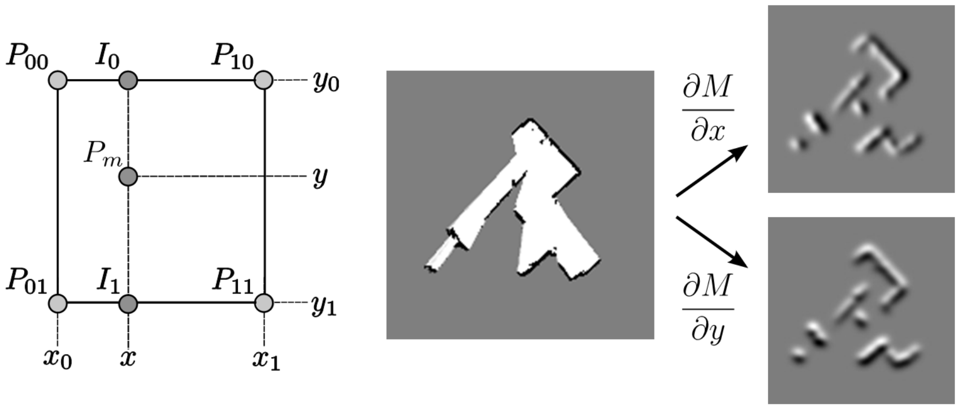


Figure 2.9: Illustraton for some of the aspects of the filtering of obtaining sub-grid cell accuracy. The coordinate P_m lies somewhere within the grid cell centers P_{00} through P_{11}, and its value can be approximated through a weighting of these values. Figure courtesy of [37].

Equations 2.3 and 2.4 rely on $\nabla M(P_m)$, the gradient of the map at some coordinate P_m. The gradient can be approximated via bilinear filtering of the four closest integer coordinates in the map P_{00}, P_{01}, P_{10} and P_{11}, representing the midpoints of the surrounding map cells of P_m. The gradient can be approximated as

$$\begin{aligned}\frac{\delta M}{\delta x}(P_m) &\approx (y-y_0)[M(P_{11})-M(P_{01})]\\ &\quad +(y_1-y)[M(P_{10})-M(P_{00})] \qquad (2.5)\\ \frac{\delta M}{\delta y}(P_m) &\approx (x-x_0)[M(P_{11})-M(P_{10})]\\ &\quad +(x_1-x)[M(P_{01})-M(P_{00})], \qquad (2.6)\end{aligned}$$

where x, y, x_0, x_1, y_0 and y_1 are the map coordinates of the integer values P, as illustrated in Figure 2.9. The approximation assumes $x_1-x_0 = y_1-y_0 = 1$ in map coordinates.

An approximation of the match uncertainty can be computed as the inverse Hessian scaled by some sensor specific constant: $\boldsymbol{R} = \sigma^2\boldsymbol{H}^{-1}$.

Further optimizations of the scan matching procedure is achieved through another approach inspired by computer vision and image processing. Since the procedure is based on gradient descent, getting stuck in local minima

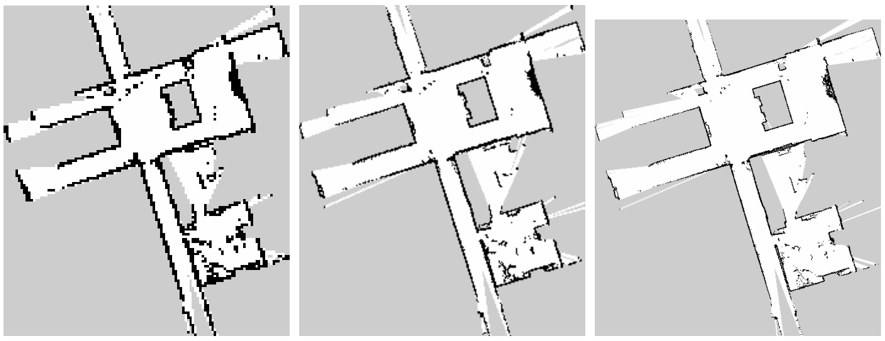

(a) 20 cm cell sides. **(b)** 10 cm cell sides. **(c)** 5 cm cell sides.

Figure 2.10: Image pyramid like structure of Hector SLAM. The same map is generated and stored at different resolutions. Images courtesy of [37].

is a source of error. Hector SLAM seeks to minimize these effects using a structure similar to *image pyramids*, where the map is stored at different resolutions, as illustrated in Figure 2.10.

The scan matching is done at all the levels of the image pyramid like structure, starting with the coarsest. As the matching works its way towards the finest representation, the position estimate $\hat{\boldsymbol{\xi}}$ is assumed to be more precise. The estimate from one layer is passed on as the start estimate for the next. Note that the generation of the different layers is not done by downsampling, but by writing scaled versions of the LIDAR scans to each map, which is more computationally effective and ensures consistency across layers.

Odometry or other sources of position data can be used to help the scan matching procedure as a start estimate for the scan matching. The estimation can be done with Kalman filters or other methods sensor fusion. The results of the scan matching can help in the other direction, correcting this scheme.

An example output of the Hector SLAM algorithm is presented in Figure 2.11. As the figure shows, the loop in the environment was successfully closed, even though the algorithm does not include an explicit handling of loop closures.

The Hector SLAM algorithm shows promise as an accurate and relatively

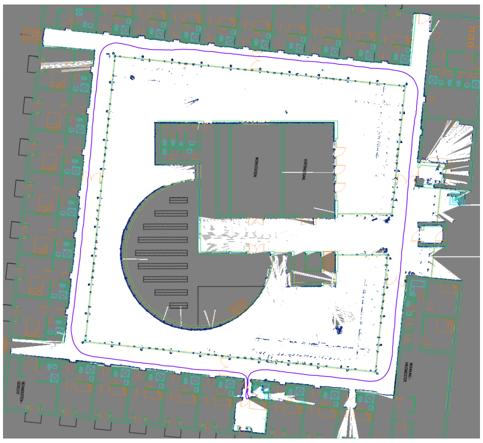


Figure 2.11: Example output of the Hector SLAM algorithm overlayed with ground truth data. The LIDAR sensor used for the experiment was an *UTM-30LX* LIDAR unit from Hokuyo. Image courtesy of [37].

simple approach to the SLAM problem. Its loose coupling with odometry data adds to its flexibility as a part of a larger system software. According to [37], it uses less than half of the available processing power of a cheap *Atom Z530* CPU, demonstrating its low computational demands.

Chapter 3

Sensors

In order to fulfill the goals of map generation, localization (SLAM) and guidance, the robot would need several sensors to supply it with measurements. As part of the problem description, a suitable set of sensors for the robot had to be decided, within budget limits. The material presented in this chapter builds on Section 2.2.

The sensors should enable the robot to perform the tasks mentioned, within reasonable limits. One requirement is that they should enable the robot to generate maps. Other requirements include ensuring that the robot can potentially be operated autonomously by being able to detect obstacles and collisions, both for safety and for efficient operation.

Some types of obstacles were however not considered, such as tables and downward stair cases.

Paying special attention to making sure the sensors are suitable for solving the SLAM problem, research into what alternatives exist and their properties are described in the following section.

Most SLAM approaches assume at least two kinds of sensors to be in place: distance sensors and odometry, see e.g. [62, 59, 18, 28]. Distance sensors give an estimate of the distance to obstacles, usually in several directions simultaneously, while odometry provides an estimate of how the robot has moved from the last time step, and throughout the execution of the robot software.

Drawing from conclusions and research presented in the previous sections, it was seen as beneficial for the robot to be equipped with both range sensors and odometric sensors.

3.1 Discussion of Range Sensors

Popular distance sensors include infrared distance sensors, ultrasonic distance sensors, LIght Detection And Ranging (LIDAR) sensors and camera (computer vision) sensors. Their goal is to provide an instantaneous local map of the robot's surroundings. Each sensor type has different abilities and limitations. We proceed by discussing the alternatives mentioned above.

3.1.1 Vision Based Sensors

The robot was already specified to include two web cameras, as described in Chapter 1, with a main goal of providing aid for operators to control the robotic arm, and also for manual driving modes. Since the web cameras are cheap and already required, they would provide a low cost alternative for range detection. Having two web cameras, it's possible to detect range even without movement of the robot, through stereo vision; see e.g. [58].

One advantage with vision based systems is their ability to register 3D information. Reconstructing the 3D environment, they have the ability to collect data in the height direction, making it possible to detect e.g. a table, which has a small footprint, but depending on its height, it might not be safe to drive between the table legs.

Several factors also make the vision based alternative less attractive, of which the largest are accuracy, calibration and the nature of the measurements.

The robot's onboard computer is relatively fast and judging from [49] and [12], they should be able to facilitate localization in real time. However, the maps they produce seem to be of lesser quality than maps produced by LIDAR sensors (see e.g. [19] for examples), and the software required is inherently more complex than that for other sensor types [49].

Vision based systems have to be accurately calibrated [58]. The system needs to know the configuration of the two cameras relative to each other and relative to the robot. In the case of the robot in question, the web cameras were mounted on a robotic arm. This means they were moving relatively to the robot. It would make the system more complicated if their position relative to the robot had to be estimated.

Even with dedicated cameras for SLAM, the setup seems more prone to errors in misconfiguration. Initial tests also showed the robot was shaking when driving, amplified for the higher parts of the robot, where the cameras were mounted.

3.1.2 Infrared and Ultrasonic Sensors

Infrared and ultrasonic distance sensors were very popular some years ago [49], and are still used for some applications [62], mainly because they are small and very cheap. Some of their limitations include the width of their *cones* of measurement, which can be up to 30° [49]. They are also prune to multipath reflections and cross-talk between sensors, leading to erroneous measurements [51].

Given the low angular resolution of these sensor types, they seem more suitable for SLAM in small environments like tables and small rooms. This assertion is supported by some of the work done for SLAM with such sensors in recent years, like [3], which uses such sensors to map an area of 2×1 m. While their results are impressive for a small robot with cheap sensors, the maps do not seem accurate enough to map large areas.

3.1.3 LIDAR Based Sensors

A LIDAR sensor unit uses laser light to measure the distance to surrounding objects. They are sometimes called "laser radars", because the sensor output data can be compared to that of a radar. LIDAR units exists which can measure distances in some plane, a cone or several cones, which are used in "high end" applications like [42].

In recent years, LIDAR sensors have become cheap enough to become popular for many applications, even in consumer products such as the *Neato Robotics* autonomous vacuum cleaner range [1]. They can be very precise, with an error of only a few per cent over distances as large as 80 meters or more (e.g. *SICK LMS5xx* [4]). They can measure houndreds of distances several times per second.

A LIDAR sensor would be the most costly alternative. The cheapest LIDAR sensors on the market currently goes for around 1000 USD, an order of magnitude above many cameras. However, it also seems to be the most accurate for planar (2D) map generation and the most reliable for detecing obstacles. While they supply a considerable amount of data points, several thousand per second, much information can be extracted from the readings without pre-processing them. Detecting the existence of an obstacle in the forward direction can be done simply by looking at the distance in that direction.

LIDARs do not suffer from cross-talk, and multipath does not occur often. They can give false readings when pointed at a mirror, window or when facing a matte black surface, but their overall performance seems reliable and predictible [8].

LIDARs are popular for use in SLAM research, with a large body of articles using them as their main sensor.

3.1.4 Suitability For Different SLAM Strategies

As mentioned in the introduction to SLAM in Chapter 1, there is a divide in the world of SLAM algorithms between landmark-based algorithms and grid based algorithms: the first must extract features from the measurements and match them between consecutive measurements, the last discretizes the world and tries to reveal the nature of each cell in some grid.

One goal for choosing sensors is to choose for the future, meaning we do not want to limit the system to use a specific strategy for its software. It was desirable to choose sensors which enabled both SLAM strategies. See Table 3.1 for a comparison.

To summarize the table, it seems the only bad fit is vision based SLAM with metric maps. Most of the work for vision based SLAM is directed to landmark-based SLAM approaches. One reason might be that the amount of data would be very large for large environments.

Sensor Type	Landmark-based	Grid map based
Vision	Yes. Examples: [7, 12]	No, although some work exists, e.g. [20] (uses Kinect-style camera).
IR / Ultrasonic	Yes, see e.g. [62, 3]	Yes, but very inaccurate [59, 10]
LIDAR	Yes, see [45, 44]	Yes, see [28, 27, 18, 1, 37].

Table 3.1: A comparison how different sensors perform with different map representations for SLAM.

3.1.5 Other Considerations

Among other considerations when choosing a range sensor, we can look at their abilities to enable the robot to navigate without accidents.

For avoiding walls or wall-like objects, LIDARs and IR/ultrasonic sensors should have no problems, although LIDAR can fail to detect a window or in extreme cases matte black walls or objects.

LIDARs which only measure in one plane would also fail to detect downward-going staircases or other “cliffs” in the environment, which could lead the robot to drive down such a staircase. Vision based systems could be able to detect such obstacles. Because IR and ultrasonic sensors are so cheap, several of them could be bought and placed downward to detect such obstacles. The same can be said for obstacles like tables, where some plane near the ground might seem like a free area, but where the robot might not fit under.

3.2 Discussion of Odometric Sensors

Odometric data for robotic vehicles is usually obtained from rotary encoders attatched to the robot’s wheels. When the wheel turns, the encoder generates signals interpreted as a rotation in a direction. An odometric reading consists of the amount of such signals registered in a time interval, from which software can easily obtain an estimate of the motion of the robot in that time interval. Odometric data can also be obtained from an inertial measurement unit (IMU) [37], which is particularly useful for robots without wheels.

This can be used for *dead reckoning*, calculating an estimate of the robot's position relative to the previous time frame or the robot's starting point.

Odometric readings from driving wheel-attatched encoders suffer from several problems, such as wheel slip. An alternative is to have separate wheels for the encoders, which are not attached to motors. Since these wheels do not drive the robot, their slippage will be very small.

They will also be suitable for detecting crashes. If the robot were to crash, the driving wheels might still turn, spinning on the ground. If we know that the control signal to the motors is such that the wheels should turn, but the odometric data indicate that we are infact standing still, this is a sign that the robot's path is obstructed and the robot's motors should be switched off.

3.3 Sensor Decisions

It was decided that the most "future proof" approach, with future projects and students in mind, was to equip the robot with both odometric and range sensors. This would provide a hardware basis for many different SLAM algorithms.

3.3.1 Range Sensor

When considering which range sensor the robot should be equipped with, infrared and ultrasonic sensors were excluded first, because of their lack of accuracy. It was assumed to be of benefit for the project and for future students improving the software if the sensors provide more accurate measurements than what seems possible with infrared or ultrasonic sensors.

When deciding between LIDAR and camera based systems, future projects were again taken into account. The measurements from a LIDAR are very easy to interpret and requires little post processing. On the other hand,

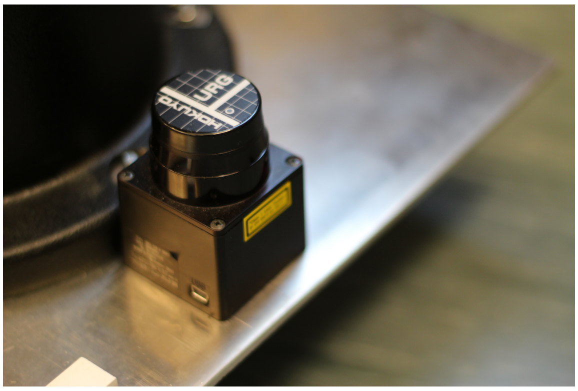


Figure 3.1: The *URG-04LX-UG01* LIDAR unit by *Hokuyo*, an entry level LIDAR recommended for use with autonomous robots by the manufacturer, as mounted on the robot.

vision based systems need relatively complex software for post processing.

LIDAR measurements provide the easiest means of knowing that the robot isn't driving into an obstacle in the plane of measurement. Vision based systems could possibly detect more obstacles, but also seem more error prone.

In the end, a LIDAR based system was chosen.

3.3.2 LIDAR Sensor Unit

An *URG-04LX-UG01* LIDAR unit from *Hokuyo Automatic Co.,Ltd* was mounted on the front center of the robot. The unit is an entry level planar scanning range finder, which the manufacturer recommends for autonomous robots. Its technical document [32] cites the following specifications:

- 240° scan angle with 0.36° resolution, giving a total of 681 distances per measurement output.
- 5600 mm maximum distance, with an accuracy of ±3% in the interval above one meter and ±30 mm in for shorter distances.
- 100 msec/scan time, giving measurements at 10 Hz.
- Weight of approximately 160 g and a footprint of 50 × 50 mm with an height of 70 mm.

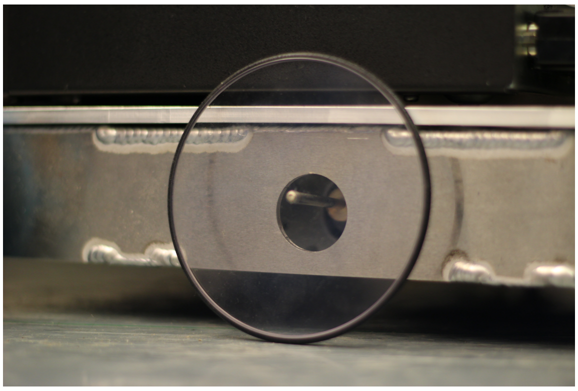


Figure 3.2: The encoder wheels as mounted on the robot.

Its laser is of class 1 safety, infrared 785 nm wavelength, and is completely safe to use without eye protection.

The LIDAR was mounted on the front center of the robot, carefully positioned so that no parts of the robot itself was obstructing its sector of measurement.

With an accuracy of ±3%, the accuracy at 4 m is ±120 mm. While the accuracy is limited, its high angular resolution and update frequency are helping factors. LIDAR units with higher accuracy and maximum range are available, and would improve the performance of map generation and localization. However, such units was not within budget limits of this project.

3.3.3 Discussion and Choice of Odometric Sensors

Although equipping the robot with an IMU would provide many options and interesting oppourtunities for future work, wheel encoders were chosen because they provide simple interfaces for the software and because they are cheap. This decision was done in cooperation with the student which was given the task of mounting the encoders and providing an interface between the encoders and the robot's PC.

The robot was equipped with two wheels used for obtaining odometric data via rotary encoders. They are mounted on the left and right side at the approximate axis of rotation. Their data thus represent how the wheels would have behaved if this was a true differential wheeled robot. A picture can be seen in Figure 3.2.

The encoders attached to the odometry wheels have a resolution of 200 pulses per revolution, grey coded to support both increments and decrements. With a diameter of approximately 10 cm, this implies approximately 600 pulses per second when driving at 1 m/s, a good enough resolution for most applications.

Chapter 4

Implementation

In order to fulfill the specifications of the problem description, a software system capable of on-line and off-line SLAM, with path planning, guidance and manual control of the robot was implemented. This chapter covers a number of decisions made for providing this functionality, both with respect to providing the best short-term results and for providing a solid base products for future work.

Throughout the chapter, details of the code implemented is presented. The chapter can serve as a discussion of design decisions, a walk through of code principles and implementation, and as a manual for future development.

4.1 Language, Environment and Operating System

One of the very first decisions had to be made with respect to the platform on which the implementation should be carried out. A choice had to be made whether or not to build upon already available software, which programming language should be used and which operating system should be targeted.

When the project started, past projects had already developed a SLAM environment for LEGO robots as described in Subsection 1.5.1, which could be built upon. Also available was *Robot Operating System* (ROS), which provides a wealth of libraries and tools for robot applications, including SLAM, path planning and guidance.

The LEGO robot project was based on MATLAB uses a landmark-based SLAM algorithm developed from the CAS Robot Navigation Toolbox. Sensor schemes included IR-sensors, ultra sonic sensors and web-cameras, but the accuracy presented in articles seemed to show limited accuracy, as seen in Figure 1.3. The LEGO project was also focused on small-scale robots and environments.

In order to develop from the LEGO robot system, adaptations for use with a LIDAR scanner would have to be made in order to gain precision and accommodate larger-scale environments. To the author's knowledge, no work of such kind had been carried out before on the LEGO project.

The SLAM algorithm and maps of the LEGO project was of the landmark-based type, and further development of this software would likely be locked to this paradigm. The system architecture was based on computations to take place at a computer not on the robot itself, and the single EKF based SLAM algorithm was presumed inefficient for large environments with its quadratic growth. See also the first parts of Section 4.5.

Developing further on the LEGO project software was therefore ruled out as a platform candidate.

ROS was a more seemingly more fitting candidate. Available from `ros.org` under an open source, BSD license, the *meta-operating system* can be modified to accommodate many needs [13]. Containing tools including drivers, libraries and visualizers, the user can pick and choose between modules in order to set up a system fitting her needs. A range of different SLAM algorithms, like Hector SLAM and the Rao-Blackwellized particle filter based grid mapping algorithm GMapping are readily available.

ROS provides visualization modes like *RVIZ*, a 3D visualization environment where the robot and map can be displayed in real-time. The user can control the robot manually or via path planners.

ROS includes a runtime *graph* communication system, a peer-to-peer network of processes using the ROS communication infrastructure. The framework seeks to support code reuse in robotics research and development.

An open question is the user friendliness for end-users. ROS robots are controlled via applications running on the user's hardware, which poses requirements on the client platform. The main goal of ROS is stated to be within research and development, not explicitly to provide an easy-to-use interface.

According to [13], ROS was at the time of writing only running on Unix-based platforms, primarily tested on *Ubuntu* and *Mac OS X*.

On the contrary, the project for this thesis was planned to include a robotic arm attached to the robot. According to [6], the control software for this arm posed implications for the choice of operating system: it could only run on *Microsoft Windows* based platforms. Plans for the physical implementation of the system included only one powerful computer. While running two different platforms on the same computer is possible through virtualization, such a scheme complicates the project and could impair performance. It was thus decided that the most optimal choice of operating system for the robot as a whole was Windows.

With a wish of providing an integrated and intuitive, user-friendly, platform-independent interface in compliance with other software running on the same computer, a choice was made not to base the software on ROS. With this established, a new choice had to be made on what programming language to use.

A wide range of suitable programming languages are available. Using *C++* and *POSIX* poses some special advantages, as much code is readily available or can be interfaced, for example from the ROS project. The NTNU Eurobot team also used this combination from 2004 to 2010 [50]. Some of the drawbacks of using C++ is the need for dealing with low-level operations like memory allocation. Also, dealing with threads can be cumbersome.

Another option is to go for *Python*, which is a dynamic language often used for scripting, although programs can be compiled to executables. Python has a module for thread-based parallelism [24], making thread based programs easy to implement. Python drawbacks include its performance. *The Computer Language Benchmarks Game* serves as a comparison of programming languages' performance, memory use and code length, available at `benchmarksgame.alioth.debian.org`. Here, C++ g++ is over 50 times faster than Python 3 in some of the tests. Python's dynamic typing typing can also lead to static errors at runtime.

A more recent addition to the world of programming languages is *Go*, often referred to as *Golang*. Go is a Google-maintained open source language. Go has been successfully used by the NTNU Eurobot team [8, 50]. It has good constructs for concurrent programming and error handling [46] as part of the language, and features *channels*, a built-in message passing system. While statically typed [2], the language attempts to make types feel lighter weight than in typical object oriented languages. It also has some of the other benefits of typical scripting languages like Python, in it's rich list of packages simplifying for example image handling and writing web servers. Go also includes built-in tools for interfacing C code. With regards to the benchmark game mentioned in relation to Python, Go scores comparably with C++ in most tests.

While C++'s benefits of ROS code reuse and performance were appealing, Go was chosen as the primary language of the code to be implemented. The new language was expected to simplify the development phase compared to C++, while also providing good performance and simple mechanics for concurrency. The choice also allowed for reuse of modules from the NTNU Eurobot project, and could in the future contribute to that project.

In conclusion, it was decided that a software solution for the project was to be written in Go, targeting the Windows platform, without building explicitly on some preexisting environment like ROS.

4.1.1 Short Introduction to Go

For the further discussion of the implementation, a short introduction to Go is provided for readers unfamiliar with the language. More detailed introductions are given in [50] and [46], while even more information can be found on the language's official website `golang.org` and the documentation in [2].

Go is a strongly typed language [50], where all casting is explicit. This means explicit casting must be done even for converting for example a *float32* to a *float64* number. Conditions for *if* statements must be booleans, so using an integer as condition is a compile-time error.

Moreover, the language features *structs*, much like C, but unlike C, the structs can have *methods*. There is no explicit inheritance, but a structure can be an *anonymous* part of other structures, allowing other code to access the anonymous field as if it was part of the struct.

```go
func functionName(x int, r float64) int
func (s *StructName) methodName(b bool) (int,
    error)
```

Listing 4.1: Go function and method signatures.

The Go function and method signatures are on the forms shown in Listing 4.1. Notice that the data types come after the variable names, and that a function or method can have several return values. Methods are defined outside the structure on which they work, by declaring a *method receiver* between the `func` keyword and the method name.

Go also has *interfaces*, which consist of a series of methods. The code does not explicitly declare what interfaces some struct satisfies – any struct fulfilling the interface's needs is allowed.

Go has built-in concurrency, via it's "title keyword" `go`. Any function call preceded by the keyword will run in it's own *goroutione*, concurrent with further execution of the calling code. Goroutines can be thought of as "lightweight threads" [2], and are handled by an internal scheduler.

Message-based communication is built in via `channels`. Channels pass some data type in one or both directions. Both sending to and receiving from a channel can block, which provides simple means of synchronization.

Go code is modularized into *packages*, which can have sub-packages. Packages can be imported from some local source, or via some remote source over the Internet. For example, importing the matrix package `go.matrix` simply by the URL of its repository `github.com/skelterjohn/go.matrix` is allowed.

Go is *garbage collected*, meaning memory is freed automatically once it can no more be referenced from the runtime.

Go also has built-in features for package testing and benchmarking via code-declared tests and benchmarks, and automatic compilation of code documentation from code and comments.

4.2 Controllers

A central controller for the system was implemented for centralizing the coordination of the execution. The central controller structure `Controller` can be used to access several sub-controllers, like `SlamController`, `SensorController` and `MotorController`.

The central controller can be thought of as holding the global *state* of the system. Thus, it also includes an instance of `model.Robot`, a mathematical model of the robot currently used, for sub-controllers to leverage.

4.3 Sensor Module

The sensor module standardizes and centralizes control of sensors and sensor data, implemented in the `sensors` package.

The package includes a `SensorController` structure, which is used to provide a list of available sensors. It can also tell the status of the individual sensors, for example if they are connected or currently running (producing data).

All sensor drivers must implement a common `Sensor` interface, standardizing interaction. The interface specifies methods such as `Start()`, `Stop()`, `Connect()` and `Disconnect()`, along with some methods for acquiring sensor meta data and status. Sensor drivers are currently implemented as sub-packages of the `sensor` package; `lidar` and `odometry`. A structure `BasicSensor` provides common features from which sensor drivers can be built.

Sensors exploit Go's `channel` objects in order to distribute the data they acquire. Each sensor has methods `Subscribe()` and `Unsubscribe()`, where the first returns a channel and the second takes a channel as input and terminates distribution over that channel.

Sensor reading is initiated by the sensor itself, for example at regular intervals or whenever new sensor data is available. The measurement is distributed to all entities which have a subscription. If some subscribing entity has not received the previous sensor reading, the new sensor reading is discarded. Note that discarding happens on a per-subscription basis, so that one misbehaving entity does not block for the entire system.

The implemented LIDAR sensor driver uses the original *C* driver provided by the manufacturer. The C-code is compiled and produces libraries which the Go code links to via Go's `cgo` package.

The odometry driver communicates with the encoders via a serial port over USB.

The sensor module also includes two sub-packages `logging` and `logreader`, which deal with the storing and reading of *sensor log files*. Such files are specified to a *comma-separated values* (CSV) format, where each line begins with a sensor identifier and a time stamp. Note however, that sensors are free to specify an arbitrary number of values.

The `logging` package logs all sensor data in such files, in a sensor specific format provided by the sensors themselves through methods described by the `Sensor` interface. Sensors should encode their data such that the reading can be fully restored.

The `logreader` package is capable of re-constructing these sensor readings in a similar fashion. It can also *play back* sensor log files in real-time, so that a recorded scenario can be "re-lived". The rest of the system is agnostic to whether sensor readings are originated from the sensor itself or from the log reader.

These two packages are useful for development and tuning, because they allow for comparing the outcome of different SLAM algorithms, implementations and tuning parameters through near perfect repeatability.

4.4 Motor Module

The `motor` package was implemented for setting control signals to the connected motors. It includes a `MotorController` structure used for centralized communication.

Central to the package is a sub-package `driver`, which communicates with the motor controller card over a serial interface. However, the `motor` package also features a wrapper over the driver. This wrapper provides *easing* so that the actual control signals sent to the motor do not change too quickly. The easing is implemented as a P-controller, where commands are set as reference values and the P parameter is configurable.

The driver and wrapper provides a method `SetSpeeds()`, where the arguments are floating point numbers indicating relative speed settings in the interval $[-1, 1]$ for the left and right hand side motors. The motor controller card does not provide explicit setting of angular velocity, and though this could have been implemented through a tuning parameter, this was not regarded as important.

Due to frequent failures of the motor controller card under testing, the motor driver was implemented such that it automatically reconnects if connection is lost. Thus, the `SetSpeeds()` method can be called without explicitly initiating a connection.

The `MotorController` also features functions for initiating path following and path planning. A small package `collisionavoidance` was written. The `MotorController` instantiates a `CollisionDetector` object from that package, which provides channels communicating that some object is detected in some configurable sector in front of the robot. In path following mode, the `MotorController` responds by stopping the robot, backing up slightly and waiting a configurable amount of time. If the obstacle still persists, the `MotorController` will initiate planning a new path to the goal location.

4.5 SLAM Module

SLAM is a central part of this thesis, and much time and concern was devoted to the implementation of the SLAM parts of the software. A SLAM approach had to be chosen, from the rich body of approaches presented in literature of recent years.

The choice with the largest impact was to choose between landmark-based and grid map based SLAM. As discussed in Section 2.1, the choice has implications for other aspects of the system, with path planning being one of them.

The outcome of the decision was dependent on the problem description and the use cases of the system. For the robot to be controlled remotely, but manually, it is central to have maps which are easily interpretable by humans and which allow for effective and reliable path planning.

Landmark-based algorithms do not necessarily mark areas as free to be navigated, and are harder to visually interpret. Additionally, [18] regards grid maps as better suited for path planning.

A choice was therefore made to develop the system for grid map based algorithms.

Among the grid map based algorithms, the largest division is between Rao-Blackwellized particle filters and the simpler algorithms which only considers a single hypothesis. Points of consideration include demand for computational resources, suitability for navigation, code complexity and reliability.

Assuming maps should be of a resolution down to the error of the LIDAR, memory usage quickly arises as an issue. A map of 100×100 meters with a resolution of 1 cm has 10^8 cells and amounts to approximately 400 MB when using 32-bit representations. Rao-Blackwellized particle filter based methods hold many of these at a time, which means very high memory demands.

Solutions like DP-mapping could lower the demands, but comes at a cost of high code complexity and raises the learning curve for future work. Indeed, the open implementation of DP-SLAM was found to be of low quality and hard to understand.

Another complexity which comes from multi-hypothesis tracking has consequences for navigation. If the particle with the highest weight is always used as a position and map estimate, the position and map could change rapidly and cause inconsistencies with the planned path. A method could be to track one hypothesis while traversing, but the survival of the hypothesis in the particle filter cannot be guaranteed. Note that solutions to the problem would break the modularity of the code. In other words, a guidance module would have to consider the inner workings of the SLAM module.

Note also that several single-hypothesis algorithms could be extended to track several hypotheses. This is already mentioned for TinySLAM. A requirement is for them to be probabilistic, so that the different hypotheses do not stay exactly alike.

Although multi-hypothesis tracking gives advantages in scenarios like loop closing, and can overall be less prune to produce inconsistent maps, a decision was made to focus on single-hypothesis mapping for this thesis. The main reasons were complexity of code and modularity.

An interface `Slam` was written, which every implemented SLAM algorithm must fulfill. The approach allows for several SLAM algorithms to be interchangeable, and requires methods like `GetPosition()`, returning the current position, and `GetMapImage()` which returns an image representation of the current state of the map.

4.5.1 TinySLAM

In order to provide a starting point for initial tests, the TinySLAM algorithm was implemented for the system. As discussed in Subsection 2.8.1, the algorithm is small, easy to understand and works with grid maps. Its underlying principle is scan matching.

The original implementation [55] was translated to Go and fitted into the rest of the system as a SLAM module, fulfilling the `Slam` interface. A structure `TinySlam` holds the overall state of the SLAM process.

Support for odometry was not integrated in the Go implementation because odometric data was not available at the time.

The algorithm was tested with a data set from the *Intel Research Lab* in Seattle, available from `radish.sourceforge.net` [33]. The result can be seen in Figure 4.1.

In order to arrive at such a result, the algorithm had to be allowed to utilize 10 000 iterations for each scan matching. Even if odometry had been integrated, the number seemed too large for the algorithm to feasibly run in real-time.

The maps of TinySLAM have "blurry" walls, a result from the algorithm's scan matching approach further discussed in Subsection 2.8.1.

Seen in relation with the quality of the map, a decision was made to halt further development of the TinySLAM implementation, and look for

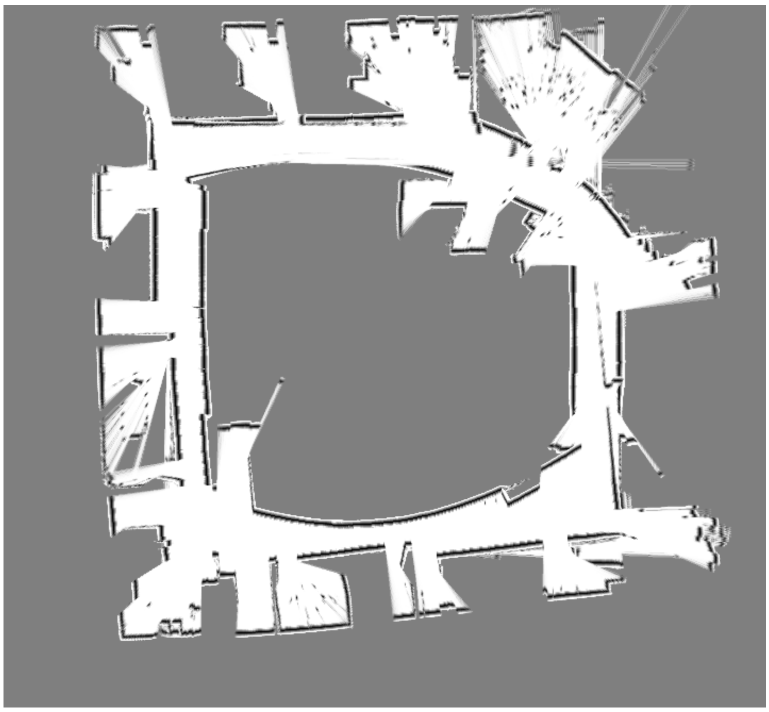


Figure 4.1: The Go port of the TinySLAM implementation, running an excerpt from a publicly available data set. The SLAM algorithm was allowed to use 10 000 iterations in each scan matching step, but did not produce a consistent output, as can be seen from the double wall to the right.

other approaches which could generate similar or better results with less computational power. The TinySLAM code is still integrated in the system, although some work has to be done in order to make it work with the rest of the system, which has been developed further.

4.5.2 Hector SLAM

As mentioned in Subsection 2.8.2, the SLAM approach introduced in [37] is available as open-source software at [36], in C++. Though the implementation is of considerable size, it was found to be easily understandable and maintainable. The implementation was highly modular, clearly defining the purpose of each module.

The system discussed in this thesis was implemented in Go. Schemes exist for compiling C++ with Go code [14]. However, they complicate interaction between code in the two languages, which makes it harder to improve on the implementation. Combining this with an assumption that the rest of the system could benefit from aspects of the Hector SLAM implementation, it was decided that the implementation was to be ported to Go.

Moreover, a decision was made to make the ported Hector SLAM implementation independent on all other code, in order to make it possible to release the code as a stand-alone project.

The original and ported implementations feature a modular implementation of grid maps. A `MapDimensionProperties` object lies at the core of each map, specifying the map's *cell length* (resolution) and dimensions in world and map coordinates.

Grid map cells implement an interface `Cell`, which is free to represent the cell by any means, providing the representation can be converted to a floating point number for comparison. They also specify `IsFree()` and `IsOccupied()` methods returning boolean values. Note that these methods are not opposite – both can be false at the same time, meaning the cell's occupancy is unknown.

All grid maps must implement the interface `GridMap`, which is done simply by extending the structure `GridMapBase`. Occupancy grid maps implement `OccGridMap`, where `GridMap` is a subset of the requirements.

Many occupancy grid maps can do this by extending the OccGridMapBase structure.

Finally, the standard Hector SLAM implementation uses OccGridMapLogOdds maps, which is an OccGridMap representing cells by the approximated *log odds* that the cell is occupied. The structure's implementation is small, and most of its functionality is inherited from the base structures – illustrating that other cell representations could have been introduced easily.

A feature was introduced in the port which is not present in the original C++ implementation, rendering PNG images from maps. The package mapimages creates three-color images, where cells are either free, occupied or unknown. The package can generate whole map images or generate tiles of 256×256 pixels representing a part of the map, with a specific *zoom level*. The latter is inspired by the *Google Maps JavaScript API* [25] and is useful for graphical user interfaces.

Hector SLAM has an interface MapRepresentation where an extra layer of abstraction is facilitated. The image pyramid-like functionality of the scheme explained in Subsection 2.8.2 lies here, with a structure MapRepMultiMap , which has a number of maps, matching each LIDAR scan to each of them.

During porting, a bug in the original C++ code was discovered in the multi-map representation, causing it not to utilize the full power of the scheme. The bug was confirmed by one of the co-authors of the original article [37] via e-mail. The bug involves how scan matching is performed with multi-layered maps, and was reported to the Hector SLAM project's bug tracker[1].

The HectorSlamProcessor structure represents the top layer of the Hector SLAM implementation, and is the structure which other code primarily should interact with. It is responsible for initializing and maintaining all substructures, builds and provides map data, and for obtaining current position estimates.

[1] https://code.google.com/p/tu-darmstadt-ros-pkg/issues/detail?id=4 Note that at the time of writing, no response has been given in the bug tracker.

4.5.3 Odometry Integration

As mentioned in Subsection 2.8.2, Hector SLAM in itself does not integrate odometric information. The algorithm only considers range measurements, typically from a LIDAR unit.

This is not to say that information from other sensors can't be used to improve its accuracy. Every scan matching step of the algorithm accepts a position estimate as input, acting as a start point for the matching procedure. A better initial estimation could result in more reliable scan matching.

The information on which a better estimate can be made include both the SLAM process itself and other sensors. The original article [37] presents a method using an inertial measurement unit (IMU), using an extended Kalman filter to integrate the measurements. This report considered only odometric sensors. From the SLAM process alone, an estimation can be obtained by considering the speed of the robot and its mathematical model.

An extended Kalman filter was implemented, with five states:

$$\boldsymbol{x} = \begin{bmatrix} x & y & \theta & v_l & v_r \end{bmatrix}^\top, \tag{4.1}$$

where the first three states represent the robot's position, and the latter two are ground speeds of the left and right odometry wheel, respectively. The filter is updated by scan matching from the SLAM process and from odometric measurements.

Since the measurements cannot be assumed to arrive in a synchronized manner, some extra precautions were taken. When propagating the previous state, v_l and v_r are considered in order to predict the current position, along with the difference in time from the last update Δt. The time stamp from the sensor reading is used in order to provide this time difference.

For odometry updates, a mathematical model of the robot is used in order to generate an estimation of the full state, by dead reckoning based on the last update.

Updates from the SLAM process come in as estimates of the robot's pose $[x, y, \theta]^\top$. Rough estimates of the left and right wheel speeds are computed to obtain a full state estimate.

The update step discriminates between updates from the two different sources by manipulating the computed Kalman gain matrix $\boldsymbol{K}$. For the odometry update not to affect the covariance error estimates of the position, the upper three lines are set to zero. Likewise for the SLAM position updates, the lower two lines are set to zero.

When a new LIDAR measurement arrives, triggering a SLAM iteration, a position estimate is fetched from the Kalman filter, propagating the position based on estimates v_l and v_r considering the time since last filter update.

The original Hector SLAM article [37] cites another way to integrate SLAM position updates in such a Kalman filter, using *covariance intersection.* The method is thoroughly described in [34]. The purpose is to prevent overconfident estimates. This method was also implemented, but without success. Sufficient documentation could not be found. The resulting error covariance matrix was often ill defined and SLAM results were poor.

Prior to implementing the Kalman filter, a much simpler approach was taken, only using any odometry arriving between the SLAM updates to propagate the last position estimate. While successful, the approach seems less rigorous.

4.5.4 LIDAR Scan Correction

As specified in Chapter 3, the LIDAR used produces scans at 10 Hz. Note however that the scans are not instantaneous, the individual distance measurements of the scan are not taken at the exact same point in time.

No specific data on this concerns could be found in the LIDAR's specifications [32], other than that the laser rotates in counter-clockwise direction. Assuming it maintains a constant speed, one scan is taken over a duration of $240°/360° \times 0.1 \approx 0.067$ seconds. This means the robot could have travelled approximately 6.7 centimeters between measurements in an individual scan, assuming a speed of 1 m/s.

A scheme for correcting for this effect was constructed based estimated states from the EKF described in the previous subsection. The important states are v_l and v_r, from which both the angular speed $\dot{\theta}$ and the translation speed v can be estimated as

$$\begin{aligned} \dot{\theta} &= \frac{v_r - v_l}{b} && (4.2) \\ v &= \frac{v_r + v_l}{2}, && (4.3) \end{aligned}$$

where b is the base width of the robot. Thus, the corrections are implemented as

$$\begin{aligned} x_i^c &= x_i^p \cos(d_i\dot{\theta}) - y_i^p \sin(d_i\dot{\theta}) && (4.4) \\ y_i^c &= x_i^p \sin(d_i\dot{\theta}) + y_i^p \cos(d_i\dot{\theta}) + d_i v, && (4.5) \end{aligned}$$

where d_i is the delay which should be imposed on measurement $\boldsymbol{x}_i^p$ in order to obtain the corrected measurement $\boldsymbol{x}_i^c$.

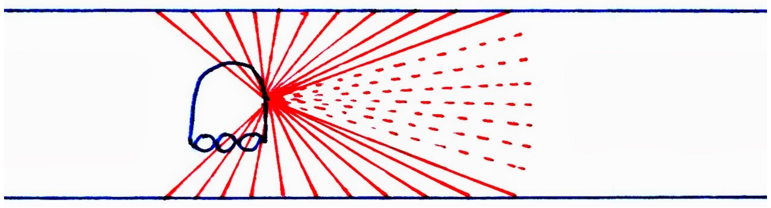
Correcting the measurements can be turned on and off from a configurable parameter.

4.5.5 Unknown Map Triangles

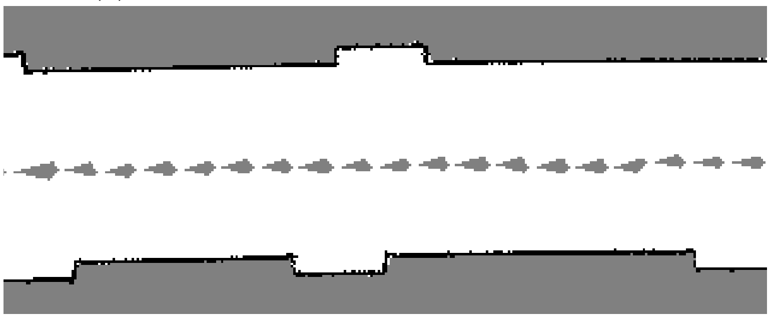
An artifact of the SLAM implementation needs mentioning. When mapping only with a single pass of the environment, triangles or rhombi (four-sided shapes) of unknown cells occur as a trace of the robot's path. They occur in situations where some of the front-faced beams of the LIDAR experience maximum range. They can look confusing, but have a natural explanation.

To understand where the shapes arise, consider the fact that the maps are only updated after a movement of a configurable rotation or translation, for example configured to 0.4 radians or 0.4 meters. When moving along a corridor, the maps are then updated every 0.4 meters.

At the same time, the LIDAR beams facing forward experience maximum reading because they cannot reach the end of the corridor. This means each update of the map has a triangle of unknown area between the farthest measurement on each side of the corridor, and the LIDAR unit itself. See Figure 4.2.



(a) Robot and LIDAR beams in corridor scenario.



(b) Resulting map.

Figure 4.2: 4.2a shows the underlying problem: A number of front-facing LIDAR beams (red) return maximum readings (dotted). The resulting map has triangles or rhombic shapes, as seen in 4.2b.

4.6 Map Representations

After implementing Hector SLAM, a decision was made to reuse the code for grid map representations throughout the rest of the program. As previously mentioned, the map representation code was highly modular, and contains elements general enough to allow for different cell representations. Indeed, a the path planning algorithms use a binary map representation, discussed in Subsection 4.8.1.

As an example of the modularity, the image pyramid-like approach Hector SLAM has is in fact “hidden” in an object `MapRepMultiMap`, which satisifies the `MapRepresentation` interface. A representation using only one such map layer is also implemented, `MapRepSingleMap`, which also satisifies the same interface. The rest of the algorithm is agnostic to which of them is used, and to what the underlying cell representation is.

It is important to note that this `MapRepresentation` interface is agnostic to the scan matching approach used. The only asumption made is that some scan matching algorithm is available, returning an estimated pose based on some start estimate and some range scan.

For this reason, the author can see no reason why for instance the map representation and scan matching of TinySLAM couldn't be implemented within this map representation paradigm. Much of the TinySLAM algorithm could have been implemented simply as a `MapRepresentation` object, encompassing both the underlying likelihood field representation and the scan matching procedure.

4.7 Map Storage

In order to save maps and load previously obtained maps, a package `mapstorage` was implemented. The package has a corresponding assets folder where all maps are stored, and is responsible both for storing maps in a suitable way, and for reconstructing them from a file identifier.

In order to safely encode and decode maps, the `encoding/gob` package of Go was used. The package helps with encoding arbitrary Go structures to binary code [2], which can be transmitted or stored, before they can be decoded. The package can be compared with Python's `pickle` [24] module, for readers familiar with Python.

In order to safely use `gob` encoding for arbitrary maps, map representations have the possibility to implement the `GobEncoder` and `GobDecoder` interfaces by specifying their own `GobEncode([]byte)error` and `GobDecode()([]byte, error)` methods. The map storage package was implemented so that these methods are honored.

A file type was specified for the maps, where the map data is encoded to binary and stored together with

- a `MapMetaData` object containing
  - Map name
  - Map description
  - Map dimension properties (for quick access)
  - Map type, and
- a map thumbnail image of 256×256 pixels in PNG format.

The `MapMetaData` object is `gob` encoded. The PNG thumbnail makes it possible to get a glimpse of the map without decoding and loading the full map data.

As the maps can be of considerable size, the three entities are compressed together in a `zip` archive. This reduces the disc size of stored maps dramatically: a map of 4096×4096 cells can take up 110 MB uncompressed, but the finished file might be no larger than 1.2 MB.

The `mapstorage` package provides functions `Save()`, `Load()`, `Rename()`, `Copy()` and `Delete()`, `LoadMapMetaData()` and `LoadMapThumbnail()` for use by the rest of the program, in addition to a function returning a list of available maps.

4.8 Path Planning

Path planning is crucial for autonomous navigation. Given start and end points in a map, the system should be able to plan a suitable path, which

- is near optimal in the sense that it chooses a path reasonably close to the shortest available path between the points,

- avoids known obstacles and walls,
- sticks to areas known to be free, avoiding unknown areas when possible,
- circumnavigates obstacles at a set distance, holding some minimum distance to walls.

Additionally, the path planning algorithm should run on-line, so it should be able to plan paths quickly.

4.8.1 Binary Maps

The path planning algorithm was to use maps from the SLAM algorithm, that is, grid maps with resolutions down to 2.5 centimeters or less. This is a level of accuracy not needed for path planning, which was assumed to have more benefit of speed than of excessive accuracy.

Therefore, a *binary grid map* package `binarymap` was created, where, as the name suggests, cells are either free or occupied. The binary maps implement the same `OccGridMap` interface as the SLAM output maps.

A function was written in order to create these binary maps directly from any occupancy grid map, supporting an argument for scaling the maps down by some factor, so that down-sized maps could be created easily. The function defines a cell in the down-sized binary map as occupied if any of the cells in the original map are occupied.

Additionally, the down-scaling function considers a parameter `checkRadius` , which is the additional distance from any obstacle in the original map which should be considered occupied. This is done in order to help search algorithms avoid obstacles at some distance.

The result is a map taking less space in memory, which is easier to handle and easier to use for a variety of search algorithms. A package `pathplanning` was implemented, featuring an `PathPlanner` interface which arbitrary path planning algorithms can implement. All such algorithms must return a `Path` object as specified in the sub-package `path`. Such a path consists of line segments represented by their start and end coordinates.

4.8.2 A* Path Planning

Path planning using the A* search algorithm is perhaps the most straight forward approach. The well-known algorithm is easy to implement and maintain. A* was implemented as a package `astar`.

Each cell of the binary map is considered a state, with consideration of the robot's orientation. In other words, the implementation has a 2-D vehicle state representation. This was assumed not to be a problem, since the robot can turn on a dime, that is, rotate around it's approximate center.

The algorithm searches the binary map for a path to the goal location. The heuristic is the Euclidian distance to the goal location, multiplied by some configurable factor if the corresponding cell of the original map is unknown, in order to punish paths through such locations. This heuristic is *admissible*, thus guaranteeing optimal paths with respect the binary map and tuning parameters.

One of the drawbacks of the plain A* algorithm is its discrete paths, leading to zig-zag patterns instead of diagonal lines, that is, it artificially constrains the path to move in only four directions (eight could easily have been implemented). An example can be seen in Figure 4.3a.

4.8.3 Hybrid A* Path Planning

Attention was given to find the fastest path planning algorithm possible, which would allow for more frequent re-planning. An article about *Junior*, the Stanford entry in the 2007 *DARPA Urban Challenge* presents an approach to high speed path planning which they call *Hybrid A** [42]. Junior is an autonomous car, based on a Volkswagen Passat.

The algorithm is given its name because it plans a continuous, smooth path in a discrete grid map. The vehicle state is represented by 4-D discrete grid, where the robot pose is three of the dimensions, with a forth representing the direction of motion: forward or reverse.

The hybrid A* algorithm implemented for Junior features two heuristics, both admissible. The first one considers the non-holonomic constraints of the car, and can be completely precomputed for the entire 4-D space, helping by approaching the goal with the correct heading. The second is calculated online by dynamic programming, indicating the shortest path given obstacles.

The algorithm's nodes are still the cells of the grid map. However, the transitions from one cell to another are computed by control signals and their predicted effect on the vehicles state through some mathematical model. Though the nodes are discrete, they are associated with continuous states. The effect is that a control signal for the path can be guaranteed to exist.

A package `hybridastar` was implemented, performing an approximation of the algorithm described above. The non-holonomic heuristic was discarded, under the assumption that it would have a lesser impact for the application of this thesis – as previously mentioned, the robot is approximately differential wheeled and can turn on a dime, whereas Junior has Ackermann steering like a regular car. Also, instead of using dynamical programming, the Euclidian distance was used as the heuristic.

Four motions were used as transitions: directly forward, forward left turn, forward right turn and directly backward. An underlying problem is that the algorithm with Euclidian distance heuristic does respect in-place turning, that is, turning without moving forward or backward, as these actions do not impose a change in the heuristic.

The result was a working path planning algorithm which produced smooth curves, but which was suboptimal in the sense that it did not honor the maneuvers which the approximate differential wheeled robot was capable of.

4.8.4 Path Smoothing

Whereas Hybrid A* paths are smooth, A* paths can contain zig-zag patterns, leading to inefficient motions of the robot, with frequent turns. Other frequently used path planning algorithms like *Field D** also posses this flaw [42, 21], although to a lesser extent.

A path smoothing method was therefore implemented in the `path` package. The used is presented in *Unit 5* of the course *Artificial Intelligence for Robotics* at `Udacity.org`, and is a gradient descent optimization which smoothen paths through a weighting of two goals: staying close to the orig-

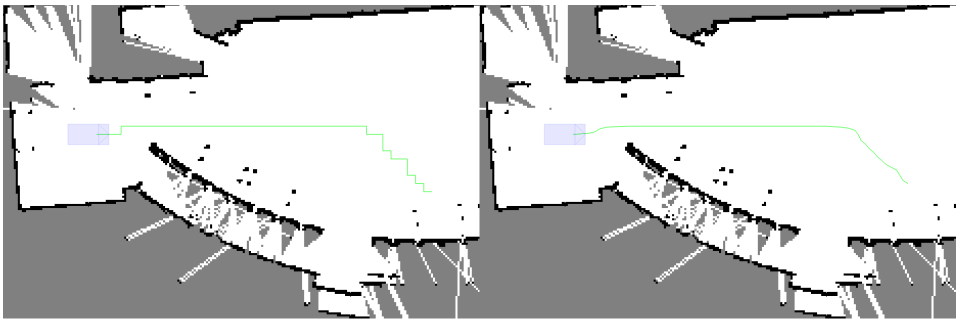

(a) Non-smoothened path **(b)** Smoothened path

Figure 4.3: Non-smoothened and smoothened paths planned in auditorium *EL5* at NTNU. Both paths are planned by the A* algorithm implemented as described above. The figures are screen shots from the web interface, described in Section 4.11.

inal path and minimizing the length of the path.

The algorithm starts off by assigning a new path, which is a copy of the path to be smoothened, $\boldsymbol{Y} = \boldsymbol{X}$. The paths are defined by locations $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ of nodes in some coordinate system.

The two optimization criteria are defined as

$$||\boldsymbol{x}_i - \boldsymbol{y}_i|| \to 0, \tag{4.6}$$

minimizing the distance from the new point $\boldsymbol{y}$ to the original point $\boldsymbol{x}$, and

$$||\boldsymbol{y}_i - \boldsymbol{y}_{i+1}|| \to 0, \tag{4.7}$$

minimizing the distance of consecutive points of the new path.

The algorithm is implemented as the two equations

$$\boldsymbol{y}_i^{k+1} = \boldsymbol{y}_i^k + \alpha(\boldsymbol{x}_i - \boldsymbol{y}_i^k) \tag{4.8}$$

$$\boldsymbol{y}_i^{k+1} = \boldsymbol{y}_i^k + \beta(\boldsymbol{y}_{i+1}^k + \boldsymbol{y}_{i-1}^k - 2\boldsymbol{y}_i^k), \tag{4.9}$$

where α and β represent weights for smoothing and for remaining true to the original path, respectively. The update equations are executed iteratively for all path nodes except the first and the last, until the path converges.

An example of the application of the smoothing procedure can be seen in Figure 4.3.

4.8.5 Result

The implementation ended up with two different alternatives to path planning. The Hybrid A* algorithm produces smooth paths, while the A* algorithm produces paths which need to be smoothened. More path planning algorithms can easily be added at a later point, perhaps taking advantage of both the binary maps and the path smoothing.

Although the path smoothing can theoretically cut corners around so that the resulting path is invalid, the effect did not turn out to be a problem in initial tests.

4.9 Path Following

Path following was implemented in order to control the robot along the paths planned by the modules discussed in Section 4.8. The problem was to, given a pose estimate from the SLAM algorithm, find control signals for the motors in order to follow a given path.

A package `pathfollowing` was created, containing a `PathFollower` interface which any such algorithm can implement, only demanding the methods `SetPath()` and `SpeedUpdate()`, where the latter returns control signals given some position estimate. This allows other path following strategies than the following to be implemented.

A method for solving this problem was obtained from [23], where the application is path following for marine craft. The algorithm is referred to as *line of sight with lookahead-based steering.*

The algorithm considers line segments one by one, starting with the first. It recognizes when the robot has passed this segment, and continues with the next until the whole path is completed.

While a more comprehensive derivation can be found in [23], this text seeks to provide an intuitive explanation.

A transformation matrix to a line segment fixed reference frame $\boldsymbol{R}_p(\alpha_k) \in$

$SO(2)$ is constructed by the angle angle α_k between the line segment and the x axis. From this, the coordinates $\boldsymbol{\varepsilon}(t) = \boldsymbol{R}_p(\alpha_k)^\top(\boldsymbol{p}^n(t) - \boldsymbol{p}_k^n)$ of the robot in line segment fixed reference frame is computed.

The coordinates $\boldsymbol{\varepsilon}$ now consist of

$$\boldsymbol{\varepsilon} = \begin{bmatrix} s(t) \\ e(t) \end{bmatrix}, \tag{4.10}$$

where

$$s(t) = \text{along-track distance}$$

and

$$e(t) = \text{cross-track error.}$$

The objective of path following is thus simplified into the problem of controlling $e(t) \to 0$. This is done by computing some desired heading direction $\chi_d(e)$ based on the cross-track error. The desired heading consists of two parts

$$\chi_d(e) = \chi_p + \chi_r(e), \tag{4.11}$$

where χ_p is identical to the angle used for computing the transformation matrix $\boldsymbol{R}_p(\alpha_k)$. The angle $\chi_r(e)$ is computed as

$$\chi_r(e) = \arctan\left(\tfrac{-e}{\Delta}\right), \tag{4.12}$$

where Δ is some configurable *lookahead distance* along the line segment, ahead of the projection of the current location into the line segment.

Whenever the along-track distance $s(t)$ of the current position estimate for a line segment is found to be larger than the length of the line segment, the line segment is swapped with the next.

With the algorithm's output being a desired heading direction χ_d, an additional derivation had to be done to find appropriate control signals. The derivation considers the following equation, suitable for differential wheeled robots:

$$\dot{\theta} = \frac{v_r - v_l}{b}, \tag{4.13}$$

where v_r and v_l are speeds of each wheel, and b is the base width of the robot. We want a P-controller for the heading,

$$\dot{\theta} = p'(\chi_d - \theta(t)). \tag{4.14}$$

Combining equations 4.13 and 4.14 yields

$$v_r - v_l = p(\chi_d - \theta(t)), \tag{4.15}$$

where $p = p' \cdot b$ is the controller's P parameter. The control signals are thus given as

$$u_r = u_c + \frac{v_r - v_l}{2} \tag{4.16}$$
$$u_l = u_c - \frac{v_r - v_l}{2}, \tag{4.17}$$

where u_c is a constant ensuring the robot maintains a forward velocity – setting $u_c = 0$ makes the robot only turn in place.

4.10 Configuration

The total software system features many run-time constants which could change over time and which are subject to tuning. Some are dependent on the environment of the computer. Examples are the location of directories, the COM ports on which sensors and motors are found and tuning parameters for SLAM algorithms.

Allowing such parameters to be configured without needing to rebuild the program executable would make the program more flexible.

Instead of writing such a module from scratch, an open source package was used. *Goconf*, available from `code.google.com/p/goconf` reads and writes to easily human-readable text files. It abstracts the process of reading a configuration parameter to one function call.

A very simple package `config` was written in order to provide the configuration values to the rest of the system. The package defines one variable for each of the configuration parameters, and populates them when the program is started. The parameters are available at runtime for the rest of the program simply by importing the package and referring to the appropriate variable, for example `config.EXAMPLE_VARIABLE`.

4.11 Web Interface

The problem description specifies that some interface should be available via a wireless connection. The interface should offer services including manual control of the robot, inspection of map data and estimates of the robot's current position, in addition to automatic control and guidance.

While the interface could have been built as a client side native application with a graphical user interface, a decision was quickly made to implement it as a *JavaScript* driven *web interface*. Success with this way of providing an interface had previously been shown by the NTNU Eurobot team [50].

The advantages of using web pages for control are numerous. Web pages with JavaScript are platform independent, and can be accessed from smartphones, tablet computers, laptops, desktops and even game consoles. They require no installation on the client side.

Many functions of the web pages should be updated without refreshing the page on the client side, thus sparking the need of a server side API for obtaining new data. The side effect is an opportunity of letting other applications access the same interface, opening for creating new applications strictly client side, enabling more use cases of the software in the future.

4.11.1 Server Side

A package `web` was created based on the `net/http` package included in Go. The package was inspired by code from the NTNU-Eurobot project. A structure `WebServer` was used to allow communication with the rest of the program, and functions both for serving *static* files such as CSS or JavaScript and for serving dynamic pages where some of the content is dependent on data from the rest of the program was written. The dynamic pages were implemented using the `html/template` package of Go.

Additionally, a framework for API functions was set up. It was constructed so that its functions could access the central `Controller` of the program, with modules such as motor control, SLAM control and sensor control. The functions are free to return different status codes and arbitrary data when relevant, such as images. However, most API functions would return data serialized with the *JSON* format, easily readable both for human inspection and for many different programming languages.

Documentation of the implemented API calls can be found in Appendix B.

4.11.2 Client side

The client side file templates and static files are stored in the `assets/web` folder.

The interface was implemented based on the *Twitter Bootstrap* framework available at `twitter.github.io/bootstrap`, in addition to the *jQuery* JavaScript library.

Twitter Bootstrap was built by *Twitter*, the social networking service. It provides simple solutions to many common problems in web development, such as fluid layouts which adapt to different screen resolutions. This made making the interface work nicely with both desktop computers and smartphones.

jQuery is currently the most popular JavaScript library [64]. Among its chief advantages is a simple CSS selector based way of addressing elements, its simple AJAX API for asynchronously fetching data from servers and a framework for building *jQuery plugins*, which can help code encapsulation.

Log Streaming

The interface of the NTNU Eurobot team includes log view, displaying output from the program's internal logs. Log messages can be useful for debugging and for documentation of events during execution.

While the Eurobot team let the client side continuously poll the server for changes of the log, an improvement was made in order to reduce the latency of log updates and the number of requests to the server. The jQuery plugin `jquery.websocket-0.0.1.js` was used in order to create a streaming log service based on *web socket* technologies. The server's log messages could thus be displayed in near real time, transferred through a single, open connection between the server and the client.

At the server side, the solution for maintaining such a connection was implemented through the use of the `go.net/websocket` package available from `code.google.com/p/go.net/websocket`.

SLAM Interface

One of the major components of the client side is the *SLAM graphical user interface*, pictured in Figure 4.4.

In order to display interactive maps generated by the system, drawing traces of the robot's position and the paths planned, the *Google Maps JavaScript API* [25] was used. The API has a wealth of functionality, and includes functions for drawing polygons and lines.

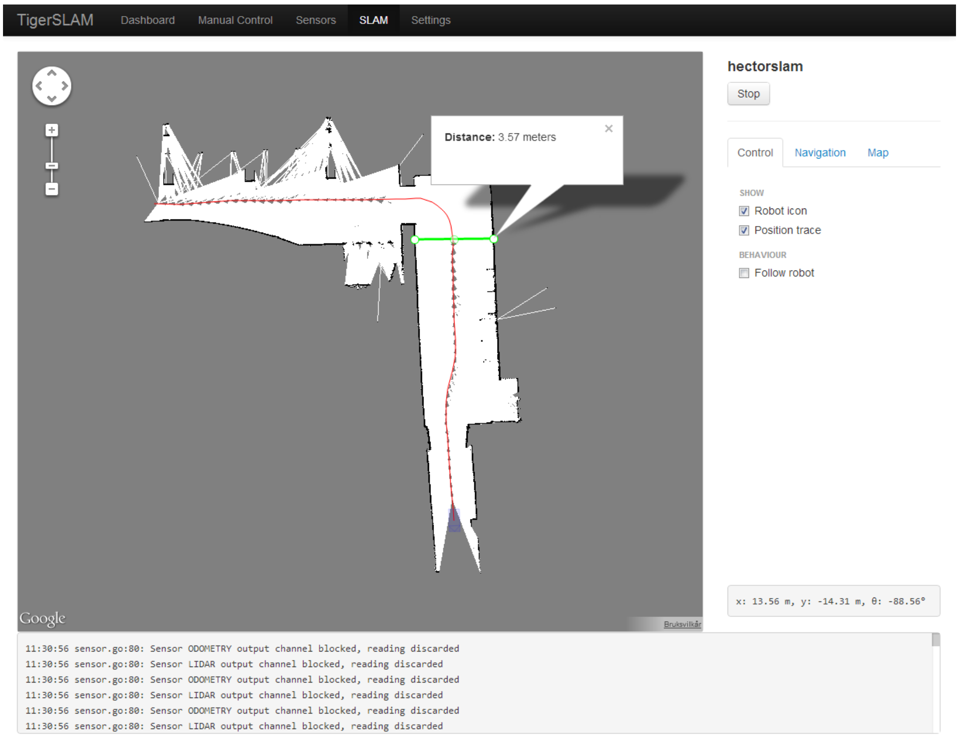
Some work had to be done in order to make the Google Maps API work with Euclidian maps like the output of the SLAM algorithms – the standard maps work with a *Mercator projection* suitable for maps of spherical objects like planets. A map projection for planar maps was implemented in order to support the planar maps of the SLAM algorithm.

Adjustments were also made in order to support dynamical updates of the map, where the current map is replaced with one loaded from the server. This is done by giving each tile it's own ID, allowing the image source address of the element to be replaced. The new `ImageMapType` has a method `update()` which automatically updates all tiles presently in the map. The code for the map adjustments are found in the file `googlemaps.slammap.js`.

Maps displayed in the client side interface are transferred from the server as *tiles*, all measuring 256×256 pixels encoded in PNG format. Tiles are associated with a specific *zoom level*. More details are explained in [25].

The Google Maps API's line and polygon drawing functions was used for drawing robot position, position trace and planned paths.

All other SLAM interface client side code is found in `jquery.slamstats`



```
11:30:56 sensor.go:80: Sensor ODOMETRY output channel blocked, reading discarded
11:30:56 sensor.go:80: Sensor LIDAR output channel blocked, reading discarded
11:30:56 sensor.go:80: Sensor ODOMETRY output channel blocked, reading discarded
11:30:56 sensor.go:80: Sensor LIDAR output channel blocked, reading discarded
11:30:56 sensor.go:80: Sensor ODOMETRY output channel blocked, reading discarded
11:30:56 sensor.go:80: Sensor LIDAR output channel blocked, reading discarded
```

Figure 4.4: SLAM graphical user interface as seen on a desktop computer. The Google Maps JavaScript API is used for displaying a map, overlaying the robot's current position (semi-transparent blue), a trace of the robot's positions (red line), providing a tool for measuring real world distances in the map in addition to visualizing planned paths (not pictured). A console log is also included in the interface.

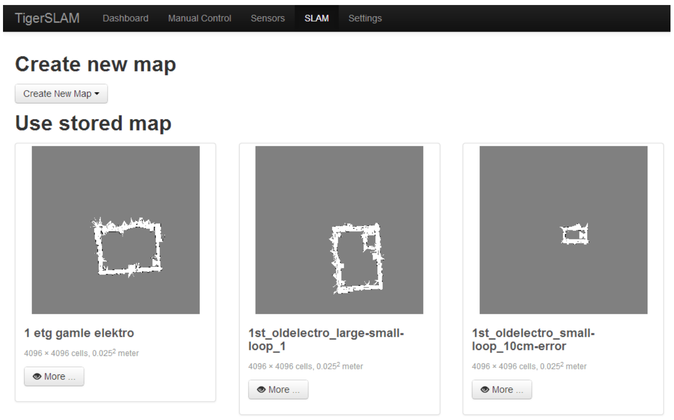

TigerSLAM
Dashboard
Manual Control
Sensors
SLAM
Settings
Create new map
Create New Map ▾
Use stored map
1 etg gamle elektro
4096 × 4096 cells, 0.025² meter
More ...
1st_oldelectro_large-small-loop_1
4096 × 4096 cells, 0.025² meter
More ...
1st_oldelectro_small-loop_10cm-error
4096 × 4096 cells, 0.025² meter
More ...

Figure 4.5: Map manager GUI. The view is used to display and edit map meta data, as well as loading them into the SLAM algorithm.

`.js`. As the name suggests, the code was written as a jQuery plugin, following a pattern in order to provide options for configuration and to let other code control the plugin. This plugin can start and stop the SLAM algorithm. It also provides means of requesting path planning and following of planned paths, controls what should be displayed in the map, while also continuously polling the server for changes. The plugin is responsible for automatically triggering map updates when the robot has moved a configurable distance, and for updating the robot's current position, updating planned paths and traces of past positions.

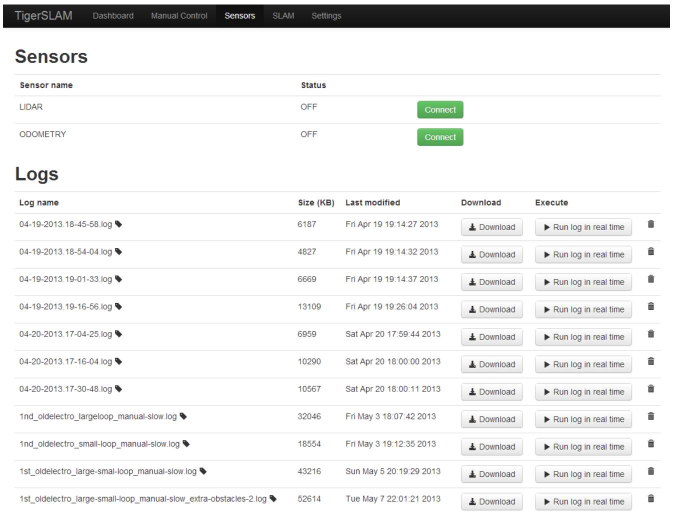
Map Storage Manager

An interface for inspecting and editing meta data of stored maps was created. The interface is also used to load such maps into the SLAM algorithm. A screen shot can be seen in Figure 4.5.

The interface loads thumbnail images and meta data of the maps from the API.

Sensor Control

A sensor control view was implemented, displaying the statuses of the LIDAR and odometry sensors available. The interface can be used to start



Sensors

Sensor name	Status	
LIDAR	OFF	Connect
ODOMETRY	OFF	Connect

Logs

Log name	Size (KB)	Last modified	Download	Execute	
04-19-2013.18-45-58.log	6187	Fri Apr 19 19:14:27 2013	Download	▶ Run log in real time	
04-19-2013.18-54-04.log	4827	Fri Apr 19 19:14:32 2013	Download	▶ Run log in real time	
04-19-2013.19-01-33.log	6669	Fri Apr 19 19:14:37 2013	Download	▶ Run log in real time	
04-19-2013.19-16-56.log	13109	Fri Apr 19 19:26:04 2013	Download	▶ Run log in real time	
04-20-2013.17-04-25.log	6959	Sat Apr 20 17:59:44 2013	Download	▶ Run log in real time	
04-20-2013.17-16-04.log	10290	Sat Apr 20 18:00:00 2013	Download	▶ Run log in real time	
04-20-2013.17-30-48.log	10567	Sat Apr 20 18:00:11 2013	Download	▶ Run log in real time	
1nd_oldelectro_largeloop_manual-slow.log	32046	Fri May 3 18:07:42 2013	Download	▶ Run log in real time	
1nd_oldelectro_small-loop_manual-slow.log	18554	Fri May 3 19:12:35 2013	Download	▶ Run log in real time	
1st_oldelectro_large-smal-loop_manual-slow.log	43216	Sun May 5 20:19:29 2013	Download	▶ Run log in real time	
1st_oldelectro_large-small-loop_manual-slow_extra-obstacles-2.log	52614	Tue May 7 22:01:21 2013	Download	▶ Run log in real time	

Figure 4.6: Sensor GUI listing the statuses of available sensors and sensor logs.

and stop the sensors, and also provides a list of the saved sensor logs.

The listed sensor logs can be renamed, deleted or downloaded. Each sensor log also has a button for initiating real time playback.

Manual Control

The web interface features a page for manual control. A jQuery plug-in named `joystick-js` provides a virtual joystick. Values are read from the joystick at regular intervals and sent to the API. The interface monitors the status codes sent back from the API and warns the user if errors have occurred. A screen shot from the graphical user interface can be seen in Figure 4.7.

4.12 Summary

A overall sketch of the program is sketched in Figure 4.8. The sketch is not accurate in details, but gives an overlook of the architecture. The controller with subcontrollers act as the backbone of the program, with subfunctions. The web interface sends commands to the controller, and can obtain data from controller and subcontrollers.

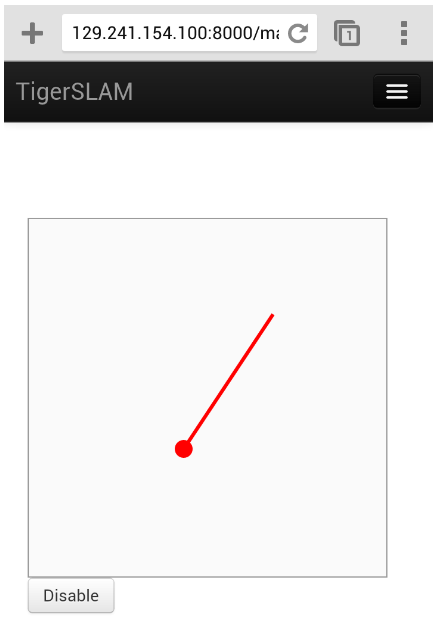


Figure 4.7: Manual control interface as seen from a smartphone. Currently steering forward and to the right.

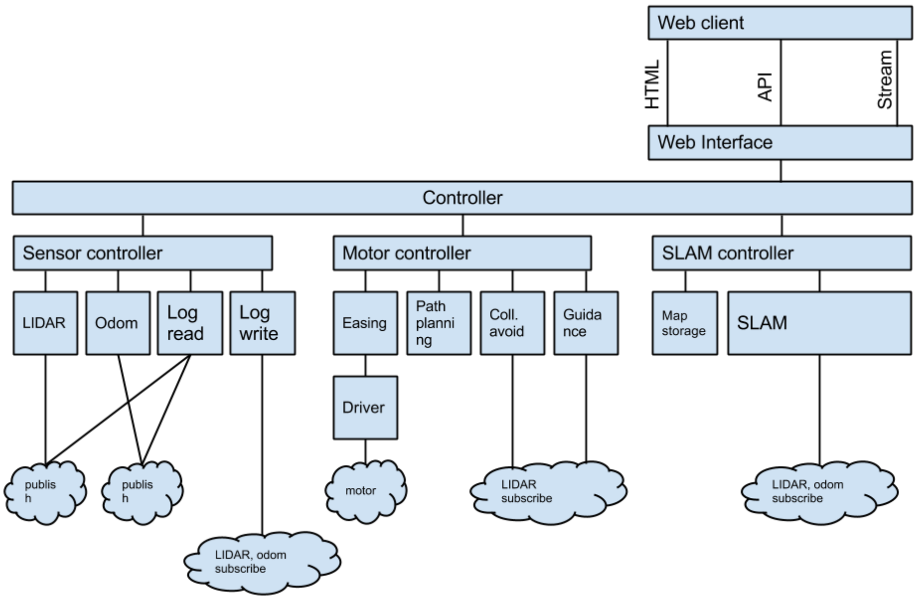


Figure 4.8: Software architecture sketch.

All sensor data is published as messages over channels. The log reader is also capable of publishing sensor data.

Chapter 5

Results

In order to demonstrate the software described in Chapter 4 in combination with the robot described in Chapter 1 and the sensors from Chapter 3, a series of experiments were planned and conducted.

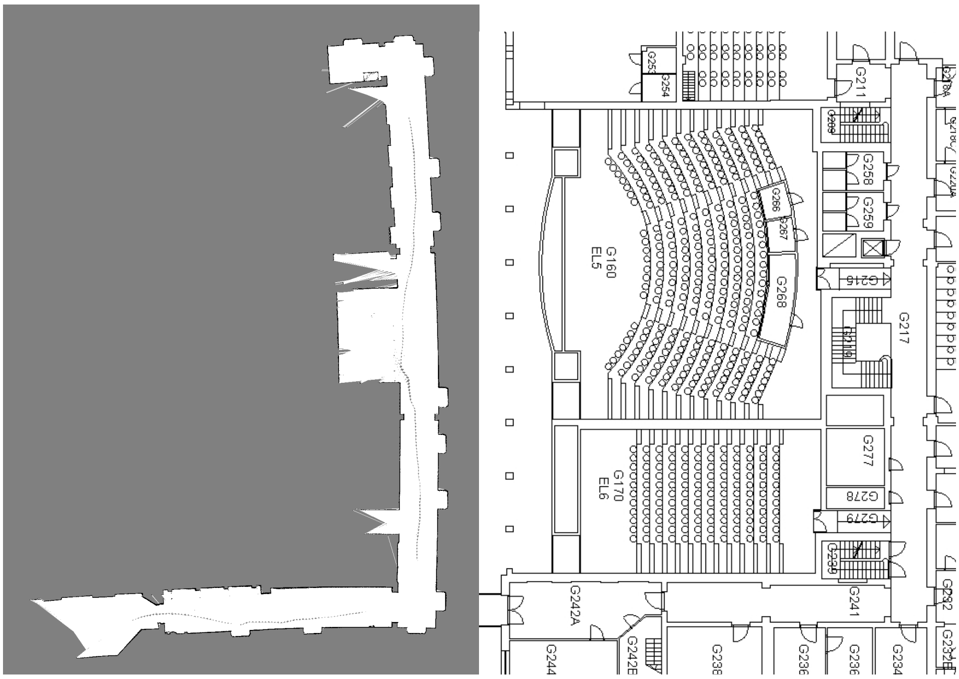
This chapter describes the experiments and their motivations, before presenting their outcomes. A short discussion of the results follow immediately after each result, while an overall discussion is found in Chapter 6.

When controlling the robot on-line, the robot's on-board computer was wirelessly connected to the building's wireless network, which has several access points. The web interface was available over the Internet. The robot was followed by an operator with a laptop computer.

5.1 On-Line Mapping

While the software is able to perform SLAM in several fashions, with manual control, automatic path planning and re-planning, off-line and on-line, one of the most useful basic skills it can show is it's ability to passively on-line create a map of the environment in which it is maneuvered.

An application could be assistance for manual control when navigating to some location. The robot could be controlled by the web interface described in Chapter 4, or it could be controlled by some other interface. The operator could be walking behind the robot, or control the robot from some remote



(a) Map obtained by robot, single pass. **(b)** Floor plan.

Figure 5.1: 5.1a shows a map obtained by the robot, on-line in real time. The map was obtained on-line during a single passing of the environment. The robot was controlled manually, via the web interface, during the experiment. 5.1b shows a floor plan of the same area, courtesy of NTNU (`ntnu.no/kart`.)

location via the Internet.

The experiment described below was designed to explicitly show the fulfillment of point 3c of the thesis' problem description, as well as demonstrate the abilities described above.

5.1.1 Execution

The robot was started from a repeatable location, accurate to within a few centimeters. The start position will become the map's origin in both x, y and θ coordinates. The location corresponds to the lower left corner of the map in Figure 5.1a, in room G242A of the floor plan 5.1b.

The LIDAR and odometry sensors were connected and started, initiating sensor logging. Then, the SLAM algorithm was started, running on-line in real time, before manual control was initiated. The robot was navigated to room G211 in the upper right corner of the map.

At one point during navigation, the operator lost control over the robot, as the program lost communication with the motor controller card. While the program sent correct control messages, the motors continued at a constant speed. The operator had to disconnect the motor card and re-plug it in order to regain control. After this incident, the experiment carried on without further disturbances.

5.1.2 Discussion

The experiment replicates a real-world scenario where the robot is controlled manually, while simultaneously creating and displaying a map along with its estimated current position.

While some inaccuracies occur, the map is found to be mostly accurate. When comparing the obtained map to the floor plans of the building, all distinctive features such as door openings and hallways are mapped. Their relations are accurate enough for comparison with floor plan.

The corridors of the map are slightly bent. The building is old and could have some constructional inaccuracies, but blaming the bending on the construction seems far-fetched. When drawing a line from one end of the longest corridor to the other, the distance from the wall to the midpoint of this line is 0.55 meters.

The scheme for correcting LIDAR scans from effects of delay between individual beams was investigated for errors. However, disabling this correction did not seem to have much effect.

A more likely explanation could be found considering the end-point densities. The small triangles of unknown spaces discussed in Subsection 4.5.5 indicate the robot's path when acquiring the map. A trend can be noticed that the corridors seems to bend towards the side of the corridor on which the robot was driving.

Considering the geometry of the LIDAR beams, the density of the endpoints will fall more closely on the closest side of the corridor than on the farthest side. This could lead one to suspect the scan matching procedure

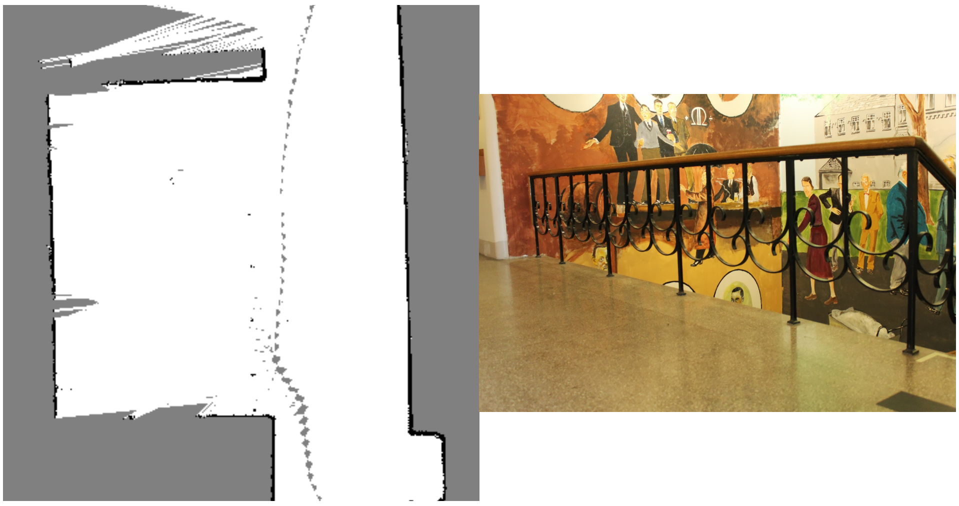


Figure 5.2: Representation of railing in the map, and a photograph of the scene.

could have a tendency to lean towards this closest side.

Also featured in the map is a hall in the corridor, with a staircase and railing. A magnified view of the area of the map is figured in Figure 5.2 along with a picture of the railing. Notice that the the railing is not accurately mapped. Five posts are mapped, but the photograph shows the correct number of railing posts is six. This shows such thin obstacles may not be accurately mapped, and might not be usable for the scan matching. However, in automatic drive modes, the robot will not crash into them, as they are recognized by the collision avoidance module.

Although the map has these flaws, the overall map quality is satisfiable. The map is consistent and features are placed in recognizable locations. Their spatial relationships seem consistent. See also the experiment comparing distances in the map to physical measurements, in Section 5.5.

5.2 Odometry Only

Assessing the accuracy of the odometric measurements could be important for verification of the implementation and gaining insight in the process for further development. If the odometry was near perfect, the SLAM process would be trivial, only a matter of writing LIDAR data to a map at positions

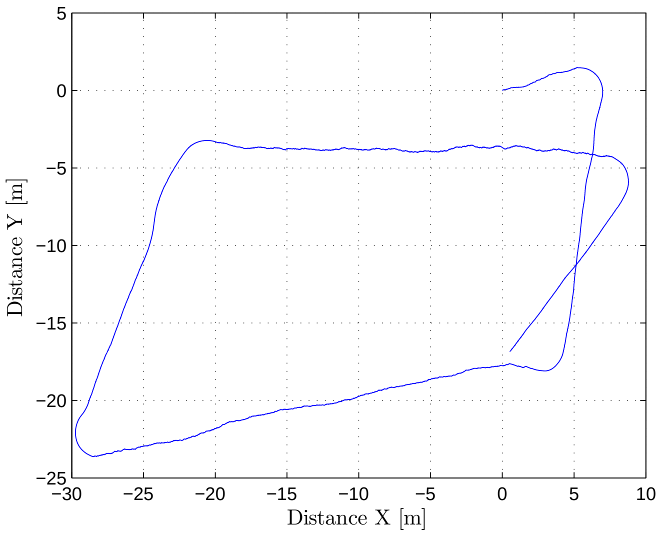


Figure 5.3: Path of the first floor of the Gamle Elektro building at NTNU obtained from odometry alone. The actual path closely resembles a rectangle, where the navigation was continued for some time after loop closure.

given from odometry.

An experiment was set up so that odometry data alone could be used for computing an approximation of the path traversed by the robot. A MATLAB script was written, reading from sensor log files and recreating the path through the same mathematical model of the robot used by the SLAM process. Note that no Kalman filtering was performed on this data.

A sensor log was obtained from a traversal of a large loop in the first floor of the Gamle Elektro building at NTNU. The loop is approximately 125 meters. In addition to this length, the robot was navigated a further distance in the same loop for comparison.

A plot of the computed path using odometry alone can be seen in Figure 5.3. The environment is the same as figured in Figure 5.12, where a floor plan can be seen.

The parameters of the model used for making this plot were set to the ones obtained from physically measuring with a tape measure. The base width of the robot was set to 0.389 meters, while the radius of the encoder wheels were set to 0.0505 meters.

5.2.1 Discussion

The experimental results should verify that the conversion from the raw encoder measurement's *pulses since last reading* is accurately converted to odometric measurements, as well as give an idea of the accuracy achieved.

The results show that the odometric readings are correctly implemented, as the distances of the straight line segments approximately coincide with physical measurements of the area, and with maps produced by the SLAM algorithm of the same area. This is the same area as the "large loop" in Section 5.6.

The angles of the turns are somewhat off. Note however that the errors of the different turns do not seem to be much correlated. The angles of turns 1, 3 and 5 are too acute, while angles 2 and 4 are too obtuse when numbering the turns in order of appearance from the robot's start point in $(0, 0)$. This suggests tuning in order to make one turn more straight could further hurt others.

Sources of error in the odometry, especially regarding turns, include errors in the mathematical model. The mathematical model used is that of an differential wheeled robot. Weight distribution and motor tuning of the robot is not perfect, so the robot might turn around an axis slightly offset from the geographical center of the drive wheels. However, if this was the source of the error, one could assume the errors would be more consistently leaning towards either acute or obtuse.

Another source of error is discretization errors. In the worst case, when driving slowly, the encoder on one side might experience zero pulses, while the opposite encoder might experience one pulse. In truth, the first encoder might have turned just too little to generate one pulse, while the other might have might have turned just enough. The resulting error in rotation can be calculated as

$$\varepsilon_\theta = \frac{2\pi \cdot r_w}{\rho b} \approx 1.019 \times 10^{-3} = 0.058^\circ, \tag{5.1}$$

where r_w is wheel radius, ρ is encoder pulses per revolution and b is the base with of the robot.

While Equation 5.1 shows the error per time step is small, making a turn a turn can consist of over 100 such time steps, of approximately 100 milliseconds. Even so, it is not easy to rationalize that the discretization errors should account for all the error.

With good performance on straights, the odometry should be able provide some useful data for SLAM algorithms, for instance in situations with plain corridor walls or open spaces. The initial assumption that odometry alone could not have been used for SLAM purposes is however validated – a map purely made from these measurements would have been very inaccurate.

5.3 With versus Without Odometry

As discussed in Subsection 2.2.2, the importance of odometric measurements is open for question. Future work could benefit from knowledge about how the odometry influence the SLAM algorithm.

Working towards simpler software and simple hardware could be of interest for the project. Assessing the importance of odometry could help ensuring future work is spent in the most beneficial areas.

Three different scenarios were considered for assessing the importance of odometry. The two first are well defined situations, where the LIDAR has a view of landmarks throughout the log. For all experiments, the same sensor logs were used with and without odometry, so the LIDAR data is identical.

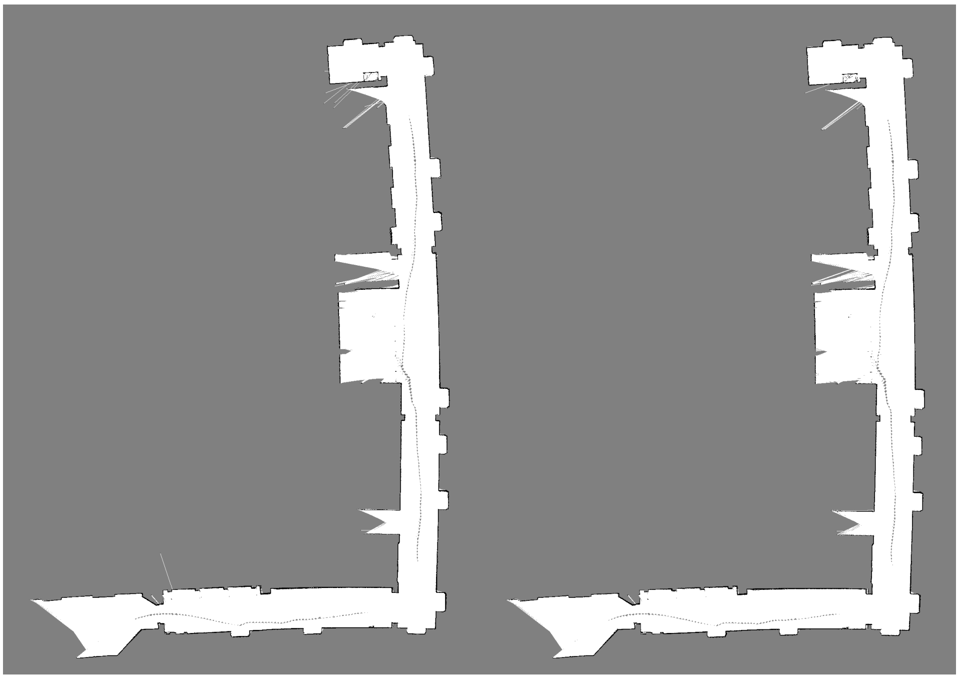
5.3.1 Without loop

The first experiment was based on a sensor log from the second floor of the *Gamle Elektro* building at NTNU. The traversed route does not contain any loops, and the environment is well defined. The route is the same as in the experiment of on-line mapping in Section 5.1.

Table 5.1 shows the resulting end positions relative to the map's origin for the first log. Figure 5.4 shows the corresponding computed maps.

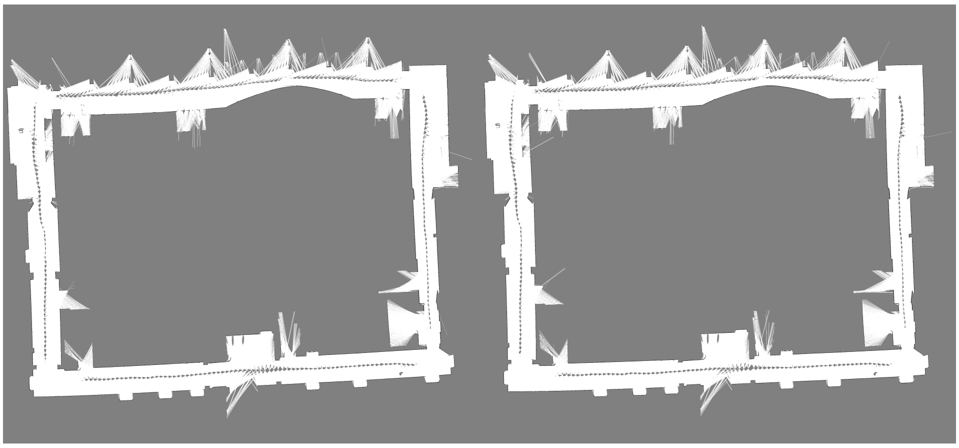
	x [m]	y [m]	θ [°]
With odometry	17.44	38.83	-179.56
Without odometry	17.64	38.74	-179.99

Table 5.1: End positions with and without odometry using the sensor log captured for Section 5.1.



(a) With odometry **(b)** Without odometry

Figure 5.4: The resulting maps from off-line SLAM with and without odometry in the second floor of the *Gamle Elektro* building. The LIDAR data for the two maps is identical.



(a) With odometry (b) Without odometry

Figure 5.5: The resulting maps from off-line SLAM with and without odometry in a loop of the first floor of the *Gamle Elektro* building. The LIDAR data for the two maps is identical.

5.3.2 Loop

The second experiment was based on a sensor log from the first floor of the *Gamle Elektro* building. The environment contains a loop, which can be used to assess the quality of the map.

Figure 5.5 shows the result of the second log, computed with and without taking odometry into account.

5.3.3 Poorly Defined Areas

The SLAM process was assumed to perform poorly in certain environments, specificity places where motion is hard to deduce from LIDAR measurements alone. Examples are open spaces and corridors with plain walls. For the former, the LIDAR's limited range could cause it not to detect distant objects. For the latter, the lack of landmarks in the direction of travel could introduce uncertainty in the same direction.

In order to assess the performance in such environments, and to provide illustration of the problem, a test was performed in the *Glassgården* area of NTNU's *Elektro* building. The area is large and landmarks are sparse.

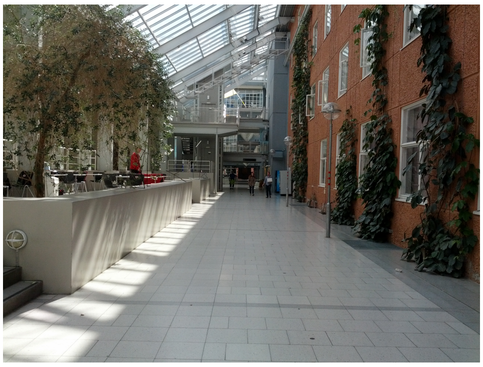


Figure 5.6: The *Glassgården* area at NTNU.

At the ground plane, where the robot was navigated, the space resembles a wide corridor, of approximately 6 meters' width.

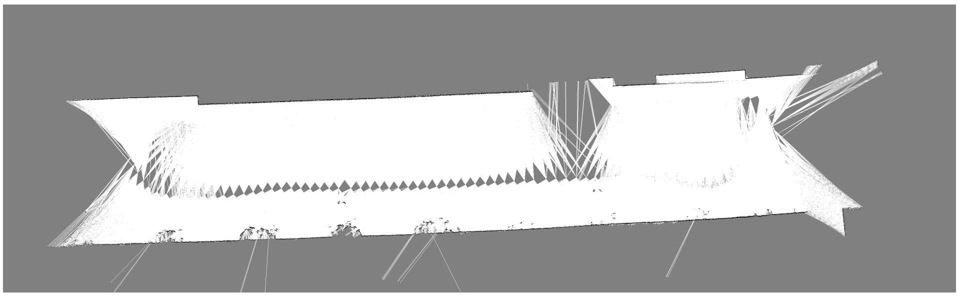
The robot was navigated in both length directions, forming a loop, in both directions near the robot's left wall. The start and end positions are in the lower left corner of 5.6.

Figure 5.7 shows the computed map both when odometry was used and when it was not used.

Figure 5.8 shows the calculated map together with the calculated path of the robot. Note that the robot started and stopped in the same area, while the map shows a distance of around 11 meters between the two points. The calculated travel distance while staying close to the upper wall was approximately 12 meters, and approximately 25 meters on the return trip. The real length of the path is approximately 25 meters.

As an experiment with the implementation, the code was modified in order to weight the odometry more while still using the same general approach. More specifically, the modified code considers scan matching estimates as measurements, while updating the map directly from the EKF. More motivation and discussion for this approach is presented in the discussion below.

Figure 5.9 shows the result with the modified code.


(a) With odometry

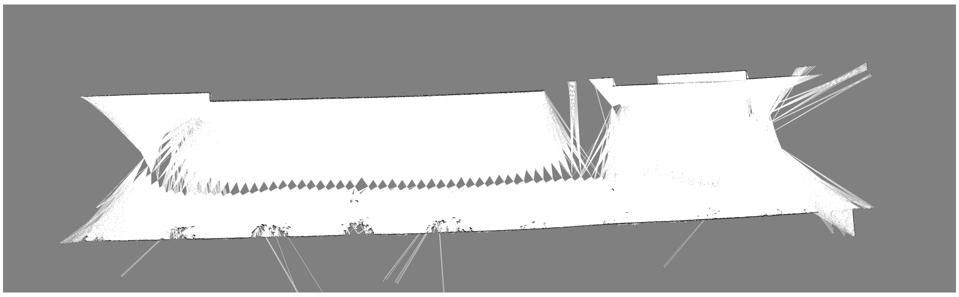

(b) Without odometry

Figure 5.7: The resulting maps shown computed with and without odometry. The LIDAR data is identical for the two maps.

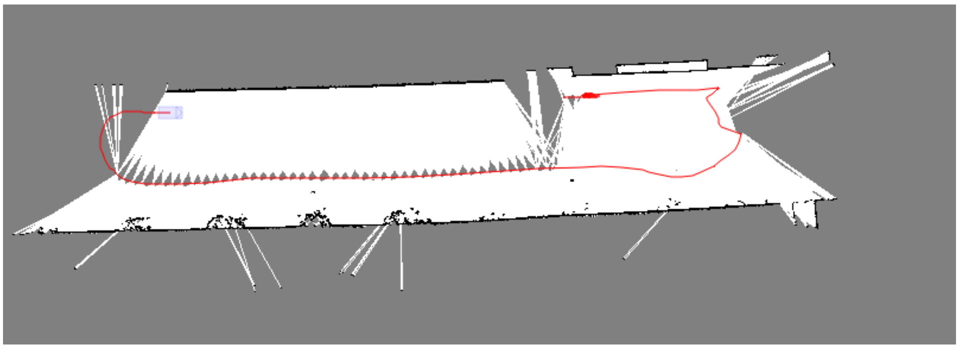


Figure 5.8: Screen shot which embeds the computed path of the robot. Note that the distances along the upper and the lower walls is equal in real life, forming a loop, while in the map, the upper distance is estimated to be much shorter. The map shown is the same as Figure 5.7b (without odometry), at a different resolution.

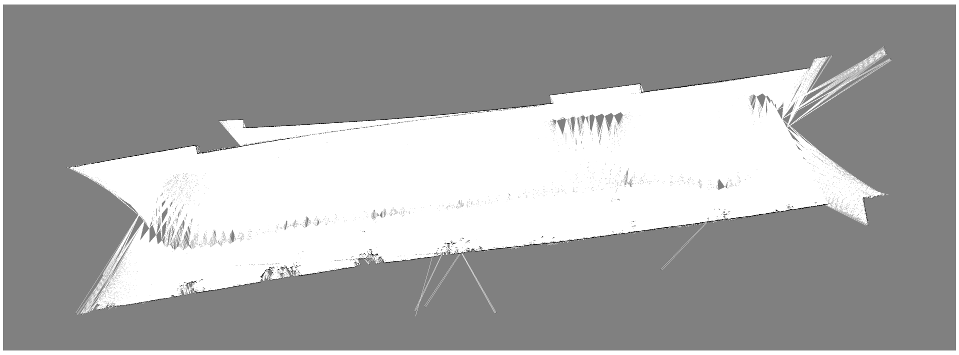


Figure 5.9: Map generated with alternative approach to mapping, which favours odometry more than before.

5.3.4 Discussion

The first two situations are well defined for the LIDAR, and the maps are successfully and accurately updated, even without odometry. In these situations, the maps with and without odometry are very similar, and only by carefully comparing can differences be found. Thus, the tests are not conclusive as to whether odometry helps the SLAM process. However, the test also shows that SLAM performance is not hurt by odometry.

The most central part of the implementation for this experiment is described in Section 4.5.3. One could argue that the integration of odometry in the SLAM process is not optimal, and is in effect giving an unreasonable large weight to pose estimates from the scan matching process. However, the test with the large loop shows that the map is of high quality and the loop is almost closed. A better integration of the odometric measurements would therefore not give large differences.

This illustrates the statements given in Section 2.2, that for well defined situations, LIDAR scan matching is more accurate than odometry and dead reckoning. To put it another way, when comparing to the experiment with *only* odometry, described in Section 5.2, it is clear that scan matching contributes much more for a correct map than odometry does.

In the last situation, where the environment is poorly defined for LIDAR scan matching, one could expect a more accurate map given the odometry. Close inspection of the resulting maps with and without odometry indeed shows that odometry does help, but only slightly. This can be seen by a double wall in the lower right corner in the map without odometry, which is more visible at the lower resolution in Figure 5.8.

In effect, while driving close to the upper wall, the environment was truncated by over 50 per cent for experiments both with and without odometry. While driving close to the lower wall, the calculated distance was correct or near correct. This side of the environment has more features, including plants and light posts, as seen in Figure 5.6.

Again, the impact of the odometry with the regular implementation is only slight, and the argument that the integration of odometry could have been done more accurately holds.

The above results motivated a study of the implementation, and an alternative approach to the mapping was conceived. As shown by the experiments, in poorly defined areas, odometry and dead reckoning can give more accurate results than scan matching. The original implementation always uses the result from scan matching for the mapping phase, and uses the EKF as a hint. The alternative places more weight on the EKF, and uses the EKF estimate for mapping, after updating it with an estimate from the scan matching.

The alternative approach was implemented and is tagged in the software Git repository as `directEKF` for documentation and later use.

Inspecting the map from the modified code, in Figure 5.9, the results for this test are considerably better than with the original implementation. While traversing along the upper wall, the wall is still truncated, but less than with the standard implementation. The mapped length of the upper wall, when following the same wall, is almost as long as on the return trip, meaning it has been much less truncated. To fully appreciate this fact, consider the starting point of the robot, shown in Figure 5.8. There is also no double walls in this part of the map.

However, on the return trip, the computed path and map diverges, taking a lower path. The SLAM algorithm with the modified code is more likely to do this. One way to look at it is that the mapping algorithm now makes more of a "compromise" between odometry and scan matching for the map updates.

When testing the alternative approach on other sensor logs, similar results were found. The alternative approach is much less reliable for well defined situations, corridors appear more bent in maps and the ability to close loops is severely hurt. For this reason, further development and testing of the alternative approach was abandoned.

5.4 Automatically Traversing a Stored Map

When the robot is operating repeatedly in the same environment, benefits can be gained from using the same map each time. This is needed for effectively planning paths to some goal location. Given a precise map to start from, a map which is known to have only small errors, the accuracy of the localization in that map is presumed to be higher than when mapping unknown environments.

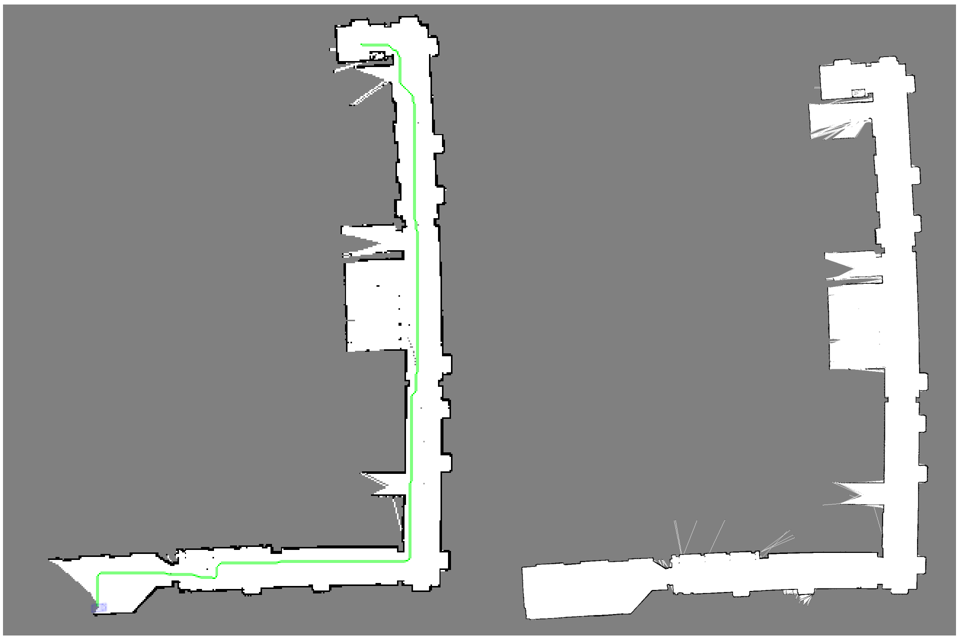
When using a stored map, the map is loaded from a file into the computer's memory as described in Chapter 4. During execution, the map is always updated together together with the SLAM process, matching current sensor scans with the map to obtain a position estimate. As a result, the map is updated with observed obstacles or the fact that obstacles previously in the map are now free areas. However, permanent features of the map should not move, but perhaps be updated to a more precise representation, such as smoothing of the walls.

The SLAM algorithm's ability to navigate in previously obtained maps is essential for several use cases. It allows the software to compute an effective route to a target location and gives a remote operator a context of the location in which the robot is estimated to be.

This experiment was designed to show the fulfillment of point 4b of the thesis' problem description.

5.4.1 Execution

The robot was placed at the approximate origin of the map obtained in Section 5.1, accurate to within centimeters. The same map was loaded and sensors were connected and started. The presented output is obtained from



(a) Path planned from map origin. **(b)** Resulting path after return.

Figure 5.10: Figure 5.10a shows the path planned from the map's origin to the most remote point of the map before path following began. Figure 5.10b shows the map after the robot successfully traversed the map to the most remote location and back, using path planning and path following both ways. The figure on the left is a screenshot and is thus depicted at a slightly lower resolution.

on-line SLAM.

A point in room G211 was selected, and path planning was initiated to this point. The path was inspected and found to be rational, as seen in Figure 5.10a. Path following was initiated, and the robot started traversing the path.

The path following was accurate, and the robot did not deviate noticeably from the path. About 4 minutes later, the robot arrived at the goal location and stopped.

Upon arriving at the goal location at G211, a new path was planned back to the origin of the map. The robot followed the path back to the origin, and produced the map seen in Figure 5.10b.

When near the origin on the way back, the robot was on one point driving at constant speed, deviating slightly from the planned path. The operator unplugged and replugged the motor controller card, and the robot subsequently corrected back to the planned path. The controller card had crashed. This did not interrupt the SLAM algorithm, and when control was regained, the path following continued without problems.

5.4.2 Discussion

On the path planned from the origin to the end point, showed in Figure 5.10a, the path seems largely reasonable. At the start, the path has a straight corner, where the it could have been diagonal. A little later, the path makes an S-formed shape with almost right angles, which can seem unmotivated. However, close inspection of the map shows the path is avoiding a mapped obstacle in the middle of the corridor. The obstacle is not static, and might have originated from a person or object at that location in the original map.

The route largely consists of straight angles, which is suitable for the path planning implementation. However, close to the end point, the path is diagonal, showing that this implementation with path smoothing is also capable of planning diagonal lines.

The path avoids walls and other obstacles, avoids unknown areas and does not cross walls. The path is considered satisfiable for the needs.

When updating the previously obtained map, the SLAM algorithm shows good performance. Note that no inconsistencies appear, indicating that localization was accurate to the original map. It is difficult to assess the absolute accuracy of the localization, but with no inconsistencies to the original map, accuracy in areas where the LIDAR measures perpendicular walls must be within the “width” of the walls, relative to the original map. The walls are in most places 3 cells wide, equating to 7.5 centimeters. This is within the accuracy of the LIDAR, having an error of $\pm 3\%$.

Some outlier readings are written to the map, seen as small white lines in the lower part of the map. These are incorrect measurements of the LIDAR, which could have been filtered out.

Note that the algorithm is not able to correct the bending of the cor-

Distance	Map [m]	Physical [m]	Error [%]
A	39.5	39.3	0.5
B	29.6	29.5	0.3

Table 5.2: Distances measured in the map obtained in sections 5.1 and 5.4, compared to physically measured distances. Error is calculated as $\epsilon = |d_{map}/d_{physical}| - 1$.

ridors. Doing so would cause inconsistencies in the map. The large-scale errors of a map will not be corrected by repeated traversals.

The traversal of the stored map was successful. The software accurately localized the robot in the map. The planned path was followed within accuracy enough not to hit walls or other objects.

5.5 Map vs. Physical Measurement

A simple measure of the accuracy of maps is to measure them against real world data. While maps can have inaccurate curvatures, incorrectly non-smooth walls or other imperfections, their size compared to the real world is an intuitive measurement of quality.

In order to assess the accuracy of the maps obtained from the SLAM algorithm and the sensor data in this manner, the map obtained in Section 5.1 and updated by the procedure in Section 5.4 was used.

This experiment does not have an explicit correspondence with any point in the problem description, but is provided as an indicator of map quality.

The finished map was loaded into the program. Two distances which were easy to identify were measured in the map using the web interface's built-in measuring tool, as described in Section 4.11.

Real world measurements were obtained by measuring the same distances using a 50 meter tape measure. Given some error in the points of physical measurement contra the map measurements, and some slack in the tape measure, the real world data is assumed to be accurate to within 0.1 meters.

Table 5.2 shows the real world measured distances compared to the distances in the map.

5.5.1 Discussion

As seen in Table 5.2, the distances measured in the map coincide very closely with physically measured distances. The error of the measurement with largest error represents an error of approximately 5 millimeters per meter driven by the robot.

Considering the error of the LIDAR measurements are on the order of $\pm 3\%$, and the fact that the LIDAR only covers a sector 5.6 meters in radius, the result can be considered satisfactory.

The results are based on a map where the SLAM algorithm successfully integrated scan matchings, with no catastrophic errors. In less fortunate situations, the errors can be larger, as discussed in Section 5.3.3.

5.6 Closing of Loops

As discussed in Section 2.5, the process of closing any loop in an environment can be a point of failure in any SLAM algorithm. If an environment contains loops, closing them is crucial to the consistency of the map. A successful loop closing can be seen as an indication of map consistency.

Several loop closing experiments were performed, in loops of several sizes.

5.6.1 Execution

For the loop traversals, the robot was started from a point in the loop. Both LIDAR and odometry sensors were connected and started, and the robot was manually controlled. The SLAM process was performed off-line at a later time.

Small loops

The first loops were done in a room of the *B block* of the *Elektro* building at NTNU. An environment was set up in a room by placing tables on their sides, such that they form a shape. The robot was navigated around these tables and out on a neighboring corridor, forming two small loops. SLAM

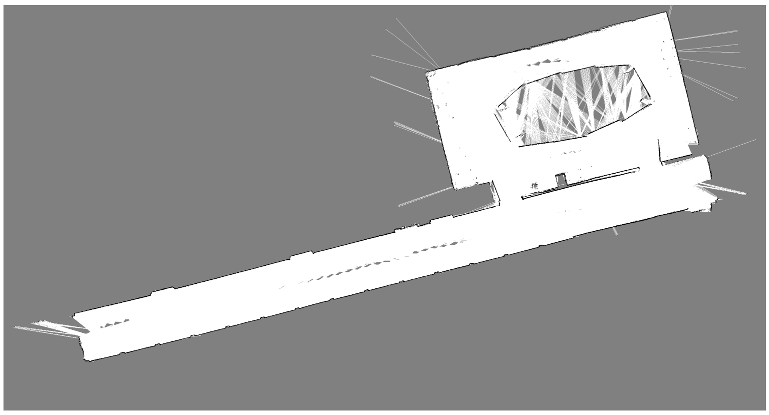


Figure 5.11: Loop in a room and out on a corridor in *Elektro B block* at NTNU. The structure in the middle of the room is made of tables set on their sides. The map contains two smaller loops.

was performed with a resolution of 1.25 cm per cell. The result is seen in Figure 5.11. The approximate length of the loop is 20 meters, while the room has measurements of approximately 10 × 6 meters.

As can be seen from the map, there were glitches between the tables in the center structure. The LIDAR detected some of the wall on the opposite side of the structure while the robot was driving.

Both of the two loops were successfully closed. After loop closing, the SLAM process continued without generating inconsistencies.

Large Loop

A larger loop was traversed in the first floor of the *Gamle Elektro* building at NTNU. The loop forms an approximate rectangle, with three sides consisting of corridors, while the last is open on one side to a large space with columns.

Some sections of the loop are poorly defined for the SLAM process, as they feature corridors with few landmarks. Therefore, three trash bins were placed at these locations, increasing the chances of a consistent map.

The loop was not successfully closed. Close inspection of the map shows a distance of approximately 11 cm at the point of loop closure. There is also an angular difference of approximately 9°. If the SLAM process was allowed

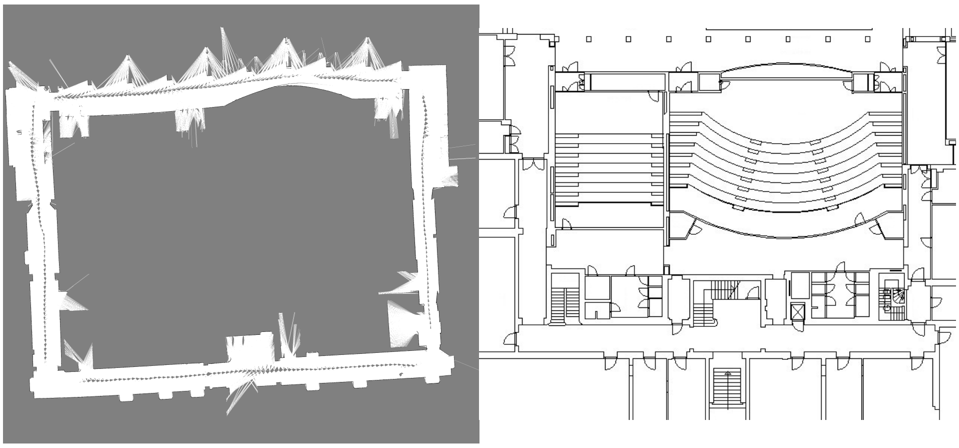


Figure 5.12: Larger loop of the *Gamle Elektro* building at NTNU figured together with a floor plan for reference. A the top of the map, the loop is open to a large space, only enclosed by columns. The robot started in the top center of the map, traversing the loop in clockwise direction. Upon return to the start point, the map is not fully consistent.

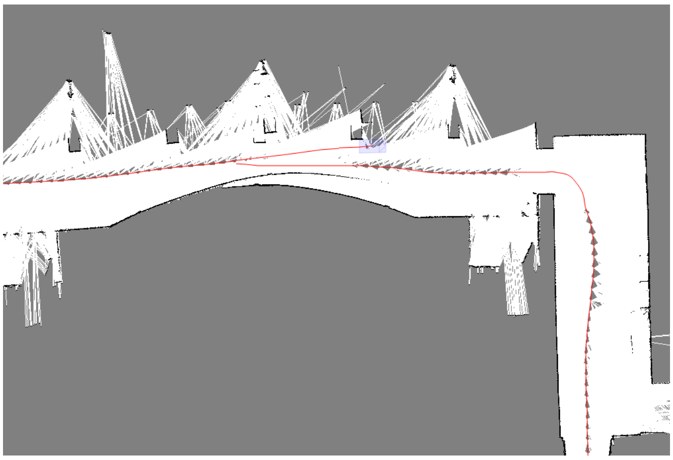


Figure 5.13: If allowed to continue after an unsuccessful loop closure of the large, the SLAM process diverged from the previously obtained parts of the map. This figure shows a different, but comparable, execution from that of Figure 5.12.

to continue, the map inconsistencies were increasing, as seen in Figure 5.13.

5.6.2 Discussion

The results of the loop experiments show two successfully closed loops, and one instance where the SLAM algorithm was unable to close the loop with enough precision to avoid map inconsistencies.

The precision needed for loop closure is hard to assess. It is dependent on the scan matching scheme, the LIDAR and the configuration used.

Generally, a higher number of map layers used in the scan matching algorithm causes gives a better likelihood of loop closure because the algorithm is less likely to get stuck in local minima. However, a too high number could cause erroneous matches elsewhere.

The LIDAR contributes by the number of backward-facing laser beams. A *lower* number of such beams means the scan-matching is more likely to “jump” over the error, matching instead with the meeting part of the loop. However, if the LIDAR was configured to using less of the backward-facing beams, map quality overall could be reduced. Also, if the scan matching algorithm has to “jump” in this way, the map quality could be poor in this area and subsequent traversals of this part of the map could face the same problem. This approach to improving the chances of successful loop closure was therefore not considered.

The final point for assessing the accuracy needed has to do with inner workings of the scan matching. The method is iterative, theoretically improving the match for each iteration. Intuitively, using more iterations could be an alternative to improving the result. The current approach is to use 3 iterations in each of the smaller-resolution map layers, and 5 iterations for the layer with highest resolution. However, increasing the number of iterations did not seem to improve results.

While the hard limits for the tolerable error in loop closing in terms of translation and rotation is difficult to assess, the experiment with the large loop at least shows that 11 centimeters and 9° was not close enough with the current algorithm and configuration.

For increased consistency and ability to close maps, two of the solutions which seem the most likely to work is multi-hypothesis mapping and increased LIDAR range.

Multi-hypothesis mapping, as performed by Rao-Blackwellized algorithms as discussed in Section 2.7 helps by maintaining a *distribution* of maps, essentially permutated by updates from a probabilistic motion model of the robot. By maintaining multiple maps, it is more likely that one of them is accurate enough to successfully close the loop.

An increased LIDAR range helps by allowing a more accurate map to be constructed to begin with. One of the applications of the original Hector SLAM algorithm [37] is a hand-held device with an IMU and a LIDAR with 60 meters range. In a corridor scenario, such range would help by having a constant view of the end of the corridor, assuming the corridor is no longer than 60 meters. Moreover, by measuring range to points on walls further down the corridor, a more precise assessment of the curvature of the corridor is facilitated, leading to fewer errors of rotation. The article using this LIDAR shows there is no need for explicit loop closing.

Nonetheless, the SLAM algorithm was able to close the two smallest loops with enough precision for it to continue creating a consistent map. This demonstrates that the software in its current state is able to handle such small loops of the environment.

The map of the large loop does not contain any significant or visible inconsistencies up until the point of loop closure. It did also only miss by 11 centimeters, which must be considered a low amount given the size of the 125 meter long loop, the 5.6 meter range of the LIDAR and its errors. Although the loop was not successfully closed, the result suggests precision of the map.

The experiments show the software's ability to handle loops in the environment, and gives an intuitive evaluation of the maps' accuracies. Small loops can be successfully closed, but the current composition of hard- and software is not able to reliably close all loops found in indoor environments. With a single-hypothesis SLAM algorithm and the small range of the LIDAR used, the author regards the results as highly satisfactory.

Map size	Map levels	Tot no. cells	Average CPU [%]	RAM [MB]
4096	4	22.2×10^6	7	2790
4096	3	22.0×10^6	6	2710
4096	2	20.9×10^6	5	2460
4096	1	16.7×10^6	2	1980
2048	4	5.57×10^6	5	715
2048	3	5.50×10^6	4	690
2048	2	5.24×10^6	3	670
2048	1	4.19×10^6	2	534

Table 5.3: The SLAM process' computational power usages at different map configurations. Map size is denoted in the number of cells in each direction, along with the total number of cells across all map levels.

5.7 Computational Requirements

While the robot's on-board computer in this case was a powerful PC, future projects based on the same system may want to cut costs or weight by using a less powerful computer. Therefore, a short documentation of the system's needs of computational power follows.

All tests in this were experiment performed on a desktop computer with an *Intel® Core™i5-2500* 3.30 GHz CPU and 8 GB of RAM, similar to the robot's on-board computer.

While the CPU usage is roughly constant by the SLAM process, the RAM usage is typically growing until Go's garbage collection mechanisms start. The RAM usage was noted after stabilization.

A summary of the noted computational resource consumptions for the implemented SLAM is presented in Table 5.3. Map levels are image pyramid like representations, each level of half the resolution of the next, as explained in Subsection 2.8.2. Figure 5.14 shows the same data, plotted as RAM usage versus the number of cells.

5.7.1 Discussion

In terms of CPU usage, the results reveal low needs of the SLAM algorithm. In the worst case, the algorithm used less than 10 % of the available computational power. This coincides well with the findings presented in the

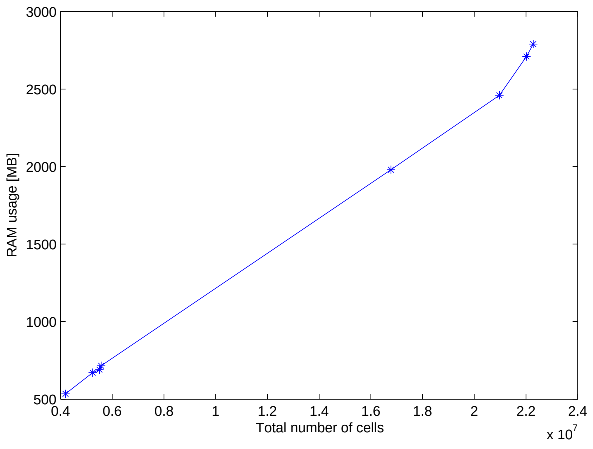


Figure 5.14: RAM usage in MB against the total number of cells across all map levels. The raw data is presented in Table 5.3.

original article, which runs the algorithm on a relatively low cost Intel Atom Z530 CPU.

The need for RAM memory is comparatively large. Most of the initial tests were executed on a configuration of 4096×4096 cells and 4 layers, the most costly alternative of Table 5.3. Note that the almost 3 GB of RAM is excluding memory consumed by operating system or other applications running simultaneously. Future systems using this algorithm at these map sizes should therefore have more than 4 GB of RAM, 8 GB is recommended in order to avoid use of *virtual memory* on the hard drive.

Note that the total usage of RAM is nearly linear with the number of cells used by the SLAM algorithm. This shows the amount of memory used by the system can be approximated to 130 MB per 10^6 cells, when calculating the SLAM algorithm's needs for a new on-board computer or proposing larger resolutions.

The current implementation of the log-odds maps for Hector SLAM, as discussed in Section 4.5, uses *float64* numbers. A case could be made that the RAM usage could be reduced by using *float32* numbers instead.

When considering the need for CPU power of potential on-board computers, please also consider the needs of the path planning algorithm, as

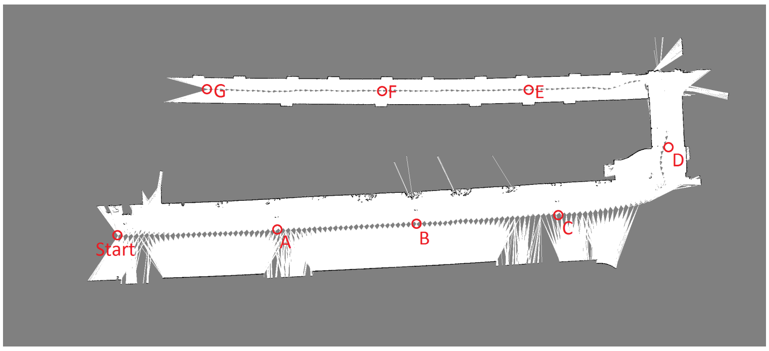


Figure 5.15: Goal locations A-G defined for the path planning experiment. The start point of the paths is also figured.

discussed in Section 5.8.

5.8 Path Planning

As discussed in Chapter 4, planning paths in grid maps is not trivial, and many path planning algorithms exist. The implemented A* algorithm is optimal to its constraints and tuning, but needs documentation of how well it performs, possible failure modes and time consumption.

An experiment was therefore set up, using a map of the *Glassgården* area at NTNU, with a corridor in a neighboring building. Seven goal locations were defined, designed to illustrate performance and weaknesses. The map and goal locations A-G can be seen in Figure 5.15. The path planning was done with 0.4 meters of "check radius" (minimum obstacle distance threshold), smoothing data weight of 0.5, smooth weight of 0.3 and a punishing factor for unknown areas of 100.

The software was able to successfully find paths for all goal locations. Even so, point E was problematic. Between point D and E, there is a narrow passage, which the path planning algorithm was unable to pass through with the configuration used, which had a minimum distance to obstacles of 40 centimeters. This configuration was found to be a comfortable distance through initial testing. The opening is 0.80 meters wide.

Instead of failing when computing a path to point E, the path planning

Goals	A	B	C	D	E	F	G
Time [s]	19.5	18.7	18.7	17.7	21.0	19.7	18.9

Table 5.4: Time spent for computing paths to goals A-G.

algorithm found a much longer route, as illustrated in Figure 5.16. The path goes through unknown environments.

For points F and G, the path planning algorithm found an alternative route through an unmapped area. Paths planned for points D, E and G are illustrated in Figure 5.16.

The time consumption of the path planning is summarized in Table 5.4. Note that the time for path planning includes the creation of binary maps and path smoothing, as discussed in Chapter 4.

5.8.1 Discussion

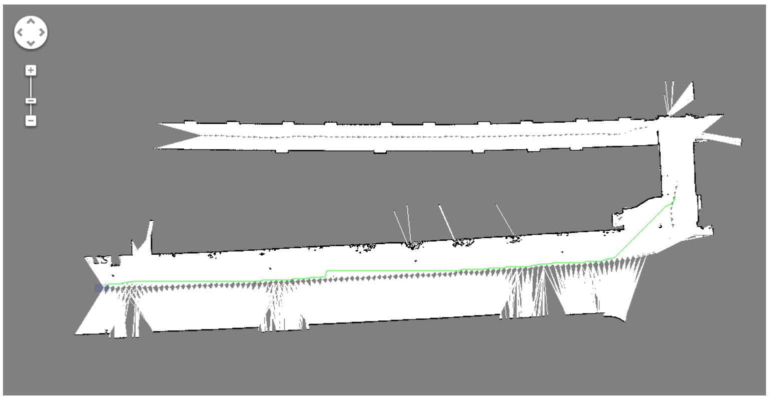
The path planning algorithm possesses some flaws. Even with path smoothing, paths are sometimes jagged, which is suboptimal. However, further smoothing of the paths might cause paths to converge in to unsafe distances to obstacles. When the robot is following such lines in practice, the movement is slightly jerky, but does not pose a big problem.

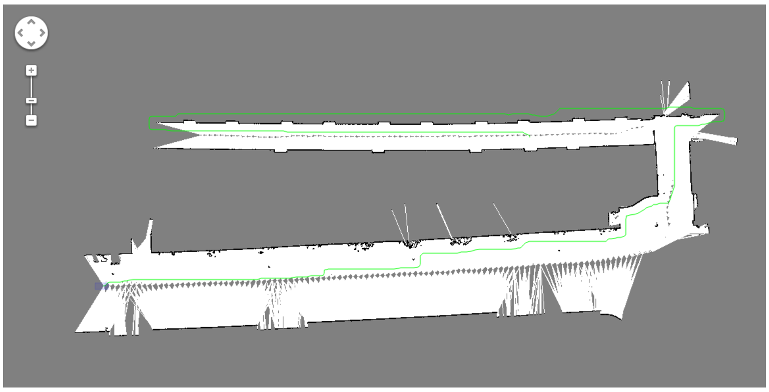
The path planned to goal E represents a bigger problem. When the path was made, the robot obviously passed through the passage between D and E without problem, but the path planning algorithm is unable to plan such a path.

It is important to note that lowering the minimum distance-to-obstacle threshold of the algorithm would have allowed it to plan a path through the passage. This was confirmed by setting the distance threshold to 0.30 meters. However, this might have resulted in the algorithm planning unsafe distances elsewhere, particularly when turning around corners, as the smoothing step will cut the corners slightly. A distance of 30 centimeters gives the robot only 10 centimeter clearance on each side.

Another example output from the path planning algorithm can be seen in Section 5.4. Here, the algorithm performs without annoyances.

Overall, the path planning algorithm seems to perform well. For clearly


(a) Path to point D.


(b) Path to point E.

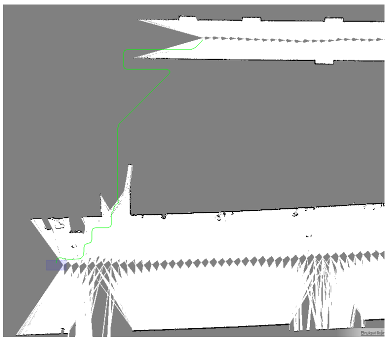

(c) Path to point G.

Figure 5.16: Paths planned to goal locations D, E and G.

defined areas without very narrow passages, the algorithm has no problems. Passage problems can occur at door openings with the current parameters.

When it comes to computational speed, the algorithm shows an almost constant time consumption for paths to all the goals, which is somewhat contra-intuitive given the differences in distance. Some of this effect can be accounted for by the creation of binary maps, which takes constant time for a given map size. The results suggest that the main body of work is done in this step.

The path planning executes in around 20 seconds. For known maps, this is not a big problem, as path planning happens only once. When planning path through unknown areas, or when the goal location is in an unknown area, paths might be planned which collide with unknown obstacles. When the software through the collision avoidance module described in Section 4.4 detects such an obstacle, a the path will be re-planned based on the updated map. If such obstacles are frequent, the execution time of the path planning significantly slows down the automatic traversal. Note that this is not a problem if the robot is controlled manually, as paths are not planned in manual mode.

5.9 Automatic Exploration

Automatic exploration of unknown environments is supported by the software. An experiment was conducted in order to demonstrate the software's abilities in this respect.

Automatic exploration is done by selecting a target in an unknown area. The software plans a path to this location using the available information of the map. Should the robot meet an obstacle in this path, a new path is planned. The process continues until the robot arrives at the goal location or no path can be found to this location.

5.9.1 Execution

The robot was started in a location of the second floor of the *Gamle Elektro* building at NTNU. A goal location was selected at a location known to be outside reach, in order to allow the robot and software to explore the environment in an attempt to reach the goal.

Screen shots from the interface during the automatic exploration can be found in Figure 5.17. Note that Figure 5.17a is taken some time after start-up. The sub-figures a-f are shown in chronological order.

5.9.2 Discussion

The experiment demonstrates how path planning and traversing with re-planning can be used for automatic exploration.

The robot starts off with a map of only what is visible to the LIDAR in that position. In attempts to reach the, in this case unreachable, goal destination, paths are re-planned as they prove impossible to traverse because of obstacles. The path traversing is stopped because the collision avoidance module detects an obstacle in a sector in front of the robot, as explained in Section 4.4.

It should be noted that the scheme could be done more efficiently. When the robot stops, information that the path is blocked is often already available. If the path planning algorithm was faster, path planning could have been done repeatedly, while the robot was driving. Because the path planning takes many seconds, the robot would have typically already moved significantly from the start position of the planned path, if done while driving.

Nonetheless, the results show that the robot is capable of autonomously exploring and mapping the environment, fulfilling point 4a of the problem description.

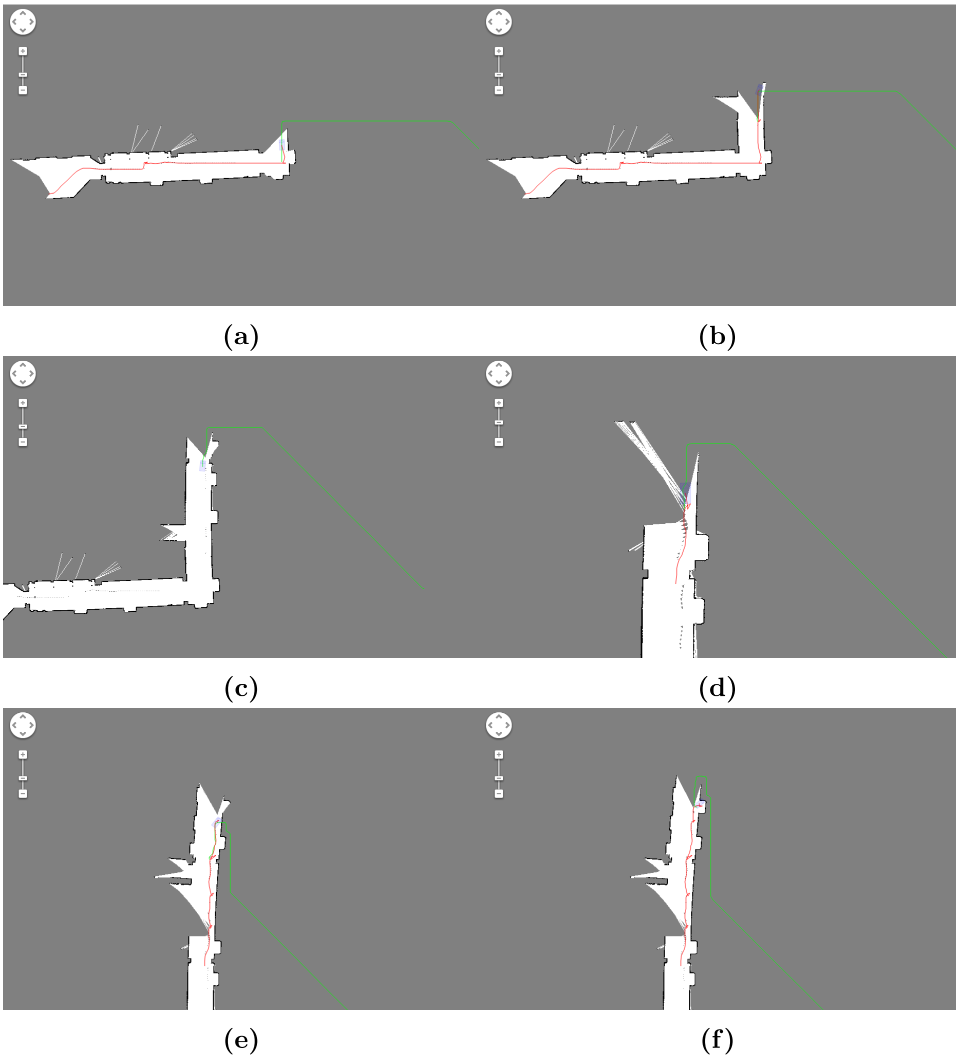

(a) (b) (c) (d) (e) (f)

Figure 5.17: Automatic exploration, shown as a series of screen shots from the web interface. Planned paths are shown in green, trace of robot path is shown in red. A path to the same goal location is re-planned as paths show impossible to traverse.

Chapter 6

Discussion

This chapter is devoted to a discussion of the results presented in Chapter 5, as well as a discussion of elements of the implementation.

6.1 Interface and Manual Control

The web interface was used for run-time interaction with the software during development, initial testing and experiments. It was used from desktop and laptop computers as well as from smartphones.

When controlling the robot manually while walking behind the robot, it was often found convenient to control the robot from a smartphone rather than a laptop computer. The smartphone's touch screen made for a particularly efficient way of controlling the interface's virtual joystick. This illustrates the advantages of providing a multi-platform interface.

The interface was stable and easy to use. Being able to control the robot from several platforms was an advantage. For example, at one occasion, the laptop used for remote control was out of battery. Control could continue from a smartphone, without problems. The only functions not available from touch units is those which require right-clicking on the SLAM interface's map. This is an implementation issue.

Some short comings have to be discussed. Some of the interface requirements, most notably the map view of the SLAM interface, requires an Internet connection. The map view is based on an on-line API, which the robot software does not serve itself. Thus, if the robot runs on a network not connected to the Internet, the map view will not work.

With the SLAM interface's automatic updates of the displayed map, an operator is able to control the robot remotely.

Implementing the human interface as a web service was regarded as a success.

6.2 SLAM Performance, Reliability and Improvements

Chapter 5 presents a number of experiments providing grounds for a discussion of the implemented SLAM process' accuracy and reliability.

Many of the test cases show good results, and some results show failed maps. Performance was particularly poor in areas where the LIDAR and scan matching can't be used to track position, as seen in Section 5.3. Summing up the SLAM performance experiments seem to result in a pattern, where these areas are the main points of failure. The same suspicion was made from initial testing during development.

For areas where LIDAR scans have enough landmarks for the scan matching to be successful, results seem to be very good. Measurements with tape measure show that the accuracy of the maps are highly satisfactory, almost within the accuracy of the tape measure when considering the long distances it was used for.

For the more accurate maps, the SLAM implementation's nature means that localization must be accurate to within centimeters when traversing previously obtained maps. Otherwise, the SLAM process would write LIDAR measurements from erroneous pose estimates, causing map inconsistencies. Although experiments were not done to explicitly assess localization accuracy, the lack of map inconsistencies suggest the accuracy comparable to the error of the LIDAR, which the maps also suffer from.

Loop closing experiments reveal that while maps of environments well defined for the scan matching do contain errors, the errors are small even when built up over a distance of up to 125 meters. However, they also show that the software-hardware combination in its current state cannot reliably close such loops.

Problematic environments for the LIDAR-scan matching approach would assumingly be fewer if the LIDAR had a longer range, although no such LIDAR was available for the work of this thesis, so no definitive conclusions can be made. The discussion of Section 5.6 presents arguments for this case.

Using a multi-hypothesis SLAM algorithm could help close loops, but would not in itself help for the poorly defined areas which were presented in Section 5.3. None of the particles in Rao-Blackwellized particle filters would move correctly further if the same underlying scan matching was done in order to propagate them, as done in [27].

The computational requirements show that maps are already taking up much of the memory which even modern computers can maintain effectively. If no compromises should be made in terms of map quality, this suggests multi-hypothesis mapping would have to be done with a map data structure for example like DP-Mapping, presented in [17], complicating the software. Note also the complications of multi-hypothesis tracking presented in Section 4.5.

Some of the errors made in such poorly defined areas might be compensated for through better use of the odometric data. The experiments comparing maps generated with and without such data show only small differences. An experiment with an alternative implementation showed better results could be obtained in situations poorly defined for scan matching, at the cost of overall accuracy and reliability.

However, as discussed in Chapter 4, other alternatives for Kalman filter updates from scan matching exist. One of them is used in the article describing Hector SLAM, but was not successfully implemented for this thesis.

Another alternative for odometric integration could be to use particle filters, for which a short introduction was given in Chapter 2. Particles of robot pose could be propagated with a probabilistic model, as described in [59], then weighted and resampled based on how well they fit with the map as obtained so far. The error covariance presented by the already implemented scan matching might be enough to weight the particles. The distribution of particles could be used to give a good estimate of position before scan matching.

Note that this is very different from using Rao-Blackwellized particle filters for SLAM. The author could not find any articles describing this use of particle filters. The implementation of such a scheme was not done because of time constraints. An implementation is assumed to be more reliant on odometry, to a point where the algorithm might need to explicitly consider the case where odometry is not available.

Global localization refers to the ability to start anywhere in a previously obtained map, and derive the current location. Such a step could be used when starting from a stored map, before the regular SLAM process takes over in order to maintain the map. Global localization was not implemented for this thesis, but could prove very useful for applications of the robot.

A common approach to global estimation is Monte Carlo localization, already mentioned in Section 2.4 and with regards to the NTNU-Eurobot robot in Chapter 4. Maps are assumed to be static, and a particle filter is used to estimate a probability distribution over poses in the map, further explained in [59] and [8].

Global localization would allow the robot to be started from any position, not just the map's point of origin, as with the current implementation.

6.3 Path Planning and Guidance

While SLAM is the most prominent field of study for this thesis, path planning and guidance is not to be forgotten. Two path planning algorithms were implemented, before a combination of A* with path smoothing was implemented and used for testing.

The path planning algorithm, with construction of binary maps, works satisfactory in most cases. A contra-example is found in Section 5.8, where a narrow passage proves a hindrance for path planning. If this proves to be a problem for the robot's functioning, the problem should be investigated further. Apart from this case, the path planning algorithm gives satisfactory results, both when planning in known and unknown areas of the map, shown by experiments in sections 5.8 and 5.4.

Apart from the quality of the paths, speed is another measured quality of path planning. The current path planning algorithm spends a relatively long time of up to over 20 seconds before returning a result. As discussed in sections 5.8 and 5.9, shorter time spans could facilitate more efficient automatic navigation.

Time posed a limit on the number of path planning algorithms which could be implemented. A* and Hybrid A* was implemented, as explained in Chapter 4, but A* was more versatile and honored the robot's ability to "turn on a dime". However, Hybrid A* is cited to produce results after 100 milliseconds [42], suggesting that further investigation on optimization and heuristics of the A* algorithm could lower the execution time.

No significant errors of the implemented guidance scheme were uncovered by experiments and initial testing. Figure 5.17 shows examples where the differences of the planned path and the robot position trace are hard to see, suggesting the scheme was accurate. The implemented line of sight-scheme is seen as both intuitive and successful.

6.4 Autonomous Abilities of the Robot

Experiments show the software-hardware combination is capable of generating maps both from manual control and by automatically planning paths in unknown environments. The robot can accurately follow these paths while simultaneously avoiding collisions. During testing and experiments, no collisions occurred except for one occasion when the motor controller card crashed. However, the motor card fault was not a concern for this thesis.

The robot does posses flaws in terms of its ability to avoid certain obstacles. These include downward stair cases and other "cliffs", low obstacles such as curbs or doorsteps, tables and other obstacles which the LIDAR cannot sense. As long as the LIDAR is the robot's only mean of detecting obstacles, these limitations will remain. Note that dynamic environments, such as when people are present during mapping, was not considered for this thesis.

Experiments suggest that the robot can operate autonomously in environments which do not contain such obstacles as described above, which has no long stretches of plain walls or open spaces (see experiment for “poorly defined areas” in Section 5.3) and which does not contain larger loops, which also can pose inconsistencies. The current hardware-software combination poses limitations, but many environments free of such obstacles or under supervision are still usable.

More reliable autonomous abilities can be achieved by manual control for initial mapping. The operator can control that the robot does not fall down stairs or crashes into unobserved obstacles. Maps can be inspected to make sure obstacles are mapped and that they are consistent. Whenever such a map is created, the robot has in experiments (“automatically traversing a stored map”) shown abilities to reliably localize, plan paths, perform guidance and update the map. For operation with the current implementation, this would be the preferred strategy.

6.5 Fulfillment of Problem Description

6.5.1 Point 1

Point 1 of the problem description states that sensor decisions should be suitable for generating maps and for autonomous operation. Experiments with on-line mapping, maps versus physical measurement and closing of loops show the sensors are successful in facilitating map generation. The LIDAR and encoder wheels combination of sensors for SLAM is common in literature [45, 17, 49] and should be usable with many SLAM approaches. The combination has also proven successful in avoiding obstacles, as shown with the experiment of automatic exploration.

Short-comings of the LIDAR, primarily in terms of range, is seen as an issue of economy and the limitations economy represented when the unit was bought.

The sensor combination facilitates autonomous operation to a large degree, as shown in particular by the experiment of autonomous exploration. The problem description is limited to not include avoidance of all kinds of collisions or driving off edges. The LIDAR, which is used to avoid obstacles, only measures in a horizontal plane, usable for detecting many kinds of obstacles, but not all. This was seen as sufficient for fulfilling the problem description on this point.

6.5.2 Point 2

A literature study of SLAM algorithms is presented in Chapter 2, which resulted in the implementation of two algorithms, of which one was supported throughout the work of the thesis. The algorithm works both on-line and off-line, as demonstrated by experiments, and handles maps of over 10 000 m^2. The point is therefore regarded as fulfilled.

6.5.3 Point 3

Point 3a describes data collection, which is implemented by the sensor module's `logging` module. Many of the experiments were dependent on this module functioning, for example the comparison of obtained logs with and without odometry.

Point 3b describes manual control of the robot, which is implemented via the web interface and the motor module.

Point 3c describes on-line SLAM with manual control, demonstrated in the experiment of on-line mapping in Section 5.1.

Point 3 is therefore regarded as fulfilled.

6.5.4 Point 4

The on-line SLAM with automatic exploration of point 4a was demonstrated in the experiment of automatic exploration in Section 5.9, while point 4b's abstracted navigation is demonstrated in the experiment of traversing a stored map in Section 5.4.

6.5.5 Point 5

Point 5 describes the interface, which is shown in Section 4.11.

Chapter 7

Future Work

This project can be developed further in a variety of different ways. The project provides a large amount of flexibility and opportunities for future project work and master theses. As a result, this chapter is relatively comprehensive.

This chapter is dedicated to presenting some of the ideas the author has for possible future work.

7.1 A Note On Extensions

All software described in this thesis is maintained in a *Git* repository hosted by the free service *Bitbucket* by *Atlassian, Inc.*. There are a number of different ways to work with the code.

If the project goes on without central management of the code, work can still continue because the repository is open.

For changes which do not break features, the preferred way to contribute is to *branch* the repository. The contributor can work on her own branch until she reaches some "stable" version, a version which has no or few known bugs. This version is then merged into the master branch.

For contributors wanting to make *breaking changes*, i.e. changes that in any significant way abandons features previously in the software, a more appropriate way of contributing is by *forking* the repository. This creates a new repository based on the same code, and the contributor is free to do as she pleases.

The repository can be found at `bitbucket.org/mikaelbe/tigerslam` .

7.2 SLAM and Navigation Related

7.2.1 Global Localization

Global localization is mentioned in Section 6.2. Given that the robot is started in some previously mapped area, the appropriate map could be selected by the operator, and a global localization scheme could try to locate the robot's position in this map. The SLAM algorithm could then be started, and achieve consistency.

The current implementation has no such feature, and always assumes the robot is started in the map's origin.

For simplification, it is also possible to give the operator the job of declaring some small area in which the robot is known to be located. This reduces the amount of computation needed to localize the robot.

The mentioned section outlines the possibility of using Monte Carlo localization to achieve this feature.

7.2.2 More Advanced Observation Model

The current implementation draws LIDAR beams to a map using a simple *Bresenham* line drawing algorithm. It is possible that a more advanced observation model could yield more accurate results.

For example, [18] presents a model based on the distance the laser beam has traveled through the space which each cell represents. One of the arguments presented for this view is that a beam diagonal to the map grid is potentially updating more cells than a beam in line with a row or column. Considering that the internal model of the cells is not binary, but a floating point number representing the cells' log-odds of being occupied, it makes sense to use a more consistent model.

The article does not present empirical results on how much of a difference the observation model yields over Bresenham line drawing, and also uses a different scan matching algorithm. Such an observation model could help in situations of obstacles with small footprints, because diagonal lines intercepting cells of the border of the obstacle would be treated more consistently.

7.2.3 Explicit Map Exploration Mode

The current implementation of automatic exploration requires the operator to select a goal location. If this is a mode to be developed further, a scheme could be developed to make exploration completely autonomous, with “the click of a button”.

The scheme could strategically place a goal location. If after some time, no path can be found to the location, or the robot reaches it, a new point could be selected. The process would terminate if no goal location is both unknown and has paths leading to it.

A way to find such strategic goal locations could be to look for neighboring cells of unknown and free state. Observed walls would cause the transition to be *unknown-occupied-free.*

7.2.4 Dealing With Non-Flat Surfaces

The software currently considers only planar environments, but many environments have non-flat surfaces. Even the areas used for the experiments presented in Chapter 5 were not completely planar, having small inclines through door openings.

When the planar assumption breaks, errors in the SLAM process can occur. With the LIDAR’s plane of measurement in a non-horizontal configuration, beams can hit the ground or exaggerate distances to walls ahead. Note that errors due to these effects were not considered in this report, but may have had a small impact.

The original Hector SLAM article [37] describes applications of the same SLAM approach where no assumption is made that the LIDAR is held completely horizontal. It uses an IMU to measure the angle at which the LIDAR is.

LIDAR scans can be rejected if the angle from horizontal is too large, or transformed into the horizontal projection.

7.2.5 Controlled Backing As Collision Avoidance Maneuver

When the robot is in guidance mode and follows a precomputed path, it can detect obstacles in its way and stop. The current implementation proceeds to "blindly" drive about half a meter in reverse, without considering that the backward direction can also be obstructed. This can be done more controlled. For more information on the backing, see Chapter 4.

Solutions include considering what the robot already knows, in other words, plan a escape path with regards to all data in the map. Another solution is to include more sensors. An example would be backward facing infrared or ultrasonic sensors, ensuring that the area is clear before backing.

If the backing procedure is planned as a regular path, this point requires the ability to traverse paths in reverse.

7.3 Architecture and Code Related

7.3.1 Start Anywhere and Scalable Maps

The current implementation features maps of constant size. Start position in the map has to be defined before starting the SLAM process.

A better way to do this would be to have maps scale when needed. Maps could start off small, then expand when the robot reaches an edge. This could relieve the on-board computer of having to maintain more memory than necessary.

7.3.2 Re-planning while Navigating

As described in Chapter 4 and later discussed, the current implementation plans a path once, then navigates the robot along the path until an obstruction is noticed very near the robot.

A better implementation would consider all the available information at all times. When the SLAM process makes a discovery that the planned path is obstructed, a new path can be planned right away – there's no need to wait until the robot has nearly crashed.

One of the reasons this functionality was not implemented during the development related to this thesis, is that the path planning algorithm is slow. It can use several seconds to plan a new path, and at that point, the map might have changed substantially.

A *Hybrid A** algorithm such as described in Chapter 4 and in [42] is an alternative which potentially could reduce planning times enough: the latter cites planning times of under 100 milliseconds. Some of the speed gain is achieved through the us of a precomputed heuristic.

7.3.3 Separation of GUI and API

The current implementation includes a web interface for controlling all of the robot's function, as described in Chapter 4. The web interface is a part of the robot software, and includes both serving of HTML and static files, and an application programming interface – an API. The API is used for communication with the robot without reloading a page in the user's web browser.

A suggestion for a better solution is to re-think this strategy, and have the software only offer an API.

Key to such an approach is that the API is exhaustive of the functions which the software should offer. In other words, the API would have to be sufficient in order to control every controllable aspect of the software.

The main advantage of this approach is the ease of which alternative interfaces could be constructed. Interfaces for the robot could be specialized, so that they only control some relevant aspect of the robot's functions. An example would be a separate application for the schedule system presented in Subsection 7.5.1. Another would be separate applications for having the robot navigating to locations decided by some unrelated process, for example an alarm.

Notice that the real improvement is the separation of code for different purposes. Having the robot react to an alarm in the facilities is not necessarily a core functionality of the robot software. Instead, the robot could facilitate such functions by allowing third party software to command the robot to navigate to a specific location.

It would be beneficial if the development of such an API included work towards formalizing the API architecture to some standard.

7.3.4 For Other Robots

While efforts have been made to make modularized and reusable code, further work could be done in order to make the software seamlessly work with other robots.

This includes having a more intuitive way of switching between drivers for different odometric and LIDAR sensors. Which driver to use could be chosen automatically or as a configuration parameter.

Code regarding path planning and guidance might also have to be updated if the software should be used with different drive schemes, such as Ackermann steering.

7.4 Hardware Related

7.4.1 Better obstacle detection

As mentioned in Chapter 3, the current sensor combination and configuration is unable to detect some crucial obstacles. If the robot should operate completely autonomously, being able to safely navigate free of all obstacles is paramount.

Obstacles which the robot could fail to detect with the current implementation includes the following:

- Downward going staircases and other “cliffs”, where the floor ends without a rail or wall. The LIDAR is unable to register changes in the floor itself, and could potentially fall down.
- Low walls, such as pavement edges and other low obstacles. If a wall or obstacle is under the LIDAR’s plane of sight, it is not detected and can potentially be crashed into.
- Tables, shelves and other objects with a footprint in the LIDAR’s plane of view small enough for the path planning algorithm to think the area is free to be traversed. Although the floor is free, the robot’s upper parts could crash into such an obstacle.

If the robot is to be made truly autonomous in environments not strictly confirmed to be free of such obstacles, all this scenarios have to be addressed.

7.4.2 Reverse Faced Range Sensor

Just like a LIDAR with longer range would help making maps more accurate, a rear facing LIDAR could also help.

To utilize more than one LIDAR unit at a time, some research and considerations would have to be done. However, as long as the transformation matrices relating each of the two LIDARs' positions to the robot are known to some precision, matching different LIDAR scans in different directions is possible.

This provides an alternative use for the currently forward facing LIDAR if a better main LIDAR is bought for the project in the future.

7.5 New Features

7.5.1 Schedule System

A "schedule" system was mentioned in Chapter 1. The proposed idea is to improve the robot's usability for remote operators by letting the operators order the robot to report at a specific location at a specific point in time.

An operator wishing to perform some work at a specific location could then use some interface to book the robot at that place, at a specific time.

Such a system would help different operators share the robot in an efficient manner. The operators would not need to worry about moving the robot to the location they require, thus saving time.

7.5.2 Automatic Recharge

Having the robot support automatic recharging at a recharging station requires work with both software and hardware. On the hardware side, the robot would have to be prepared and a charging station would have to be constructed.

For the software side, some means of monitoring the batteries' states are needed. While one possibility is to monitor the computer and motor's drive time since last update, a presumably more reliable method is to measure the batteries' voltage.

Such a system has been implemented and described by [30]. Although that article focuses on smaller robots, useful insights are provided.

7.5.3 Images and Object Recognition

The robot hardware is already equipped with cameras, which could be used for many applications.

One of them would be to capture images of the environment and link them to the position from which they were taken. The capturing could be triggered by the operator, or at regular time or distance intervals. It could be possible to implement a scheme resembling *Google Street View*, where detailed photographs are linked together with location data. Several images together with the SLAM maps and computer vision technology be used to recreate approximate 3D visualizations of the environment.

The cameras could also be used to recognize points of interest, such as doors or windows. These could be displayed in the web interface superimposed on the map, providing more details for the operator.

Bibliography

[1] Neato Robotics how it works. `http://www.neatorobotics.com/how-it-works/`. Accessed: 11/03/2013.

[2] The go programming language documentation. `golang.org/doc`, 2012. Accessed: 05/06/2013.

[3] Fabrizio Abrate, Basilio Bona, and Marina Indri. Experimental ekf-based slam for mini-rovers with ir sensors only. In *Preceedings of 3rd European Conference on Mobile Robots, European Conference on Mobile Robots*, 2007.

[4] SICK AG. Sick lms500 datasheet. `http://mysick.com`. Accessed: 11/03/2013.

[5] Kai O Arras. The cas robot navigation toolbox. *Center for Autonomous Systems*, 2004.

[6] Petter Aspunvik. Robotisert vedlikehold, 2013.

[7] David Ball, Scott Heath, Janet Wiles, Gordon Wyeth, Peter Corke, and Michael Milford. Openratslam: an open source brain-based slam system. *Autonomous Robots*, pages 1–28, 2013.

[8] Mikael Berg. Lidar localization for eurobot-ntnu, 2012.

[9] Peter Biber and Wolfgang Straßer. The normal distributions transform: A new approach to laser scan matching. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2743–2748. IEEE, 2003.

[10] W. Burgard, D. Fox, D. Hennig, and T. Schmitt. Estimating the absolute position of a mobile robot using position probability grids. *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1996.

[11] H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations.* Intelligent Robotics and Autonomous Agents. MIT Press, 2005.

[12] Javier Civera, Oscar G Grasa, Andrew J Davison, and JMM Montiel. 1-point ransac for extended kalman filtering: Application to real-time structure from motion and visual odometry. *Journal of Field Robotics*, 27(5):609–631, 2010.

[13] ROS Community. Ros introduction. `http://ros.org/wiki/ROS/Introduction`, mar 2012. Accessed: 05/06/2013.

[14] SWIG Community. Simplified wrapper and interface generator. `http://www.swig.org/`, 2013. Accessed: 11/05/2013.

[15] Albert Diosi and Lindsay Kleeman. Fast laser scan matching using polar coordinates. *The International Journal of Robotics Research*, 26(10):1125–1153, 2007.

[16] Arnaud Doucet, Nando De Freitas, Kevin Murphy, and Stuart Russell. Rao-blackwellised particle filtering for dynamic bayesian networks. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 176–183. Morgan Kaufmann Publishers Inc., 2000.

[17] Austin Eliazar and Ronald Parr. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1135–1142. LAWRENCE ERLBAUM ASSOCIATES LTD, 2003.

[18] Austin I Eliazar and Ronald Parr. Dp-slam 2.0. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 2, pages 1314–1320. IEEE, 2004.

[19] Pantelis Elinas, Robert Sim, and James J Little. /spl sigma/slam: Stereo vision slam using the rao-blackwellised particle filter and a novel mixture proposal distribution. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 1564–1570. IEEE, 2006.

[20] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. Real-time 3d visual slam with a hand-held rgb-d camera. In *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, volume 2011, 2011.

[21] Dave Ferguson and Anthony Stentz. The field d* algorithm for improved path planning and replanning in uniform and non-uniform cost environments. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-05-19*, 2005.

[22] Juan-Antonio Fernández-Madrigal and José Luis Blanco Claraco. *Simultaneous Localization and Mapping for Mobile Robots: Introduction and Methods.* Information Science Reference, 2013.

[23] Thor I Fossen. *Handbook of marine craft hydrodynamics and motion control.* Wiley, 2011.

[24] Python Software Foundation. Python v3.3.1 documentation. `http://docs.python.org/3/index.html`, 2013. Accessed: 05/06/2013.

[25] Inc. Google. Google maps javascript api v3 reference. `https://developers.google.com/maps/documentation/javascript/reference`, 2013. Accessed: 05/08/2013.

[26] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *Robotics, IEEE Transactions on*, 23(1):34–46, 2007.

[27] Giorgio Grisetti, Gian Diego Tipaldi, Cyrill Stachniss, Wolfram Burgard, and Daniele Nardi. Fast and accurate slam with rao–blackwellized particle filters. *Robotics and Autonomous Systems*, 55(1):30–38, 2007.

[28] Dirk Hahnel, Wolfram Burgard, Dieter Fox, and Sebastian Thrun. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 206–211. IEEE, 2003.

[29] Sigurd Hannaas. Samarbeidende legoroboter, 2011.

[30] Andreas Haugedal. Forbedring av de autonome egenskapene til en legorobot, 2008.

[31] Sveinung Helgeland. Autonomous legorobot, 2005.

[32] Ltd. Hokuyo Automatic Co. Scanning laser range finder urg-04lx-ug01 (simple-urg) specifications. `http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx-ug01/`, dec 2012.

[33] Andrew Howard and Nicholas Roy. The robotics data set repository (radish), 2003.

[34] Simon J Julier and Jeffrey K Uhlmann. Using covariance intersection for slam. *Robotics and Autonomous Systems*, 55(1):3–20, 2007.

[35] Stefan Kohlbrecher and Johannes Meyer. Hector slam. `http://www.ros.org/wiki/hector_slam`, dec 2011. Accessed: 05/05/2013.

[36] Stefan Kohlbrecher and Johannes Meyer. Hector slam source code. `http://code.google.com/p/tu-darmstadt-ros-pkg/`, nov 2012. Accessed: 05/05/2013.

[37] Stefan Kohlbrecher, Oskar von Stryk, Johannes Meyer, and Uwe Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on*, pages 155–160. IEEE, 2011.

[38] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence*, 1981.

[39] Trond Magnussen. Fjernstyring av legorobot, 2007.

[40] Trond Magnussen. Fjernstyring av legorobot, 2007. Pre-project.

[41] S. Mathisen, Skaran A., Nordal I., Bae M., and Vikhammer M. Prosjektrapport eurobot 2013, 2013.

[42] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[43] Michael Montemerlo and Sebastian Thrun. Simultaneous localization and mapping with unknown data association using fastslam. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 2, pages 1985–1991. IEEE, 2003.

[44] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1151–1156. LAWRENCE ERLBAUM ASSOCIATES LTD, 2003.

[45] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the National conference on Artificial Intelligence*, pages 593–598. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002.

[46] Sindre Røkenes Myren. Rt capabilities of google go, 2011.

[47] Erik Næss. Fjernstyring av legorobot, 2008.

[48] Max Pfingsthorn, Bayu Slamet, and Arnoud Visser. A scalable hybrid multi-robot slam method for highly detailed maps. In *RoboCup 2007: Robot Soccer World Cup XI*, pages 457–464. Springer, 2008.

[49] Søren Riisgaard and Morten Rufus Blas. Slam for dummies. *A Tutorial Approach to Simultaneous Localization and Mapping*, 22:1–127, 2003.

[50] A. Sperre S. Myren and A. Halvorsen. Eurobot ntnu 2012. Master's thesis, Norwegian University of Science and Technology, 2012.

[51] Shraga Shoval and Johann Borenstein. Using coded signals to benefit from ultrasonic sensor crosstalk in mobile robot obstacle avoidance. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2879–2884. IEEE, 2001.

[52] Håkon Skjelten. Fjernnavigasjon av lego-robot, 2004.

[53] Randall C Smith and Peter Cheeseman. On the representation and estimation of spatial uncertainty. *The international journal of Robotics Research*, 5(4):56–68, 1986.

[54] Bruno Steux and Oussama El Hamzaoui. tinyslam: A slam algorithm in less than 200 lines c-language program. In *Control Automation Robotics & Vision (ICARCV), 2010 11th International Conference on*, pages 1975–1979. IEEE, 2010.

[55] Bruno Steux and Oussama El Hamzaoui. tinyslam source code. 2010. Accessed: 16/05/2013.

[56] Bjørn Syvertsen. Autonomous legorobot, 2006. Pre-project.

[57] Bjørn Syvertsen. Autonomous legorobot, 2006.

[58] Richard Szeliski. *Computer vision: algorithms and applications*. Springer, 2010.

[59] S. Thrun, W. Burgard, D. Fox, et al. *Probabilistic robotics*, volume 1. MIT press Cambridge, MA, 2005.

[60] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[61] Jens Kr Tøraasen. Mapping of an environment using an autonomous robot and a webcam, 2009.

[62] Jannicke Selnes Tusvik. Fjernstyring av legorobot, 2009.

[63] Prahlad Vadakkepat, Tong Heng Lee, and Liu Xin. Application of evolutionary artificial potential field in robot soccer system. In *IFSA World Congress and 20th NAFIPS International Conference, 2001. Joint 9th*, pages 2781–2785. IEEE, 2001.

[64] Built With. jquery usage statistics. `http://trends.builtwith.com/javascript/JQuery`, 2013. Accessed: 05/08/2013.

[65] Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *International journal of computer vision*, 13(2):119–152, 1994.

Appendix A

Configuration

Configuration parameters for various modules of the software are presented throughout the thesis. This appendix presents the most important configuration parameters.

Web Interface

`realm`
: Realm for authentication.
Example: TigerSLAM

`username`
: Username for authentication.
Example: root

`passphrase`
: Passphrase for authentication.
Example: ***`$1$dlPL2MqE$oQmn16q49SqdmhenQuNgs1`*** *("hello")*

Sensors

`lidar_num_distances`
: Number of distances to measure from the LIDAR.
Example: 681

`lidar_radial_span`
: The angular span of the LIDAR measurements, in degrees.

Example: 240

`lidar_max_distance`

Maximum distance the LIDAR can measure, in millimeters. SLAM algorithms can use this information.

Example: 5600

`lidar_position_y`

Lateral displacement of the LIDAR to the robot body, in meters.

Example: 0.0

`lidar_position_x`

Distance from center of robot to the LIDAR, along the forward direction, in meters.

Example: 0.343

Robot Model

`robot_base_width`

Base width of the robot, from wheel to wheel. Based on models for differential drive robots. Defined as base width between encoder wheels for this thesis.

Example: 0.389

`robot_wheel_radius`

Radius of the encoder wheels, in meters.

Example: 0.0505

`robot_odometry_ppr`

Number of pulses per revolution of the odometric wheels.

Example: 800

TinySLAM Specific

`tinyslam_sigma_xy`

Variance in spatial dimensions for the TinySLAM Monte Carlo search.

Example: 0.30

`tinyslam_sigma_theta`

Variance in the angular dimension for the TinySLAM Monte Carlo

search.
Example: 0.07

`tinyslam_hole_width`
Width of the holes which TinySLAM produces in maps and uses for the Monte Carlo search.
Example: 350

`tinyslam_montecarlo_iterations`
Number of iterations which the TinySLAM search should perform.
Example: 1000

`tinyslam_gridmap_size`
Size of the TinySLAM grid map. Product of the length of the two sides, in meters.
Example: 16384 (128×128*)*

`tinyslam_gridmap_resolution`
Resolution in cells per meter for TinySLAM.
Example: 100

HectorSLAM Specific

`hectorslam_gridmap_size_x`
Grid maps' sizes in x direction.
Example: 4096

`hectorslam_gridmap_size_y`
Grid maps' sizes in y direction.
Example: 4096

`hectorslam_gridmap_resolution`
Resolution of maps in length of one cell in meters.
Example: 0.0125

`hectorslam_gridmap_start_x`
Origin of maps in x direction, in fraction of the map.
Example: 0.5

`hectorslam_gridmap_start_y`
Origin of HectorSLAM maps in y direction, in fraction of the map.
Example: 0.5

`hectorslam_levels`
: Number of levels (image pyramid like) maps should include.
Example: 4

`hectorslam_update_factor_free`
: Update factor for a cell when it is observed to be free.
Example: 0.35

`hectorslam_update_factor_occupied`
: Update factor for a cell when it is observed to be occupied.
Example: 0.9

`hectorslam_map_update_min_angle_diff`
: Update the map if the robot has rotated so many radians.
Example: 0.20

`hectorslam_map_update_min_dist_diff`
: Update the map if the robot has moved so far in meters.
Example: 0.40

`hectorslam_use_odometry`
: Consider odometry while performing SLAM.
Example: on

Collision Avoidance System

`collision_detection_radius` Minimum distance from the LIDAR for obstacles to be considered in the collision area.
Example: 0.4

`collision_detection_angle` Radius of the sector which is considered for collision avoidance, in radians.
Example: 3.14

Lookahead Guidance System

`lookahead_distance`
Example:

`lookahead_p`
Example:

`lookahead_u`

Example:

`lookahead_tdelta`

Example:

A* Path Planning

`astar_max_iterations`

Maximum number of iterations to perform when planning a path.

Example: 1000000

`astar_unknown_punish`

Punish factor for unknown areas.

Example: 100.0

`astar_smoothing_data_weight`

Weight given for retaining the original path.

Example: 0.5

`astar_smoothing_smooth_weight`

Weight given for smoothing the path.

Example: 0.4

`astar_check_radius`

Minimum distance to any obstacle for paths planned, in meters.

Example: 0.4

`astar_shrink_factor`

Enlarge cells by this (integer) factor when producing the binary maps before planning paths. Reduces the number of cells accordingly.

Example: 4

Appendix B

API Calls

This appendix lists the calls of the web API as currently implemented. The API calls are used by the client side of the web interface, and can also be used by other applications.

The API is separated into a streaming log API, get and set methods. Get methods retrieve information, while set methods manipulate data or state.

Note: The API is, like the rest of the web interface, password protected. Authentication is done by *basic access authentication*, with username and password configurable (see Appendix A).

Data Streaming API

A data streaming API provides log updates over a continuous *websocket* connection. The API URL is `/api/streaming/log/`. Log messages are separated by newline characters.

SLAM

Set Methods

`initialize`

Parameters: `algorithm string`

Effects: Initializes the SLAM algorithm with the given algorithm. Fails if SLAM is already initialized.

Return: OK on success, error on failure. *URL:/api/set/slam/initialize*

`initialize-from-stored-map`

Parameters: `algorithm string, filename string`

Effects: Initializes the SLAM algorithm with the given algorithm and the given stored map, identified by file name. Fails if SLAM is already initialized.

Return: OK on success, error on failure. *URL:/api/set/slam/initialize-from-stored-map*

`start`

Parameters: –

Effects: Starts the initialized SLAM algorithm, fails if already started.

Return: OK on success, error on failure. *URL:/api/set/slam/start*

`stop`

Parameters: –

Effects: Stops the started SLAM algorithm, fails if already stopped.

Return: OK on success, error on failure. *URL:/api/set/slam/stop*

`terminate`

Parameters: –

Effects: Terminates (un-initializes) the initialized SLAM algorithm. Fails if no SLAM algorithm is initialized.

Return: OK on success, error on failure. *URL:/api/set/slam/terminate*

`save`

Parameters: name string, description string

Effects: Saves the initialized SLAM algorithm's map with the specified name and description (not required).

Return: OK on success, error on failure. *URL:/api/set/slam/save*

Get Methods

`stats`

Parameters: –

Return: JSON object with SLAM state and estimated position, fails if

SLAM is not initalized.
URL: `/api/get/slam/stats`

`image/full`

Parameters: –
Return: PNG encoded image of the full map, fails if SLAM is not initialized.
URL: `/api/get/slam/image/full`

`image/tile`

Parameters: `zoomLevel int, tileX int, tileY int`
Return: PNG encoded 256×256 pixels image of tile at tile coordinate (`tileX`, `tileY`), with zoom level `zoomLevel`. The number of tiles in each direction for any map is decided by $n = 2^{\text{zoomLevel}}$. For more information, see Google Maps JavaScript API Documentation[1]. Fails if SLAM is not initialized.
URL: `/api/get/slam/image/tile`

Map Storage

Set Methods

`mapname`

Parameters: `filename string, newname string`
Effects: Renames map with filename `filename` to `newname`.
Return: OK on success, error on failure.
URL: `/api/set/mapstorage/mapname`

Get Methods

`package`

Parameters: `filename string`
Return: Entire map with meta data and thumbnail. Error if no such filename exists.
URL: `/api/get/mapstorage/package`

[1]`https://developers.google.com/maps/documentation/javascript/`

`metadata`

Parameters: `filename string`

Return: JSON object containing map meta data. Error if no such filename exists.

URL: `/api/get/mapstorage/metadata`

`thumbnail`

Parameters: `filename string`

Return: PNG encoded thumbnail image of map. Error if no such filename exists.

URL: `/api/get/mapstorage/thumbnail`

Sensors

Set Methods

`mapname`

Parameters: `filename string, newname string`

Effects: Renames map with filename `filename` to `newname`.

Return: OK on success, error on failure.

URL: `/api/set/mapstorage/mapname`

Sensors

Set Methods

`connect`

Parameters: `sensor string`

Effects: Connects to the sensor.

Return: OK on success, error on failure.

URL: `/api/set/sensors/connect`

`disconnect`

Parameters: `sensor string`

Effects: Disconnects the sensor.

Return: OK on success, error on failure.

URL: `/api/set/sensors/disconnect`

`start`

Parameters: `sensor string`

Effects: Commands the sensor to start producing and distributing measurements.

Return: OK on success, error on failure.

URL: `/api/set/sensors/start`

`stop`

Parameters: `sensor string`

Effects: Stops the taking and distributing measurements.

Return: OK on success, error on failure.

URL: `/api/set/sensors/stop`

Sensor Logs

Set Methods

`delete`

Parameters: `logname string`

Effects: Deletes the log with filename `logname`. Fails if no such log is found.

Return: OK on success, error on failure.

URL: `/api/set/sensorlogs/delete`

`rename`

Parameters: `logname string, newname string`

Effects: Renames the log with filename `logname` to `newname`. Fails if file is not found.

Return: OK on success, error on failure.

URL: `/api/set/sensorlogs/rename`

`start-logread-realtime`

Parameters: `logname string`

Effects: Starts distributing data from the log at the same speed at which it was acquired. Fails if the log is not found.

Return: OK on success, error on failure.

URL: `/api/set/sensorlogs/start-logread-realtime` `stop-logread-realtime`

Parameters: –

Effects: Stops the distribution of data from log.

Return: OK on success, error on failure.

URL: `/api/set/sensorlogs/stop-logread-realtime`

Motor

Set Methods

`speeds`

Parameters: `left number, right number`

Effects: Sets the relative speeds $[-1, 1]$ of both motors.

Return: OK on success, error on failure.

URL: `/api/set/motor/speeds`

`planpath`

Parameters: `x number, y number`

Effects: Plans path from robot's current position to world coordinate (`x, y`). May take time.

Return: OK on success, error on failure.

URL: `/api/set/motor/planpath`

`followpath`

Parameters: –

Effects: Triggers guidance on current planned path. Fails if no path is planned.

Return: OK on success, error on failure.

URL: `/api/set/motor/followpath`

`stoppathfollowing`

Parameters: –

Effects: Stops guidance.

Return: OK on success, error on failure.

URL: `/api/set/motor/stoppathfollowing`

`deletepath`

Parameters: –

Effects: Deletes the current planned path.

Return: OK on success, error on failure.

URL: `/api/set/motor/deletepath`

Get Methods

`path`

Parameters: –

Return: JSON path object. Fails if no path is planned.

URL: `/api/get/motor/path`

Appendix C

Disc Contents

This thesis comes with a DVD disc. The contents are described below.

TigerSLAM

Folder containing a compiled version of the program. The program is compiled for 64 bit Windows platforms. The assets folder bundled with the program contains illustrative sensor logs and ready-made maps. The sensor logs can be used for testing and further development. *Important note:* In order to use the program, the "assets root" parameter of the configuration file must be set.

Videos

This folder contains a video produced to highlight some of the features of the system, as an introduction or short summary of the project. The video shows how maps are built and demonstrates guidance abilities. The video is also available at *YouTube*[1]. Additionally, this folder contains some of the raw material for this video: videos of the mapping process in real-time.

Source

The source code of the system. Note that this source code is also available at *Bitbucket*: `bitbucket.org/mikaelbe/tigerslam`. The BitBucket repository should be the preferred source of the code. The code is provided on disc for documentation.

[1] `http://youtu.be/66rKR5-uYZ4`

Map Images

This folder contains full-resolution images of some of the maps presented in Chapter 5.