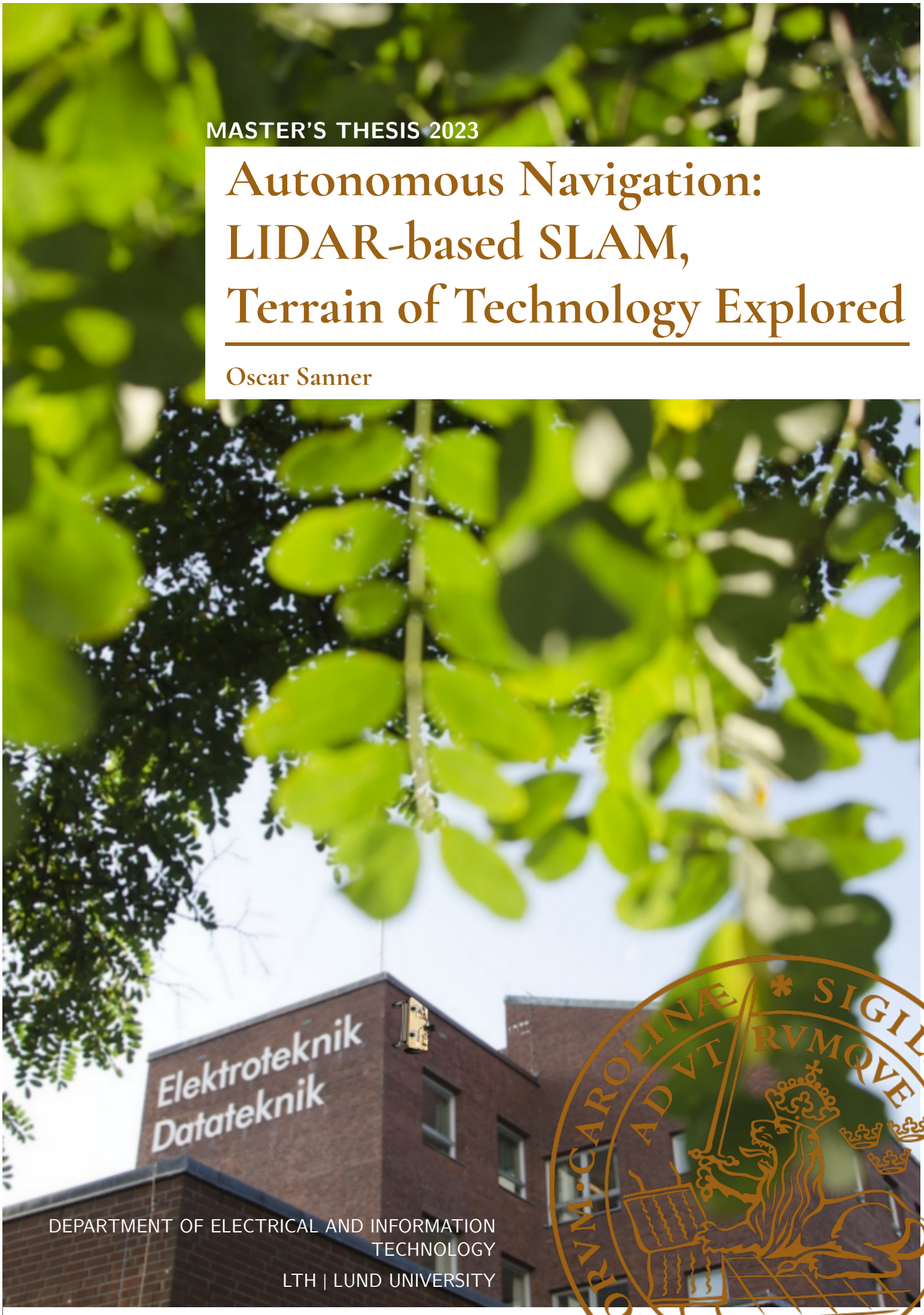
MASTER'S THESIS 2023

Autonomous Navigation: LIDAR-based SLAM, Terrain of Technology Explored

Oscar Sanner



DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
LTH | LUND UNIVERSITY

EXAMENSARBETE
Elektro- och Informationsteknik

Autonomous Navigation: LIDAR-based SLAM, Terrain of Technology Explored

Autonom Navigering: LIDAR-baserad SLAM, Teknikens Terräng Utforskad

Oscar Sanner

Autonomous Navigation: LIDAR-based SLAM, Terrain of Technology Explored

Oscar Sanner
oscar.sanner@gmail.com

June 7, 2023

Master's thesis work carried out at Department of Electrical and Information Technology, Lund University.

Supervisor: William Tärneberg, william.tarneberg@eit.lth.se
Examiner: Maria Kihl, maria.kihl@eit.lth.se

Abstract

In this thesis, an autonomous system capable of navigating and mapping unknown environments is developed. The solution uses a family of algorithms called SLAM or simultaneous localization and mapping, capable of mapping the environment and retaining accurate position data without external sensors such as GPS. Firstly four different SLAM algorithms are implemented and then four different pathing algorithms are tested with a generated map. Everything runs on a hoverboard-based robot using an RPI as the processing unit and LIDAR as the only sensor. The performance is evaluated by analyzing the processor utilization, the positional accuracy and the accuracy of the generated map. It is concluded that the RPI has good enough performance to run the program while leaving processing power for other tasks. The achieved positional accuracy is usually better than 10 cm which is a good result given the circumstances. The generated map has a map-resolution dependent accuracy causing an error of less than $10 * \sqrt{2}$ cm between points on the map, it reproduces long distances of >20 m with no further error.

Keywords: MSc, SLAM, Pathing, ICP, FastSLAM, LIDAR

EXAMENSARBETE Autonomous Navigation: LIDAR-based SLAM, Terrain of Technology Explored
STUDENT Oscar Sanner
HANDLEDARE William Tärneberg
EXAMINATOR Maria Kihl

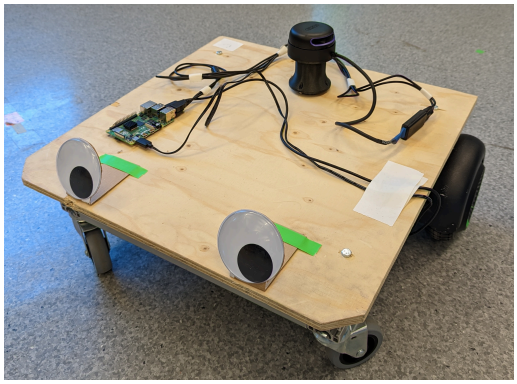
Robot på rull, autonom kartläggning och navigering

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oscar Sanner**

Ett sätt att göra robotar självkörande är genom algoritmer. I detta arbete har jag undersökt i vilka situationer olika algoritmer passar samt hur exakta resultat som kan förväntas på positionsdata och den genererade kartan.

För att en robot ska kunna köra sig själv och planera färdvägar måste den veta en del saker: hur kartan ser ut, var den är på kartan och eventuellt hur den direkta omgivningen ser ut för att undvika saker som inte är med på kartan. I många fall finns en färdig karta, exempelvis en planritning eller så finns ett positioneringssystem som GPS. I vissa situationer finns inget av dessa, då måste roboten själv hålla reda på var den är samtidigt som den gör en karta. En familj av algoritmer som gör detta kallas SLAM (simultaneous localization and mapping). Med hjälp av en eller flera sensorer bygger algoritmen en karta över omgivningen samtidigt som den håller koll på robotens position.
En vanlig sensor som används till detta är LIDAR, genom att använda laser mäter den avståndet i olika riktningar, till exempel kan den säga att rakt fram är det fem meter innan det tar stopp, om liknande mätningar görs flera hundra gånger runt i ett varv går det att få en bra uppfattning om hur omgivningen ser ut.

I mitt examensarbete har jag implementerat två olika typer av SLAM algoritmer med olika syften. Den ena används för att köra i en förutbestämt bana samtidigt som den stannar för hinder som rör sig, exempelvis människor eller andra robotar. Den andra typen kör sig själv efter en bana den själv genererar och bygger en karta över omgivningen. I syftet att bygga kartan antar den att alla hinder är statiska men roboten krockar inte även om saker rör på sig.



Algoritmerna testas genom att köra en bana och undersöka hur stort felet i positioneringen är. Den genererade kartan analyseras också för att undersöka hur bra den stämmer överens med verkligheten. Slutligen analyseras hur mycket beräkningskraft som behövs för att köra programmet.

Resultaten visar att det är fullt möjligt att navigera och kartlägga utrymmen inomhus med en enkortsdator. Felet i positionering är ca. 10 cm eller bättre och det maximala felet mellan två punkter på den genererade kartan är 15 cm.

Acknowledgements

I would like to express my gratitude to my supervisor, William Tärneberg, whose positive outlook and relevant insights greatly influenced the progression of my work. His continuous encouragement and interesting discussions played a crucial role in the success of this thesis.

Oscar Sanner
Lund, Sweden, 30th May 2023

Contents

Chapter 1
Introduction

Self-driving robots have had an explosive evolution during the last 20 years. Today they are used in a wide range of fields ranging from transportation of goods in factories or logistic warehouses to self-driving people carriers or human-interacting healthcare robots. What all of these have in common is that they need a way to traverse the environment they operate in. To do this the robot needs some kind of map to position itself on and a way of sensing the current position on the map. If it isn't guaranteed that the map is static and correct, additional obstacle avoidance is also required. In most cases, the map will not be static and moving objects such as humans, other robots or animals might be present.

To make things simpler it is usually possible to assume at least one of two things. The map may be known, this map does not have to include moving objects but only the stationary scene. What this known map means is that the robot only needs to localize itself within the map, an example of this is described in [7].

The other case is when the pose of the robot is known at all times, the pose is a unique way of describing the state of the robot. For a robot operating on a 2D plane the pose has three attributes, (x, y, θ) where x and y describe the position and θ the heading of the robot. The pose can be acquired by receiving signals, for example, laser pulses or radio signals from sensors with known positions or satellites. One example of this is an automatic lawn mower that uses GPS to get the pose and using that it can build a map of the area it operates in to cut the grass efficiently.

These cases assume that either the map or the pose is known. What can be done if both of these are unknown? One solution commonly applied in robotics is called SLAM (Simultaneous Localization And Mapping). This is a solution to the problem where an unknown environment is traversed at the same time as a map is built all while keeping track of the current pose. This makes it a combination of two problems, creating a map from sensor data assuming the pose is known and getting a pose estimate from a map. What makes SLAM so difficult is similar to the "What came first the chicken or the egg?" problem [8]. A map is needed to know the pose and the pose is needed to build the map. There exist different types of SLAM algorithms to solve this problem and they are usually split into two categories,

landmark-based and map-based. Landmark-based SLAM variations focus on positioning while building a map of landmarks. A landmark should be something easily recognizable and stationary, examples of this are walls, corners or vertical poles. Map-based SLAM does not use landmarks but instead builds a map of all environment features. To do this memory and computationally effective the world is often discretized into chunks. Depending on the expected size of the map and required resolution the size of the chunks can range between a couple of mm to meters. In this work, both types of SLAM are implemented and analyzed.

1.1 Research questions

- **RQ1:** Is it possible to run a SLAM algorithm implemented in Python with the limited computing power a single board computer like a Raspberry Pi has?
- **RQ2:** How accurate positioning can be achieved?
- **RQ3:** How accurate is the generated map?

1.2 Contributions

This thesis aims to make the following contributions, all implemented in Python and capable of running without ROS (Robot Operating System)[4]:

- A SLAM implementation that is capable of producing an accurate map in a static environment.
- A self-driving system that can plan paths and drive within a previously generated map while adapting for new obstacles.
- A landmark-based SLAM algorithm capable of following a predetermined path in a dynamic environment without crashing.

Chapter 2
Background

2.1 Elements needed to perform SLAM

To use a SLAM algorithm in a real-world application several things are needed.

- A robot that can be controlled.
- Sensors on the robot that can provide information about the surrounding environment.
- Processing unit to run the SLAM algorithm on.
- SLAM algorithm.
- Pathing and control.

These items can be divided into hardware: robot, sensors and processing unit and software: control, pathing and SLAM algorithm.

2.1.1 Hardware

Robot

The robot is based on a hoverboard that uses a differential drive system to move. A differential drive system has two drive wheels that can move independently, this results in a very maneuverable robot that requires two control signals, the speed of both wheels which can be positive or negative. To keep balance two additional caster wheels are used which provide a very stable platform. The hoverboard is made to move humans and is therefore capable of moving a lot of weight, the rebuilt hoverboard has no problem driving heavy loads of around 75kg on flat ground. The robot with everything required attached is shown in Figure 2.1 Hoverboards also need very precise control and to achieve this they have wheel encoders that give

an accurate value of the position of the wheels, these sensors are crucial in this implementation to provide rough positioning. Brushless motors are built into the wheels, see Figure 2.2 and require a three-phase motor driver. An ODrive [2] controller is used for motor control which has an easy-to-use Python library.

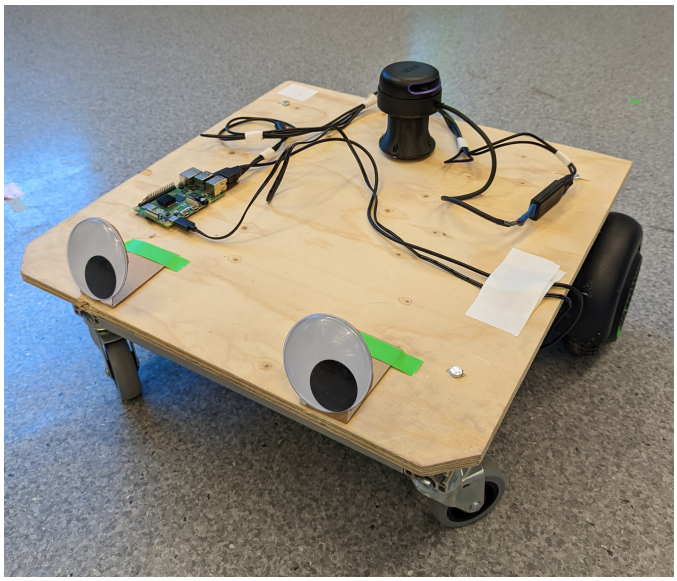


Figure 2.1: The robot, named "Oliver" by chatGPT.

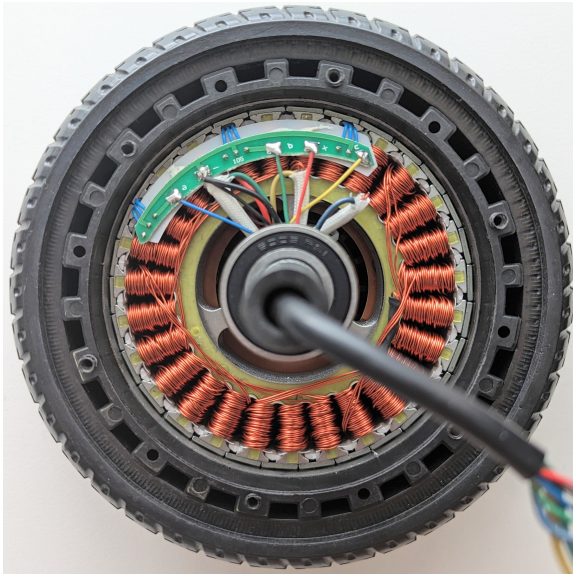


Figure 2.2: Inside of a hoverboard wheel.

Sensors

The only sensor that is used except for the wheel encoders is a LIDAR, specifically an RPLIDAR A3, it is a relatively cheap 2D LIDAR that produces measurements in 360 degrees. It is capable of producing a maximum of 16000 measurements per second and has a max range of around 25 meters, depending on the surface it is measuring against.

Processing unit

The code runs on a Raspberry Pi 4B which is a very capable and popular single-board computer. It interfaces with the LIDAR using two USB ports, one for power and one for communication. The ODrive is controlled using another USB port.

2.1.2 Software

The map-based algorithm works in two stages, the exploration stage and the driving stage. During the exploration the robot tries to build a complete map of the environment, when the exploration is done it enters the driving stage where it can drive to any reachable point on the map. The landmark-based algorithm does not explore the environment actively and only conducts the driving stage. This means that it can follow a predetermined path consisting of a sequence of points it should drive to, while doing this the robot will retain good positional data using the SLAM algorithm.

SLAM

The SLAM algorithm needs several things to work at its best. Firstly a rough guess of the robot's pose will help the algorithm greatly, this guess is usually obtained by odometry. Odometry is a process that can be used to estimate the motion of a robot by analyzing the movements of the wheels or other sensors. For wheeled robots and especially differential drive systems that don't angle the wheels for steering this is a simple problem. It only requires the distance traveled by both drive wheels to get the change of the pose. For landmark-based SLAM there needs to be a system to get the position of landmarks, this can be done in many ways and the examples implemented in this work search for corners or walls. Map-based SLAM does not utilize individual landmarks but instead does the localization by matching different measurements, in this work the measurement matching is done by the iterative closest point algorithm.

Pathing and control

The output from the SLAM algorithm is a map and pose estimate. These themselves do not make the robot move, to do that a pathing algorithm in conjunction with a control system is used. How the pathing should be performed during the map building depends on the intended usage of the robot. It might have a requirement such as it should start close to one point and explore outwards or have prioritized areas. In this thesis, this is done in different ways for landmark-based SLAM and map-based SLAM. For the landmark-based alternative, a precomputed path is used and no path planning algorithm or exploration is conducted, the control algorithm simply follows the predetermined path. For the map-based alternative, the

robot explores actively with no predetermined path. This is done by setting the goal of the pathing algorithm to be outside the map, this returns a path that is currently viable. If the robot operates in a closed area it will find an obstacle somewhere along the path which makes the path undrivable. When that happens a new path that leads outside the map is created. When it is impossible to create a path having the goal outside the map the area is closed and the robot enters the driving stage where it can drive between points.

A path can be described in many different ways, one common way is to have a sequence of points, following the points one by one results in the robot driving along the path. Driving from the current robot pose to another point can be accomplished in many different ways, in this implementation a MPC (Model Predictive Controller) is used [17]. It looks a certain number of points into the future and by testing different control signals in software it tries to minimize a cost function. This cost function can be made to rely on many different factors to tailor the controller to a specific use case.

2.2 Related work

2.2.1 Robot Motion Model

The robot has a differential drive system. This means that it has two drive wheels that can spin independently of each other, the kinematics are described in [16]. A movement of the robot is recorded as a change in the distance traveled by one or both of the drive wheels, this change can be positive or negative. If the robot starts and moves one meter forward and then takes a measurement the wheel encoder position will be one meter and the change since the last position one meter. If the robot then reverses one meter and takes a second measurement the wheel encoder position will be 0 meters and the change since the last measurement negative one meter.

Pose update

To update the pose of the robot all three pose attributes have to be updated, (x, y, θ). The pose update begins by changing θ with equation 2.1. In the equation, the wheelbase is the width between the contact patches for the drive wheels and ΔR and ΔL is the change in distance traveled by the left and right wheels since the last pose update.

$$\Delta\theta = tan(\frac{\Delta R - \Delta L}{wheelbase}) \tag{2.1}$$

Next, the position is updated where the x and y values are updated with equations 2.3 and 2.4. Here D is the total distance traveled by the center of the drive wheels on the robot calculated with equation 2.2.

$$D = \frac{\Delta R + \Delta L}{2} \tag{2.2}$$

$$\Delta x = D * cos(\theta) \tag{2.3}$$

$$\Delta y = D * sin(\theta) \tag{2.4}$$

2.2.2 MPC

A MPC works by associating different costs to different parameters and by searching, finding the input signals that produce the lowest total cost [17]. The parameters associated with the cost can include control signals, robot state and resulting motor speeds among others. In this thesis, the function that searched for the input signals that produced the lowest cost is the optimize.minimize function from the Python package Scipy [6].

2.2.3 Landmark-based strategies

Extended Kalman Filter

Even if EKF (Extended Kalman Filter) SLAM is not implemented it is included here to give some reference as it is one of the oldest SLAM methods. One of the first to implement SLAM was Philippe Moutarlier and Raja Chatila in 1989 [14] who used the EKF method and at the time achieved very promising results. One problem with EKF is the computational complexity. Each update requires quadratic time regarding the number of landmarks, this is because it links each landmark to all other landmarks. Linking all landmarks to each other might also result in issues if a wrong association between a sensor reading and the corresponding landmark is made as it might affect several other landmarks.

FastSLAM 1

FastSLAM 1 was presented by M. Montemerlo et al. in 2002 [12] and has several advantages compared to older algorithms like EKF SLAM. Both algorithms use the same probabilistic motion model described in Section 3.1.1 as well as the same measurement model described in Section 3.2.4. The main difference is that EKF applies a filter to a high dimension problem whereas FastSLAM 1 splits the filter into many 2x2 EKFs. This makes the computational complexity much better, another major stability improvement is the use of a particle filter described in Section 3.6. The particle filter makes it possible to model multi-modal distributions compared to EKF which is limited to Gaussian distributions. An example of this is shown in Figure 2.3 and 2.4 where z is the probability that the robot is at any (x,y) point. With EKF it is only possible to model the single-modal distribution and not the multi-modal distribution unless a very large variance is used to cover both peaks. With the introduced particle filter FastSLAM 1 can handle both situations effectively by splitting the particles between the peaks.

FastSLAM 2

FastSLAM 2 [13] is similar to FastSLAM 1 but with one important difference. FastSLAM 1 only uses the new odometry data when applying the particle filter and sampling the new poses, FastSLAM 2 improves the sampling by accounting for the current sensor measurements. This means that it can place more of the particles where the robot is most likely to be. As a result, fewer particles can be used to achieve the same accuracy alternatively, the same number of particles can be used to get higher accuracy.

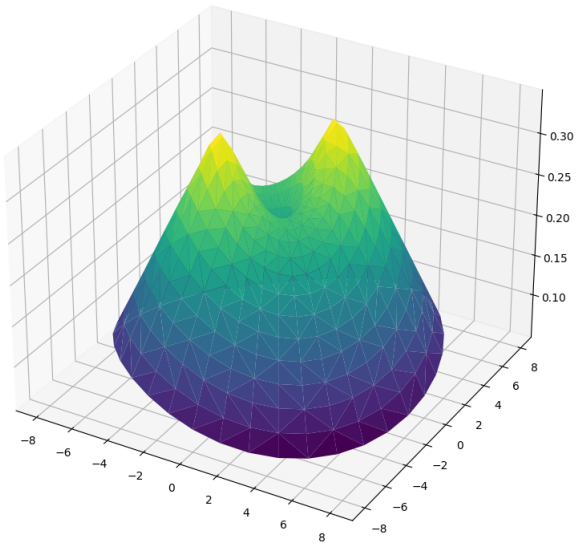


Figure 2.3: Example of a multi-modal distribution.

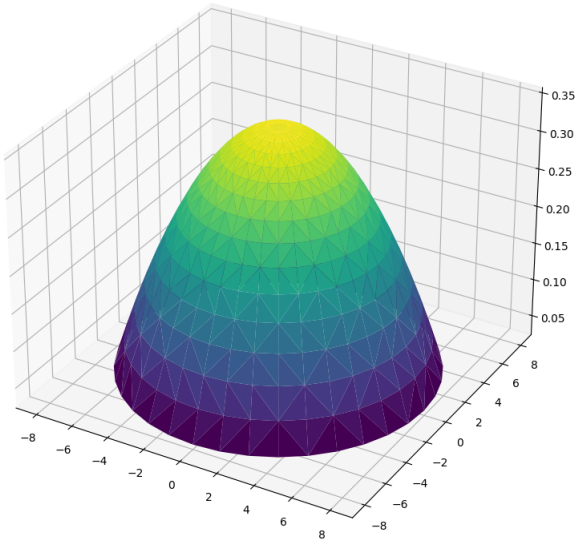


Figure 2.4: Example of a single-modal distribution.

2.2.4 ICP

The map-based SLAM algorithms use ICP (Iterative Closest Point) to match the LIDAR scans against each other, ICP works iteratively with the following steps:

1. For each point in the source point cloud, find the closest point in the reference point cloud.
2. Find a translation and rotation of the source that minimizes some function between all matched points. A common function is the root mean square distance between the points.
3. Transform the source according to the best translation and rotation.
4. Goto step 1 until a maximum number of iterations or no improvement is found.

The ICP function is not implemented from scratch but the function registration_icp from the library open3D is used [19].

2.2.5 Particle filter

The FastSLAM implementations use a particle filter [13] where each particle represents a simulated robot that contains a pose and data regarding all landmarks. Since it is possible to have any amount of these virtual robots probability distributions of any kind can be represented, for example, the distribution shown in Figure 2.3 can be modeled with half of the particles centered around each peak. A robot working on a 2D plane is also particularly fitting to use with a particle filter since it has a low dimension of only three state variables, (x, y, θ). With more dimensions, the number of particles required increases a lot. Disregarding the angle of the robot it would require 10 000 particles to cover a 1m x 1m 2D world with a resolution of 1 cm. Doing the same for a 1m x 1m x 1m 3D world the number of particles increases by 100 times to 1 000 000. The particle filter works in three steps: sample from the proposal, create importance weights and lastly, resample the particles and replace the unlikely particles with more likely ones, this is similar to survival of the fittest.

2.2.6 Pathing algorithms

RRT*

RRT* is an evolution of RRT (Rapidly exploring Random Tree)[11]. Random points are generated and paths are created by connecting points where the line between the points does not pass through obstacles. Compared to RRT, RRT* adds a cost value equal to the distance to its parent. When a new node is generated it is not only the closest existing node that is checked but also all existing nodes inside a fixed radius. The connection is made with the node that will yield the lowest total cost. RRT* tends towards the optimal path as the number of points goes towards infinity. However, due to practical limitations, it will in most cases not produce an optimal path. Also, due to the random nature of RRT* it does not guarantee it will find a path even if one exists. In most cases, a path will be found but if the environment contains long very narrow sections it might fail to find a path depending on the placement of the random points.

A*

A* is an optimal search algorithm that searches by the "best-first" rule [9]. It will continue iterating until it finds a path to the goal or until the queue of possible next steps is empty and in that case, no path to the goal exists. During each iteration, the node with the lowest total cost is chosen to be expanded.

Phi*

Phi* is a variant of A* and similarly finds one of the optimal paths [15]. One advantage Phi* has compared to A* is that it has the functionality to replan an already planned path. This makes Phi* better if frequent replanning is required.

2.3 Why Python?

Python is usually regarded as a slow language compared to other languages like C or C++. Python is an interpreted and dynamically typed language, in short, this means that the source code gets compiled as it is needed while running, slowing down execution. Furthermore, a Python process can never run in more than one thread simultaneously, this can however be bypassed by starting multiple different Python processes. The reason Python was chosen is the huge support it has in the robotics and hardware community. For most robotic hardware such as LIDARs, motors, cameras, PlayStation controllers etc. there exist easy-to-use Python packages that can be installed by a package manager with one single command.

Chapter 3

Implementation

3.1 Odometry

The first step to efficiently use SLAM algorithms is a rough guess of the pose. One common way of acquiring this guess, and the one used in this work is to utilize wheel encoders and odometry. The hoverboard motors have 30 permanent magnets with alternating polarity which result in 15 pole pairs. Three hall effect sensors are used and each can output a 0 or a 1. The combined state for all three sensors repeats for each pole pair, two full revolutions are shown in Figure 3.1. The states (0,0,0) and (1,1,1) are not used because of the placement of the sensors. With six states per pole pair and 15 pole pairs the total amount of state changes per wheel revolution is $15 * 6 = 90$. The wheel circumference is around 500 mm which means that the encoder has a resolution of $500mm/90 = 5.6mm$. Since the motor controller knows the angular velocity of the wheels it can predict an even more accurate value. If the angular velocity is $2\pi\ rad/s$ and the last encoder state change was 5ms ago the wheel should have traveled $2\pi * \frac{5}{1000} = 0.03rad$ after the state change. The Odrive controller can account for this when calculating the distance traveled.

The pose is updated as described in Section 2.2.1. This has to be done frequently since it assumes the robot has traveled linearly since the last update. If too much time passes between pose updates the odometry information becomes inaccurate quickly. For example, the robot might have a difference in encoder position of $\Delta R = 0.5$ and $\Delta L = 0.5$, if the maximum speed of the robot is 1 m/s this movement can be done in 0.5 seconds. If both wheels have the same speed throughout the movement the robot will travel straight forward. If the robot starts with only turning the right wheel 0.5 meters and then stops the right wheel and lets the left wheel travel 0.5 meters it will move diagonally forward and left.

To keep the accuracy of the odometry it is therefore very important to run the pose update frequently compared to the maximum wheel velocity or acceleration.

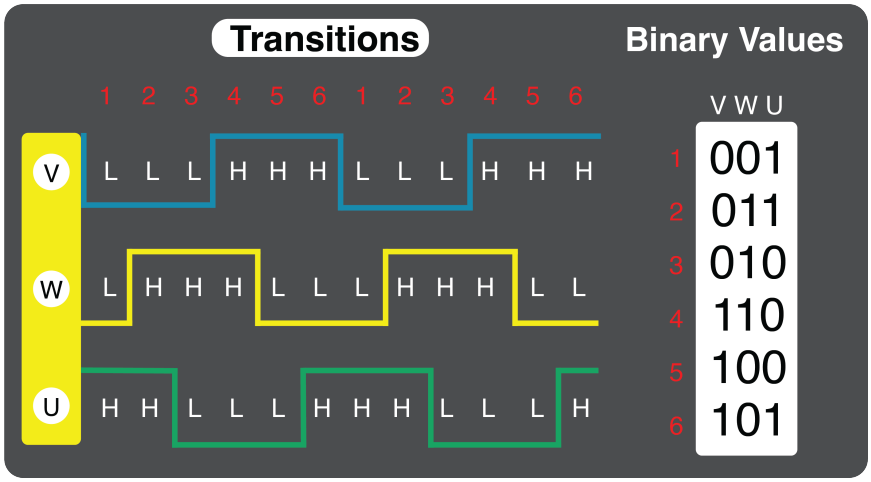


Figure 3.1: Diagram of the hall sensor states that repeat for each pole pair.

Wheel circumferences

To get good odometry results it is also important to know the diameter of the wheels accurately since that value is used when translating the pulse readings from the wheel encoders into the distance traveled by the wheels. To measure the wheel diameter the robot traveled 10 meters in a straight line and the wheel encoder positions were analyzed. The 10-meter distance was measured with both laser and measuring tape to get as accurate results as possible. The test showed that the two wheels had circumferences of 501 mm and 496 mm.

Wheelbase

When there is a difference in the movement of the right and left wheels the heading of the robot will change, the amount of this change depends not only on the distance traveled but also on the wheelbase of the robot. The wheelbase is the distance between the contact patches of the two drive wheels. It was measured by turning the robot around its axis ten times and then ending in the same pose as it started. If the wheelbase value is correct the final angle should be close to 0 or 2π. This test was conducted iteratively, if the heading after 10 revolutions was less than it should be the wheelbase value was decreased and if the heading was too large the wheelbase value was increased. During each test, the robot spun $20\pi rad$ and after the final iteration, the odometry had an error of $0.15rad$ or around 0.24% which was considered good enough. It is important to conduct this test after the wheel diameters are measured since the change in the forward distance only depends on the diameter of the wheels but the change in angle during rotation depends on both the wheel diameters and the wheelbase.

3.1.1 Probabilistic motion model

The odometry will contain some errors, the following factors might have a big or small impact on the result depending on the environment.

- **Wheel dimensions**, The radius of the wheels is measured quite accurately but there will always be some error. For example, the radius can increase by the wheels picking up small rocks/dust or decrease as a cause of wheel wear.

- **Wheelbase**, The wheels are not perfectly perpendicular, this means that the wheelbase will change a small amount over each revolution.

- **Environment disturbances**, The surface the robot drives on might not be perfect and contains small debris that can make the wheels jump or move a small distance.

- **Slipping**, During acceleration or turning the wheels might slip in reference to the ground, this means that the wheel encoders register the wrong movement.

To combat these inaccuracies a probabilistic motion model is applied to the odometry to get a guess of not only the point where the robot is supposed to be but also a border saying for example, with 95% confidence the robot is inside this area. This is done using Gaussian noise which is added to the distance traveled by the wheels which is done independently for all particles. The mean of the Gaussian is set to be 0 and the standard deviation is a base value added with a factor of how long the wheel has traveled. The reason for the base value is to represent the robot driving over something small and thus changing the position, this might happen even if the robot drives a very short distance and that is the reason it is used. The factor of the distance traveled is used since the longer the robot has traveled the bigger the inaccuracies might be.

3.2 LIDAR

SLAM needs some way of sensing the environment it operates in, this thesis utilizes a LIDAR to provide measurements of the surrounding area. It revolves at 10Hz and with each revolution it outputs around 650 distance-bearing measurements which are combined into a list before being sent to the rest of the program. The bearing for each measurement is acquired by measuring the angle of the rotating part compared to the base of the LIDAR at the time the laser pulse is sent.

To get the distance for a particular heading it uses triangulation, when the reflected laser beam returns to the LIDAR it is focused on an image sensor. Assuming that the relative positions and angles of the laser diode, focusing lens and image sensor are known the reflection angle and distance to the object can be calculated. A smaller angle means that the object is further away, an example of how this works is shown in Figure 3.2.

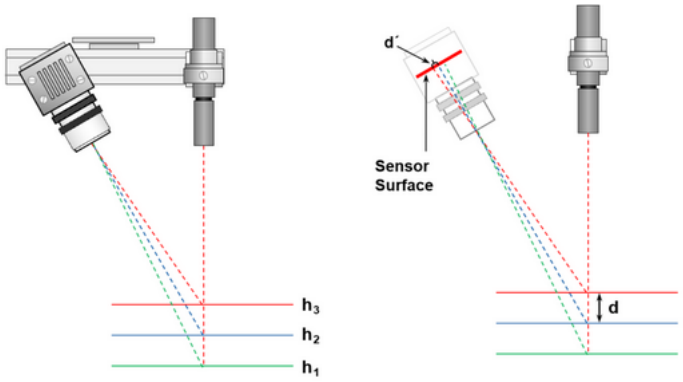


Figure 3.2: LIDAR triangulation principle, image courtesy of [1]

3.2.1 Precision

To investigate if the performance of the LIDAR would affect the result of the SLAM algorithm a test was conducted where the LIDAR measured the range to different materials. The distance between the LIDAR and the test surface was 5 meters. That particular distance was chosen since the robot operates indoors and 5 meters is a common distance the LIDAR will see. The results are presented in Table 3.1.

Material	Number of samples	Mean	Standard deviation
Black cotton	1975	5002.0	6.59
Brown cardboard	1678	5011.0	10.97
Unpainted plywood	2096	5005.2	8.70
Glossy white plywood	1372	5015.35	5.13

Table 3.1: LIDAR accuracy, mean and Std Dev is measured in mm.

3.2.2 Reflective surfaces

The LIDAR sensor works very well with both light and dark surfaces and the detection distance poses no limitation in the indoor environments the robot operates in. However, reflective surfaces pose a large problem for the LIDAR if they exist, when the laser beam is reflected on a surface it will look like there is a copy of the room on the other side of the reflective surface [18]. Mirrors and shiny screens are problematic in most angles, regular glass works depending on the angle, if the laser beam goes directly towards the glass it goes through without a problem but if the angle between the laser and glass gets too small the laser beam will reflect and produce bad measurements.

The robot used is very low and thus the LIDAR is placed around 30 cm above the ground, at this level very few reflective surfaces exist and since it operates in a testing environment the surfaces that cause problems can be moved or covered. An example of this problem can be seen in Figure 3.3, several observations have been done in the middle of the corridor's open space, these observations can be seen as very faded red areas. These are a result of the short top-left wall being a glass door that reflected the laser beam. Small disturbances like this do

not affect the algorithm since an obstacle needs to be seen several times, currently at least 5 to be considered as an obstacle.

Figure 3.3: Example of erroneous measurements in the open space.

3.2.3 Limitations with 2D vision

The RPLIDAR A3 can only measure in a plane. This results in a small amount of measurements that have to be analyzed and reduce the computational complexity compared to a 3D LIDAR but it also means that the robot is restricted to sensing in a plane around 30 cm above the floor. Certain obstacles that are lower than this or obstacles that hang from the ceiling will not be spotted. The localization algorithm has no problem with this as long as sufficiently many landmarks are observable or in the case of the map-based SLAM, enough distinct features exist in the room. The problem is that the robot can run into all obstacles not seen in the plane of the LIDAR.

In Figure 3.4 an example measurement is shown that is the result of the scene in Figure 3.5. The v-shaped obstacle is described accurately by the lidar since it has the same geometry independent of what height it is looked at. The measurement from the office chair marked with "A" does not have the same property. The plane of the LIDAR only intersects the vertical pole of the chair and as such the true width of the chair is not seen. If this data were to be given to the robot it would assume it could drive right next to the measurement which would result in the robot driving into the wheels of the chair, this will be referenced as the office-chair problem. In the space the robot normally operates in almost all obstacles such as shelves, desks, walls or trash cans have the same geometry the lowest 40 cm and thus pose no problems for the robot. However, there are plenty of office chairs and these have to be moved out of the way to avoid crashes.

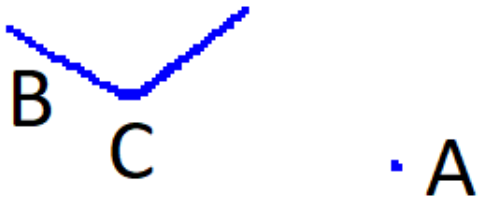


Figure 3.4: LIDAR scan of the scene in Figure 3.5 marked with different kinds of landmarks. The red dot is the LIDAR position.

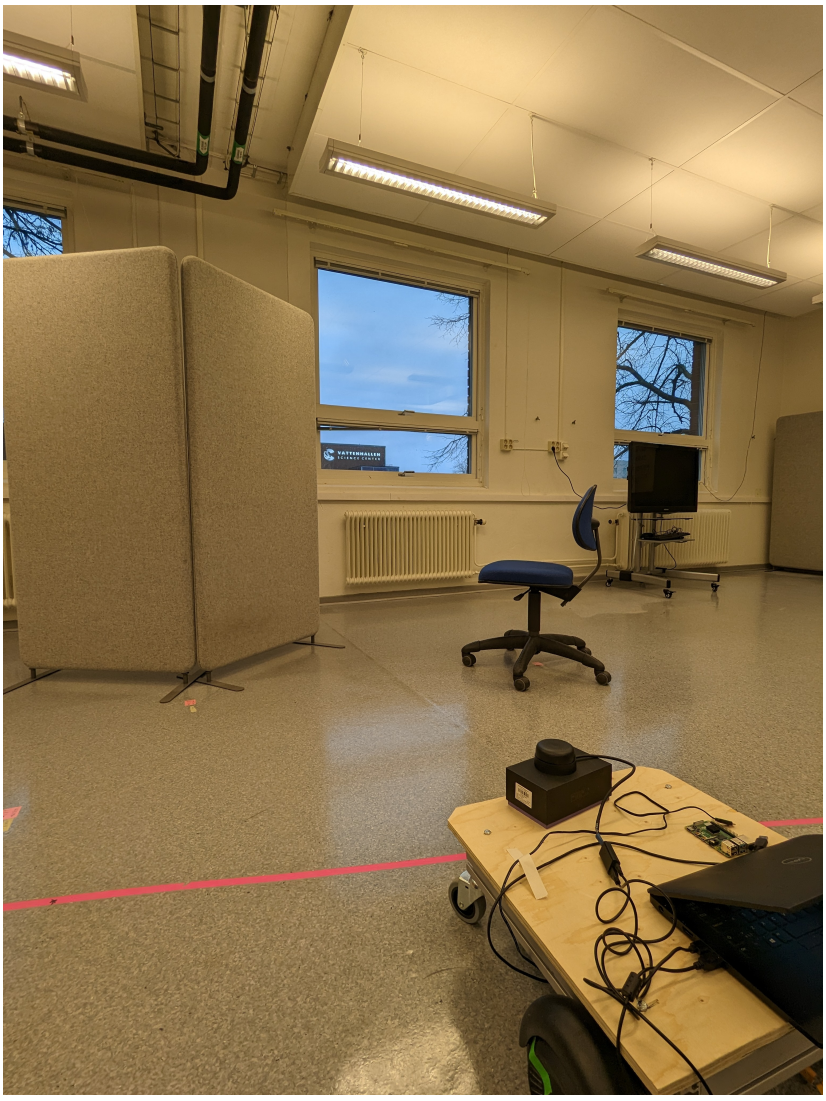


Figure 3.5: Real-world scene that resulted in the measurement shown in Figure 3.4

3.2.4 Measurement model

Similarly to the motion the sensor measurements might have some noise and do not have to be perfect. In the landmark-based SLAM variants, the measurements of a landmark are given an EKF filter for the position. In the map-based SLAM algorithm, the raw data from the LIDAR is used. This works since LIDARs are very accurate and the same reasoning would probably not work if another sensor with a lot of noise was used.

3.3 Landmark extraction

In landmark-based SLAM, landmarks have to be extracted from the sensor data. For something to be considered a good landmark, it should fulfill the following requirements.

- **Easy to spot**. If the landmark is in range of the sensor it should be detected, there should never be a landmark that is only seen once.
- **Differentiable**. It should be possible to differentiate different landmarks from each other, close landmarks might be mixed up.
- **Single position**. All landmarks should only have one position.
- **Stationary**. If a landmark is moved it will influence the SLAM algorithm, a slow-moving landmark might "pull" the robot pose with it and cause large errors.
- **Enough landmarks in the environment**. The more landmarks re-observed at any given time the better the pose estimate. The environment should contain enough landmarks for the robot to get a good pose estimate.

3.3.1 Landmark strategies

As long as the previous requirements are fulfilled a landmark can have many different shapes. One additional soft requirement is that the amount of processing power needed to extract the landmarks from the sensor data should not be too high. The following types of landmarks were thought of as fulfilling all requirements in an indoor environment:

Spike landmarks

Spike landmarks consist of something small that stands out compared to the rest of the environment. Examples of common indoor spike landmarks could be for example table legs or the legs of chairs. Using a ranging sensor such as a LIDAR these landmarks can be spotted by looking for a small region with some distance d to the LIDAR. Both sides of this region should contain some amount of space where all points are significantly further away from the lidar compared to d. An example of a spike landmark is shown in Figure 3.4 marked with "A". Here it is possible to see that the blue dot marked with A has no measurements close and is relatively small, less than 10 cm. This would be a good geometry for a landmark if spike landmarks were used. The advantages of spike landmarks are that they can usually be seen from a large variety of angles and that they are easy to find in the measurements. This

approach for landmark detection was not chosen mainly because the robot will be operated around humans. A human leg closely resembles a spike landmark and that might violate the requirement that landmarks should be stationary. Another weak point of spike landmarks is that the edge of a flat surface will look like a spike.

Wall landmarks

Wall landmarks consist of large flat areas, indoors these mainly come from walls or furniture with flat portions. A good landmark should only have one single position as described earlier. If the whole length of a wall can be seen in one measurement the landmark position can be decided as the middle of the wall. If only part of the wall can be seen then there is an issue. If the position is set as the middle of the current observed wall then the landmark position can jump around depending on what part of the wall is visible. If the edge of a wall is seen this can be used as a landmark. Advantages with wall landmarks include that they are usually plentiful in indoor environments and even if they are a bit more complicated to extract from the data compared with spike landmarks it is still not very hard. The implemented wall landmark extraction algorithm is described in Algorithm 1. In Figure 3.4 two wall landmarks are present, one of them is marked with "B". In this example, the whole length of the wall can be seen and thus a single position can be decided, therefore this would work as a good landmark.

Corner landmarks

Corner landmarks consist of two connected flat surfaces with an angle between them. It is similar to one of the cases in wall landmarks where the edge of the wall is used but adds the additional requirement that it should form an angle with another connected flat surface. This also means that it is computationally more expensive to find corner landmarks as wall landmarks have to be found first and then also combined into corners. In practice, it works by extracting all segments that can be approximated as lines and that have a minimum real-world length and a minimum amount of laser measurements. When all line segments have been extracted it loops over the segments and tries to find corners in connected segments. Not all combinations of line segment pairs have to be checked because of the structure of the measurements. It is only segments that are connected that are interesting and segments can only be connected if they follow each other directly in the sequence. Since the LIDAR outputs data in order only the following segment to the current has to be checked for the angle, the algorithm implemented is shown in Algorithm 2. A corner landmark is shown in Figure 3.4 marked with "C". To get C it is required to see both lines that create the corner but the whole length of the lines does not have to be seen.

This is the technique chosen, even if it requires more processing power the advantages of stable and correct landmark positions outweigh the additional processing time.

Algorithm 1 Line segment extraction algorithm

Require: measurements, min_measurements, min_distance
  walls ← ∅
  for i in measurements **do**
    $j \leftarrow i + min\_measurements$
    while $distance(i, j) < min\_distance$ **do**
      j ← $j + 1$
    end while
    $line \leftarrow (i, j)$
    if $distance(k, line) < \epsilon \; \forall \; k \in \{i...j\}$ **then**
      walls ← $walls + (i, j)$
    end if
  end for

Algorithm 2 Corner landmark extraction algorithm.

Require: line_segments, min_angle max_angle
  corners ← ∅
  for i in line_segments **do**
    $ls\_1 \leftarrow i - 1$
    $ls\_2 \leftarrow i$
    if $max\_angle > angle(ls\_1, ls\_2) > min\_angle$ **and**
      $closest\_distance(ls\_1, ls\_2) < \epsilon$ **then**
        corners ← $corners + corner(ls\_1, ls\_2)$
    end if
  end for

3.4 Landmark SLAM

There exist several different landmark-based SLAM algorithms. To begin both FastSLAM and FastSLAM 2 described in Section 2.2.3 were implemented and tested with the robot.

3.4.1 SimpleSLAM

During the FastSLAM implementation, an effort was made to make it require very little processing power. This removed almost all safety mechanisms of FastSLAM but resulted in a functioning SLAM algorithm with very low CPU requirements. Like the FastSLAM implementations, it uses corner landmarks but that is almost the only similarity. When one landmark has been found it will never be moved or removed. Landmark association works by checking if any existing landmark is within 20 cm of the observation. If that is the case then it is associated with the observation, if no landmark is close a new landmark is created. To perform pose correction an average is calculated between all observation-landmark pairs, if one association suggests that the x error is 5 cm and another 10 cm the correction will be the mean, 7.5 cm. SimpleSLAM also removes the particle filter, which means that it only models one belief of where the robot is further increasing the risk of errors.

3.5 Map-based SLAM

In some situations, a complete map of the environment is needed and in these cases, landmark-based SLAM is suboptimal since it only keeps track of the pose and some amount of landmarks. Landmark SLAM can be used for mapping purposes but needs to be combined with another program that uses the pose and sensor measurements to create the map and in that case, both algorithms have to be well-synced to get good results. Another solution is scan matching which is the approach used in this work. The goal of scan matching is to take at least two different datasets that have some common features and merge them into one unified dataset. This is similar to how phones create panorama photos by stitching together images that are slightly overlapping. In this thesis, the goal is not to match images but instead point clouds or sets of points. This is easier than matching images and there exist several algorithms to do this, one popular variant and the one used in this work is ICP described in Section 2.2.4.

3.5.1 Scan matching adaptations

The scan matching implementation works by converting the (distance, angle) measurements to x and y coordinates using the believed robot pose. The new cartesian coordinates are used as input to the ICP function together with the previous measurements. The ICP function returns a transformation matrix that says how the new measurements should be translated and rotated to match the old measurements the best. The transformation matrix is used for two things, firstly it is used to move the source point cloud to match the reference and secondly, it is used to correct the pose of the robot. When the source has been translated the points are used to update the environment map. The world is discretized into 5x5 cm chunks and it is only the first point that is seen in each chunk that is saved in the reference point

cloud. A smaller chunk size means that more memory and CPU are used but also allows for more precise mapping. To present certainty of the maps each chunk saves a number corresponding to the number of times it has had an observation inside it, this value is used in the visual representation where each chunk becomes a pixel and gets a brighter red the more times it has been seen. The scan matching creates two maps, one is only used for visual data presentation where only the chunks that have been seen are incremented. The other map that is used by the pathing algorithm increments all chunks that are in a radius of 30 cm. This is done because the pathing algorithm treats the robot as a point but since it has a size it cannot drive directly next to an obstacle. The 30 cm border makes sure the robot can drive everywhere regarded as free space by the map. The difference between the two maps can be seen in the results in Figure 4.8 and 4.9.

3.5.2 Dynamic map

When the robot drives along a predetermined path in map-based SLAM it can handle some amount of dynamic obstacles for example humans as the predetermined path can go through the obstacles created by the disturbances. However, dynamic obstacles might lead to difficulties in the scan matching ICP algorithm as each observation of a dynamic obstacle will result in a point being added to the map. If too many dynamic obstacles are observed at different spots the reference point cloud will get cluttered with observations of dynamic obstacles. When the ICP algorithm tries to match the source against this reference the random observations might affect the algorithm and make it output the wrong transformation matrix. The robot has some resilience against this as the ICP algorithm outputs an accuracy value, if this is too low, no points are added to the reference. However, if this value is below the threshold five times in a row the robot believes it has to inaccurate pose data and stops execution and has to be restarted.

When the robot explores the environment and runs the pathing algorithm itself dynamic obstacles will cause problems. A human walking by the robot will create a virtual wall, unpassable for the robot and as no obstacles are removed the wall will stay until the map is reset.

3.5.3 Map data structure

The map is represented by a matrix where each value is the number of times that discretized 5x5 cm chunk has been seen. The map matrix used for all experiments has a size of 600x600, together with a chunk size of 5 cm this results in a usable map of 30x30 m which was just enough to cover the testing environment. Memory wise a map of this size poses no problem even if it unnecessarily uses 64bit values, the total memory requirement of $600 * 600 * 8B = 2.88MB$ is no challenge for most Linux-based computers.

3.6 Particle filter

The pose estimate is just what it sounds like, an estimate that has some levels of inaccuracies and given enough time these minor inaccuracies might sum up to a large error. These inaccuracies can be handled in several different ways, a part of the solution used in FastSLAM 1

and 2 is a particle-based recursive Bayesian estimation. The bayes filter gives the probability of the robot being in a certain pose given the previous pose as well as odometry and LIDAR measurements. What makes the particle filter perform well is that this probability is calculated individually for each of the particles, the implementation in this thesis used between ten and fifty particles. Each particle contains the believed pose and after each movement the motion model described in Section 3.1.1 is applied equally to all particles. This means that all particles get Gaussian noise added individually to the movement. In the end, the probabilities for all of the particles are used as weights to select which particle the robot should base its pose from. This means that there will be a high likelihood of the robot basing its pose on a particle with correct pose data.

3.6.1 Landmark association

All particles get the same observations from the landmark extraction algorithm that has processed the data from the LIDAR, the observations come in the form (distance, angle). Each particle has a list of all previously seen landmarks. To associate an observation with one of these landmarks the probability that the observation corresponds to each landmark is calculated. If the largest probability is bigger than some set value, for example, 0.001% the observation and landmark corresponding to the probability is matched. If the largest probability is lower than the setpoint a new landmark is created. To get these probabilities it is assumed that the observations are represented by a multivariate normal distribution. Each landmark that is saved has a mean position and two covariance values for distance and angle. The covariance has a starting value when a landmark is created and the mean is set as the current landmark position, both values are then corrected each time the landmark is seen again. To get the probability during observation-landmark matching Equation 3.1 is used, given the mean μ, covariance Σ and observation z calculate the probability p.

$$\mathcal{N}(p) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left\{-\frac{1}{2}(z-\mu)^T \Sigma_k^{-1} (z-\mu)\right\} \tag{3.1}$$

3.6.2 Importance weights

When landmark association has been performed each observation is mapped to a landmark and the probability calculated with Equation 3.1 is saved with the matching. The final importance weight for the particle is the product of all probabilities from the landmark-observation matchings, in the case where a new landmark was created a fixed small value is used. In the resampling step, this small value from new landmarks will favor particles that did not create any new landmarks. This is used to prevent several landmarks from being created for a single real-world landmark.

3.6.3 Resampling

When all particles have importance weights resampling is conducted to reduce the amount of "bad" particles and keep the "good" particles. The resampling works according to the pseudocode in Algorithm 3.

Algorithm 3 Resampling algorithm

Require: particles
  new_particles ← ∅
  weights ← weights of particles
  $s \leftarrow sum(weights)$
  $scaled\_weights \leftarrow weights/s$
  for particle in particles **do**
    $i \leftarrow$ select i with probability $scaled\_weights_i$
    $new\_particles \leftarrow new\_particles + particles_i$
  end for
  $particles \leftarrow new\_particles$

3.7 Pathing

During both the exploring phase and the final phase in the map-based SLAM a pathing algorithm is needed to generate frequent waypoints the robot should drive towards. Several algorithms were considered and the ones that were evaluated were A*, A*+ (A* with an additional obstacle penalty), RRT* and Phi*. The base code for both A* implementations was created using chatGPT but had to be heavily modified to work, the additional obstacle cost was written totally by the author. The RRT*[5] and Phi*[3] were taken from git.

3.7.1 A*

A* searches by expanding the node with the lowest cost. The expansion is done by assigning the nodes directly next to the current node with the same cost +1 and the diagonal neighbors add to the cost similarly but have the constant $\sqrt{2}$ instead of 1. A node always keeps the lowest cost, if it receives a lower cost than what it currently has the cost and its place in the priority queue are updated. Since the algorithm uses the best-first rule one of the optimal paths will be discovered the first time the goal is reached, when that happens the search stops.

3.7.2 A*+

Regular A* works well but it tends to path close to obstacles which sometimes results in the robot bumping into the obstacles. The boundary generated in the map is a hard limit, no path can be closer than 30 cm to an obstacle. What the "+" in A*+ has compared to regular A* is an additional cost that increases as the distance to an obstacle decreases. This is a soft limit and even if it results in a very high cost the robot can path there, this helps with regular navigation and for example, the office-chair problem described in Section 3.2.3 as the robot will try to keep additional distance to obstacles. This could also be achieved by increasing the padding when building the pathing map but that would prevent the robot from going through narrow gaps, this solution tries to keep a large distance to obstacles but if required it can path close to obstacles.

A*+ cost matrix

The obstacle cost matrix in A*+ needs to have an odd size to get a center element, the different sizes that were tried are 7x7, 9x9 and 11x11. A center element is needed since the position of the obstacle has to be in one element which should be in the middle of the matrix. Each position in the matrix is a 5 cm chunk in the real world which means that the real-world size of the matrix is between 35 cm and 55 cm. The goal of the matrix is to have positions close to the center have a high cost and further away from the center a low cost, this is to encourage the pathing algorithm to place the path far away from obstacles. This is achieved by the following method. All values in the matrix get a starting value of the square of half the size rounded up, for a 7x7 matrix this means $\left\lceil \frac{7}{2} \right\rceil^2 = 16$. All values are then subtracted with the square of the distance between the value and the center element. The final 7x7 distance matrix will look like Figure 3.6.

To get the map that is used as input to the pathing algorithm each obstacle node in the map will apply the cost matrix to its neighbors with the obstacle node itself in the center of the matrix. This works in a maximum value function and not additive. If a node is 3 nodes from one obstacle, corresponding to a cost value of $\left\lceil \frac{7}{2} \right\rceil^2 - 3^2 = 7$ and right next to another obstacle, corresponding to $\left\lceil \frac{7}{2} \right\rceil^2 - 1^2 = 15$ it will take the biggest number being 15 without any addition.

$$\begin{bmatrix} 0 & 3 & 6 & 7 & 6 & 3 & 0 \\ 3 & 8 & 11 & 12 & 11 & 8 & 3 \\ 6 & 11 & 14 & 15 & 14 & 11 & 6 \\ 7 & 12 & 15 & 16 & 15 & 12 & 7 \\ 6 & 11 & 14 & 15 & 14 & 11 & 6 \\ 3 & 8 & 11 & 12 & 11 & 8 & 3 \\ 0 & 3 & 6 & 7 & 6 & 3 & 0 \end{bmatrix}$$

Figure 3.6: Distance cost matrix with a size of 7x7.

3.7.3 Path replanning

In an early version, the replanning of the path was done as soon as the current path was untraversable which had two negative consequences. Firstly it meant that the whole path had to be validated each iteration of the SLAM algorithm which was done by iterating over all points in the path and checking if they were free in the map. This rechecking did not take considerable resources from the processor as the paths tested were short, usually no longer than 100-200 points but required some processing time. The second thing that was far more limiting was the time required to create a new path. As the robot drove it would somewhere in the distance notice that the path collided with an obstacle, this would trigger the path replanning.

A* is a deterministic pathing algorithm which means that the same map will give the same output path. If the map is only slightly changed it is highly likely that the path will also only have minor differences. What this meant for the robot was that the replanning only moved the path a small amount such that it did not collide with the new obstacle. This

worked but when the robot moved during the next SLAM iteration it would discover more of the obstacle and have to do the whole process again. In short, the robot saw an obstacle, replanned, drove for 0.5s, saw more of the obstacle, replanned again etc.

The solution was simple, instead of checking the whole path it currently only checks the first two meters. This reduced the number of points in the path that had to be validated if they were inside an obstacle but also, more importantly when the robot is within 2 meters of an obstacle it is pretty certain about its geometry. Therefore the frequent replanning problem disappeared as it would map much more of the area before replanning.

3.8 MPC controller

When a path is produced the robot has to have some way of following it. This is done using a MPC described in Section 2.2.2.

In this work two parameters affect the cost:

- **Input signals** consisting of forward velocity and angular velocity.
- **Pose difference** between the predicted robot pose after the movement and the goal point.

The pose difference cost is the square sum of the difference in x, y and θ. The points from the pathing algorithm only contain x and y, the θ value is produced by the MPC controller and represents the angle between the current point and the next point in the path. The addition of the θ value helps the robot set up the path for upcoming points and results in the robot handling some situations with sharp turns better. The maximum velocity of the robot is 200 mm/s and that value is set by the rules in the MPC controller.

3.8.1 Path smoothing

The controller drives towards a point and switches to the next point when it is relatively close, the current value used is 15 cm. With a value too small the robot might miss a point and then have to stop or reverse to get within the limit. A large value means that the controller will jump to the next point early resulting in an inaccurate path. Because the controller jumps to the next point before the current is reached the actual path will cut all corners slightly. In a square, the corners will be cut and it will path better on the straight lines. In a circle that can be seen as a continuous corner, it will drive in a circle with a slightly smaller diameter.

3.8.2 Crash detection

To prevent the controller from selecting control signals that would result in the robot crashing into an obstacle a function was implemented that checked if any part of the robot would be too close, around 5 cm from an obstacle. If this was the case that particular control signal would get a very large additional cost to prevent it from being chosen. This approach is not used currently since the A*+ pathing achieved the same results at the same time as the path was possible to follow more accurately.

Chapter 4
Results

4.1 Path result

4.1.1 Example paths

Example paths of all algorithms are presented in Figure 4.1, 4.2, 4.3 and 4.4. To minimize the risk of crashes the robot should stay far away from obstacles. As can be seen in the figures the pathing algorithm that stayed the furthest distance away from obstacles is A*+ and that is the reason it was chosen for the robot and subsequent execution time tests.

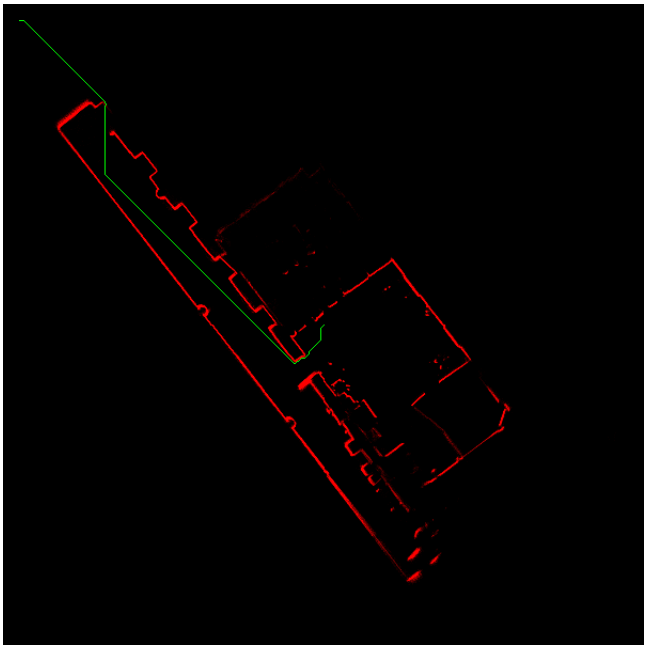


Figure 4.1: A* pathing, the path is the green line that begins in the center of the figure and ends in the top left corner.

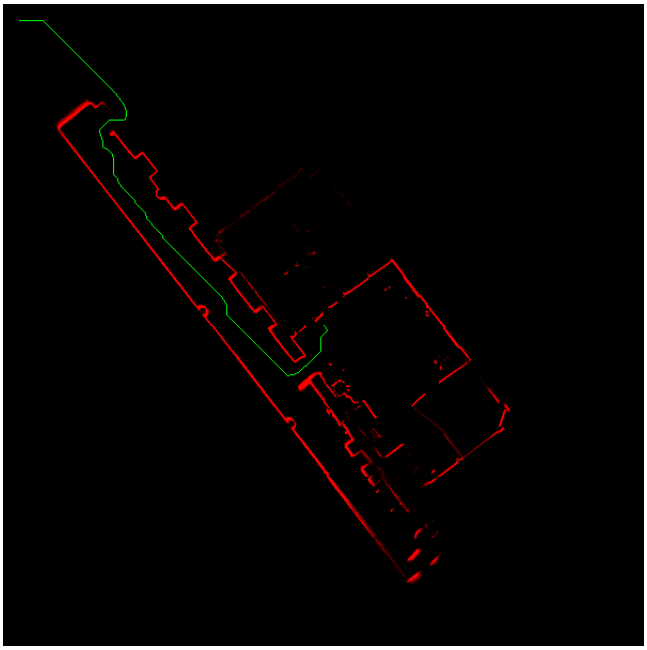


Figure 4.2: A*+ pathing with additional obstacle avoidance, the path is the green line that begins in the center of the figure and ends in the top left corner.

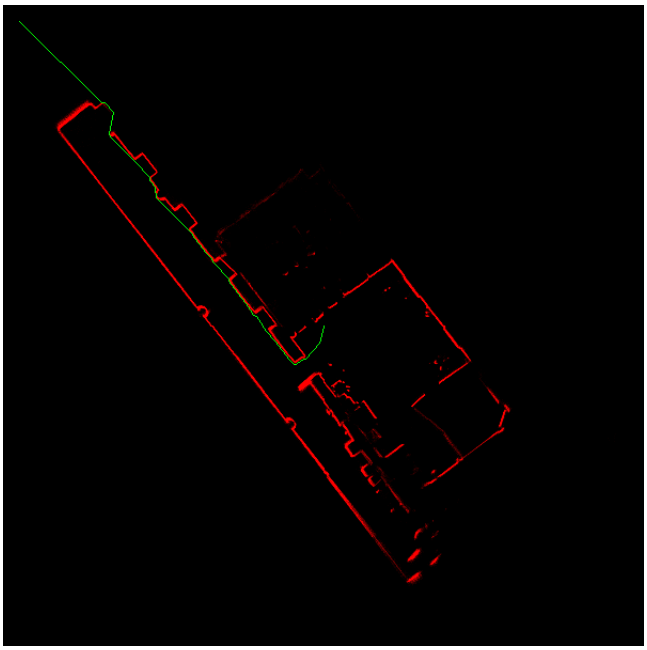


Figure 4.3: Phi* pathing, the path is the green line that begins in the center of the figure and ends in the top left corner.

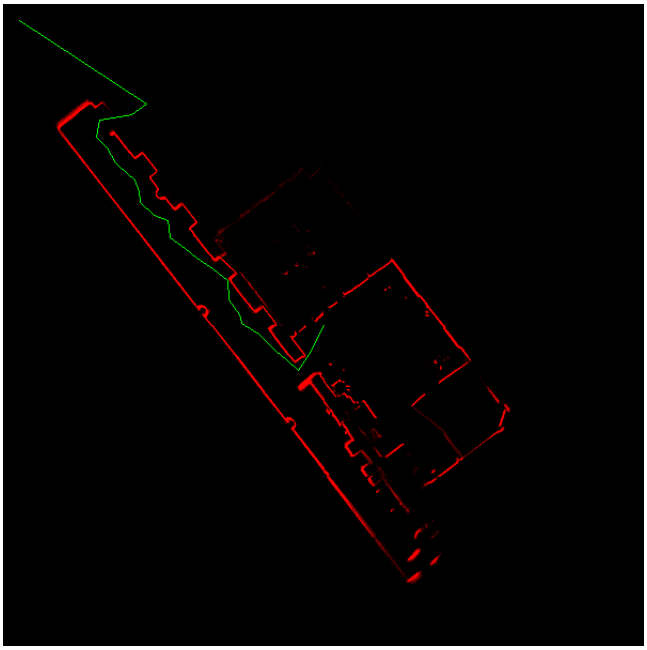


Figure 4.4: RRT* pathing, the path is the green line that begins in the center of the figure and ends in the top left corner.

4.1.2 A*+

To test the execution time of A*+ it was used to create paths between different points in an already built map. To ensure stability and prevent the temperature of the processor to affect the results all tests were done with the CPU of the Raspberry Pi locked at 1000MHz, the regular operating frequency is 1500MHz. Another thing that could affect the results is the CPU scheduler pausing the execution of the algorithm to run something else. Since the processor has 4 cores and only one was used to run the algorithm there should be other lesser-used cores that the scheduler could place other processes on. For each set of two points the algorithm was run ten times and all times were saved, the results from this are presented in Table 4.1. The map used was one of the maps built during the SLAM algorithm test, shown in Figure 4.5. The points the pathing algorithm used is marked with a white dot in a green circle. From the top left to the bottom right the following positions are marked on the map: (200,200), (300,300), (320, 340) and (350, 400). One of the points, (0,0) is outside the visible figure and is used as an unreachable path since the perimeter is closed. This is used as a worst-case situation for this specific map.

The additional obstacle avoidance helps greatly with the pathing of the robot but the map with applied cost matrices is re-computed each time the pathing algorithm runs. The runtime required to create this map for two different environments is displayed in Table 4.2. The original map for the two environments are shown in Figure 4.8 and 4.9.

As with the previous test the Raspberry Pi was running at 1000MHz and the test was done ten times for each map. The reason why the larger environment with more obstacle points requires more time to create the map is that for each obstacle point the cost matrix in Figure 3.6 has to be applied. More points results in the cost matrix having to be applied more times. The large map has around 6 times more points compared to the small map and runs 5 times slower. The expectation was that the increase in time should be linear compared to the number of obstacle points and this is almost the case for these two samples.

From	To	Mean time	Max time	Min time
(350, 400)	(200, 200)	4.37	4.43	4.34
(300, 300)	(200, 200)	4.13	4.20	4.11
(300, 300)	(320, 340)	1.72	1.72	1.70
(300, 300)	(0, 0)	4.72	4.82	4.69

Table 4.1: Times in seconds required to run A*+ between two points.

Number of obstacle points	Mean time	Max time	Min time
6484	0.33	0.34	0.33
39847	1.70	1.72	1.69

Table 4.2: Times in seconds required to calculate the distance cost matrix used in A*+.

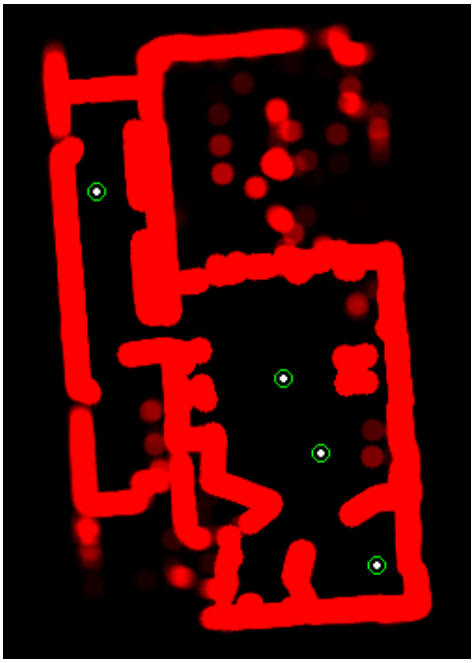


Figure 4.5: Map used for the pathing algorithm test, the white dots in green circles are the different positions the algorithm was run through.

4.2 Positional accuracy

To test the positional accuracy over time and the ability to follow a path the following test was conducted. The robot drove 10 laps in a circle with a diameter of 2 meters, in total the robot traveled around 63 meters. To introduce some controlled inaccuracies a flat cable was taped to the ground on the opposite side to the start/finish line, the cable height was 3.5 mm and width around 5.5 mm. Each time the robot passed the start it would stop for 5 seconds and the (x,y) error was recorded by taking a picture from above, an example image is presented in Figure 4.6.

Figure 4.6: Example picture used to analyze the positional error.

Map-based SLAM

The results for map-based SLAM are presented in Table 4.3.

Lap	X error	Y error
1	51.7	36.6
2	4.3	35.9
3	-40.2	43.1
4	33.0	43.8
5	-19.4	48.1
6	-34.4	43.1
7	33.0	45.9
8	-21.5	35.9
9	45.2	50.9
10	36.6	44.5

Table 4.3: Positional error in mm when following a predetermined path using map-based SLAM.

SimpleSLAM

The results for SimpleSLAM are presented in Table 4.4.

Lap	X error	Y error
1	-19.7	56.6
2	-4.3	48.0
3	-63.4	50.9
4	-33.1	54.7
5	-26.9	54.7
6	-29.3	53.8
7	-44.6	46.1
8	-40.8	37.9
9	-33.1	62.4
10	-34.6	52.8

Table 4.4: Positional error in mm when following a predetermined path using SimpleSLAM.

FastSLAM 2

The results for FastSLAM 2 are presented in Table 4.5.

Lap	X error	Y error
1	-71.9	77.0
2	-41.8	63.8
3	-34.2	47.4
4	-9.2	44.9
5	-54.6	50.5
6	-34.7	57.1
7	-48.5	44.4
8	-32.1	45.4
9	-6.1	38.3
10	-39.8	31.8

Table 4.5: Positional error in mm when following a predetermined path using FastSLAM 2.

Pure odometry

The SLAM algorithm is used to combat the inaccuracies from the odometry. In Table 4.6 the results from the pathing accuracy test using only odometry are presented. It does not have 10 rows since the robot position ended outside the measuring area, in lap 4 parts of the robot were outside the area but enough was visible to determine a position.

Lap	X error	Y error
1	-140.1	89.3
2	-131.7	92.4
3	-155.5	93.9
4	-304.9	241.8

Table 4.6: Positional error in mm when following a predetermined path using only odometry.

4.2.1 Method comparison

In Table 4.7 results from all methods in the path accuracy tests are shown, if the reference using only odometry had run all ten laps the result for that method would probably be much worse.

One common thing among the SLAM algorithms is a relatively large y error that always takes a positive value. This is probably a result of the MPC controller, as discussed in Section 3.8.1, the controller will cut corners. The pure odometry has a large error in lap 1 then similar results in laps 1-3 and then again a large error increase during lap 4. This behavior is expected

as the odometry can be very accurate if the robot isn't disturbed, the first time it passed the cable the wheels probably slipped causing the error in lap 1, the following two laps had a very similar result likely driving over the cable with very little wheel slip. During the last recorded lap, the wheels probably slipped again causing the increased error between lap 3 and 4.

Method	Mean X-error	Largest X-error	Mean Y-error	Largest Y-error
Odometry	-183.1	-304.9	129.4	241.8
Scan matching	8.8	51.7	42.8	50.9
SimpleSLAM	-33.0	-63.4	51.8	62.4
FastSLAM 2	-37.3	-71.9	50.1	77.0

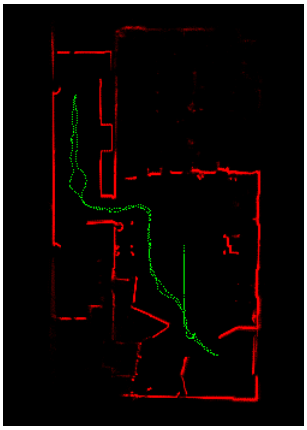
Table 4.7: Comparison of different SLAM methods, units in mm.

4.3 Map creation in a static environment

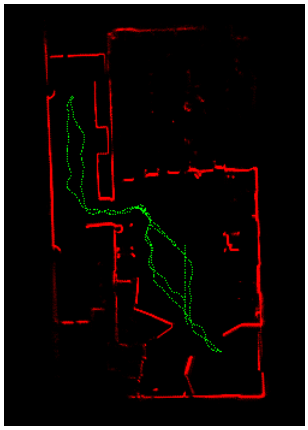
To test the pathing and map generation the robot explored the same environment 20 times. It returned to the start position when it had created the map to such an extent that the robot could not exit the perimeter. It started in the same pose for each run and the environment stayed the same during the whole test. The robot pose combined with a timestamp was saved for each loop of the SLAM algorithm which ran at 2Hz and the final map was saved at the end of the run. An exception to the 2Hz frequency was during the re-planning of the path once a path turned out to be blocked, during re-planning the robot would stop and wait until the next path was done. The results from the test are presented in Table 4.8. The path count and path delay include the initial path as well as the final path back to the start position, the number of re-plannings is therefore two less than the path count. The total time was measured as the time between the first loop of the SLAM algorithm until the robot had completed the map and was at the start position, this time includes both the driving time and path re-planning time. The distance is the number of meters traveled during the whole run. During all 20 runs, the robot had a path very similar to one of five different alternatives that are shown in Figure 4.7

Method	Path count	Pathing delay	Total time	Distance	Avg speed
Mean	6.05	15.32s	224.80s	38.505m	0.184m/s
Max	7	19.08s	263.13s	46.026m	0.192m/s
Min	5	13.18s	201.09s	34.382m	0.162m/s

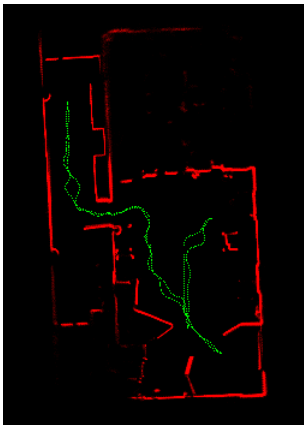
Table 4.8: Results from the 20 runs of the map-based SLAM algorithm.



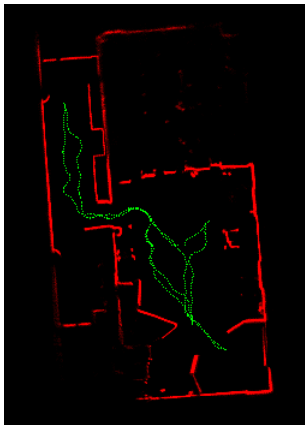
12 runs had a path similar to this.



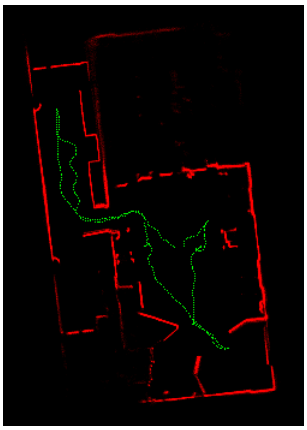
4 runs had a path similar to this.



1 run had this path.



1 run had this path.



2 runs had a path similar to this.

Figure 4.7: The five different paths that occurred during the scan matching SLAM test. The path taken is the green dots.

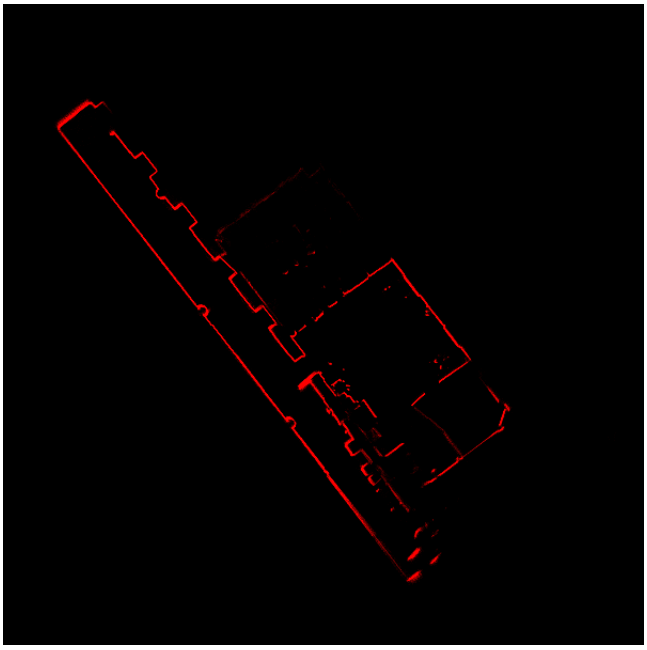


Figure 4.8: Example map only used for visual representation, each pixel is a 5x5 cm chunk and brighter red means that chunk has been seen more times.

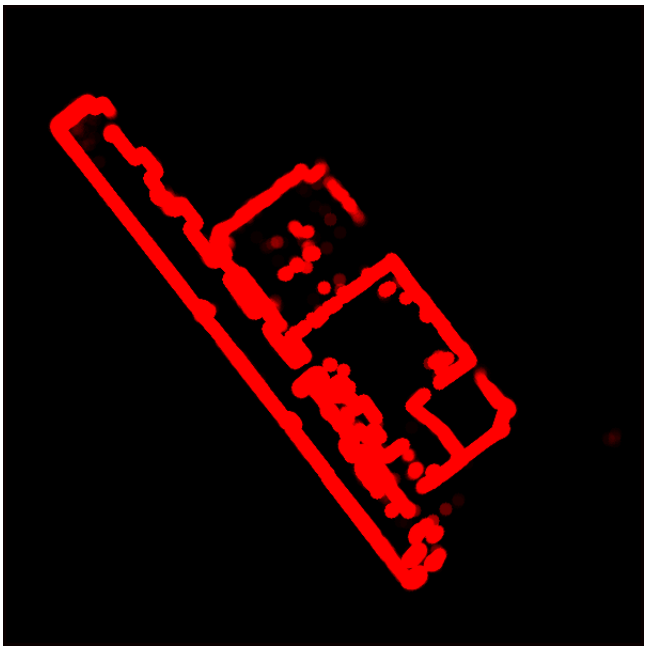


Figure 4.9: Example map used by the pathing algorithm, each pixel is a 5x5 cm chunk and brighter red means that chunk has been seen more times.

4.4 Accuracy of the generated map

To test the accuracy of the generated map the robot explored a larger part of the environment, the generated map is shown in Figure 4.8. The map used for the pathing algorithm with padded obstacles from the same run is shown in figure 4.9. Some distances in the generated map were analyzed and compared to the real world, the distances measured are shown in Figure 4.10 and the results are presented in Table 4.9. The distances in the generated map are close to the real-world measurement.

	D0	D1	D2	D3	D4
Generated map	26.61	1.85	1.85	5.82	9.30
Real word	26.65	1.98	1.97	5.95	9.39

Table 4.9: Map distances in meters compared to the real world.

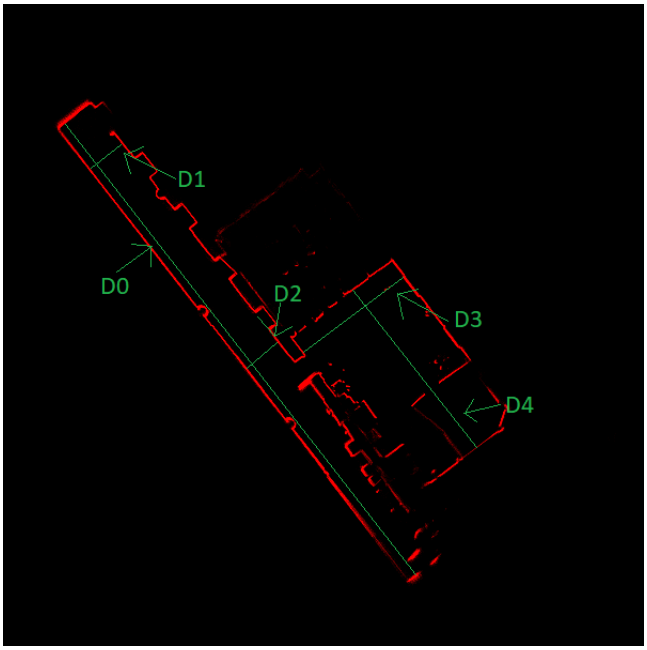


Figure 4.10: Distances compared to the real world.

4.5 CPU requirements

To test the processor requirements of the SLAM algorithms the robot was started as usual but the execution times were recorded during 50 seconds or 100 loops of the algorithm. The measurements started when the robot had been running for 10 seconds, the results are presented in Table 4.10 and all values are in ms. The results for the landmark-based methods, FastSLAM 2 and SimpleSLAM are normalized to only one landmark observation, the program has linear time complexity regarding the number of landmarks and the results would

vary a lot if the results were not normalized. The number of landmarks varied between two and four, if four landmarks were observed and it took 0.2s the result was divided by four which gives 0.05s.

The other processor-intensive parts are the LIDAR data processing and MPC, results from these are presented in Table 4.11. Notice that the "Read LIDAR data" method is in %, this is the code that reads the data from the LIDAR and sends the data in chunks to the rest of the program. The reason the result is in % is that the LIDAR runs as a generator yielding a reading once it is available and blocking the rest of the time. Starting and stopping the timer for each of the 6500 values being generated each second resulted in strange results and thus the processor utilization was used instead. This should work almost equally well since the LIDAR function runs in its own Python process and the CPU utilization is only reading from that individual process.

The results presented in Table 4.10 and 4.11 are not the only code that is being run but it is a clear majority as only small auxiliary tasks are being run outside these functions.

Algorithm	Mean	Min	Max	Std dev
Scan matching	53.1	32.7	84.6	14.9
FastSLAM 2	53.4	47.8	76.1	6.1
SimpleSLAM	3.8	3.0	8.3	0.7

Table 4.10: Execution times in ms of different SLAM algorithms.

Function	Mean	Min	Max	Std dev
MPC	65ms	8ms	104ms	16ms
Corner extraction	24ms	20ms	36ms	3ms
Read LIDAR data	74.2%	66.7%	80.0%	4.15%

Table 4.11: Execution time in ms or processor requirements for data processing.

Chapter 5
Summary

5.1 Limitations

In the content of this work, some areas were not thoroughly explored and some pre-existing solutions were used.

Loop closure

The map generation can handle inaccuracies in the believed pose of the robot when generating the map, this works as long as an area that is correctly mapped is visited relatively frequently. What the robot currently can't do is known as loop closure. If it was to drive in a big loop, for example, a circle with a large diameter the small errors would add up. When the robot returned to the start the error would be too large to create a cohesive map. Finding when a loop closure should be performed and then correcting the whole map is very difficult and not done in this work, an example implementation is shown in [10].

Obstacle removal

The map-based SLAM method implemented does not have the feature to remove obstacles once they have been observed, this is because it is assumed the environment is static during mapping. Using ray casting obstacle removal could be done, this would however require a lot of processing power. This would make it possible to handle dynamic obstacles and remove erroneous observations, usually caused by reflective surfaces.

Incomplete maps

During the self-exploring stage, the robot will continue mapping until it can't path out to infinity. In some situations, this behavior will result in an incomplete map, for example, if the room has semi-closed spaces inside the perimeter. An example of this is shown in Figure

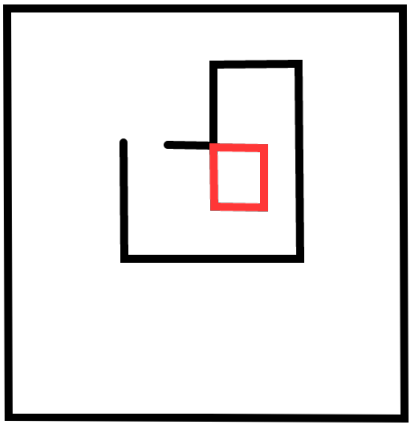


Figure 5.1: Example room geometry where the lines in red would not get mapped.

5.1, here the robot would map the whole exterior rectangle and on the way probably map the inner black lines as well. It would not on its own drive into the inner shape and thus the lines in red would not be mapped. It is possible to run the mapping functionality when the robot enters the drive-to-point stage. If the robot would try to go to a point in the inner shape the lines in red would be correctly mapped and no problem should occur.

Map data struture

The use of a matrix with a fixed size as a map is not ideal for an exploring robot where the size of the map is unknown. An effort was made to make the map much more efficient. In this implementation, the map was divided into submaps of 1x1 m, each containing 400 5x5 cm chunks requiring 3.2kB of memory for one submap. All submaps got a unique key of two values (x,y) where the coordinate system was aligned with the robot pose but scaled differently. The robot would start in the middle of submap (0,0) and as an example, moving forward one meter would result in it being in the middle of submap (1,0). The unique key of the submap was used in a hashmap to efficiently find the submaps. This system worked and a pre-recorded map could be converted into this much more space-efficient map and then back to the original form. The reason this was not used was the additional complexity it brought to many parts of the program especially the pathing and scan-matching. If the robot would operate in a bigger space where memory constraints would pose a problem this could be a good solution.

Pathing and replanning

Currently, A*+ is used instead of Phi* which is similar but optimized for replanning. The open-source Phi* implementation ran slowly compared to the A*+ implementation and it did not have the required obstacle cost matrix functionality that prevented crashes. Due to the time requirements of optimizing and implementing the obstacle cost matrix for Phi* a decision was made to use A*+ since it ran fast enough. Phi* with the obstacle cost matrix would produce similar results and require less time during replanning.

Pose accuracy recording

When analyzing the positional error a picture was taken from above the robot, the camera was placed relatively high, around 2.5 meters above the robot. The placement of the camera gave a relatively good view of the robot but the actual position where the wheels meet the ground could not be seen. The results are therefore not perfect but the accuracy should be within 1-2 cm.

LIDAR test setup errors

The test setup for the LIDAR accuracy was not perfect and did have an impact on the result. Firstly the 5 meter distance was measured with another laser distance meter, Bosch DLE50 Pro which has an accuracy of +-3 mm. Secondly, the materials that were measured had a width of around 50 cm where only the middle of the material was 5 meters from the LIDAR. This means that the edges had a longer distance to the LIDAR, Pythagoras theorem gives $\sqrt{5000^2 + 250^2} = 5006$. This difference between the center and edges of the material raised the mean and standard deviation.

5.2 Conclusions

RQ1: Is it possible to run a SLAM algorithm implemented in Python with the limited computing power a single board computer like a Raspberry Pi has?

Several different SLAM algorithms were implemented and tested successfully. The common thing among these is that LIDAR data processing requires the most CPU time, as this is easy to run in a separate process it poses no limitation on the program. The rest of the program required a total of around 0.1 to 0.4 seconds on one core depending on SLAM method and the number of observed landmarks, this makes it possible to run at the wanted frequency of 2Hz while leaving two out of four cores unused. If the LIDAR data could be handled in a more efficient way the program would only require one of the four cores on the RPI. Being able to run the program on a small and cheap single-board computer that requires very little space and has a low power consumption is important. One of the reasons for this is that it makes it possible to deploy to many different platforms while not requiring large batteries.

RQ2: How accurate positioning can be achieved?

The positional accuracy was in the range of 5-10 cm for both the map-based and landmark-based methods. The error is due to three things:

- **Map resolution.** The discretized grid map will hide some details in the data from the LIDAR, a measurement in a corner of a 5x5 cm chunk will result in a point $5 * \sqrt{2}$ cm away in the same chunk also being treated as occupied space. This means that the free spaces will be slightly smaller in the generated map compared to the real world. The effect of this can be reduced by increasing the resolution of the map. The landmark-based methods do not use a discretized map, therefore this does not affect those methods.
- **SLAM algorithm inaccuracies.** The SLAM algorithms are not perfect, they use probabilities and cost functions to decide how landmarks should be matched and how the

point clouds should be transformed, these methods are not 100% accurate and will in most cases result in some errors.

- **MPC inaccuracies.** The MPC selects the next point before the current is reached, this will produce some positional inaccuracies as discussed in Section 3.8.1.

Given all of these causes of error, the results are good. If higher accuracy is needed the maximum speed of the robot can be reduced from the current 0.2 m/s to allow for tighter tolerances in the MPC. It would also be possible to use a higher-resolution map, increasing memory requirements. The robot has a width of 60 cm and most doors and tight passageways are wider than the robot width + positional error. Therefore, no reason was found to make the robot run slower or use more memory to get better positional accuracy.

RQ3: How accurate is the generated map?

The map generated by the scan matching algorithm produces distances around 10 cm too short between points, mostly due to the resolution of the mapping as discussed in RQ2. The reason the distances are more than $5 * \sqrt{2}$ cm less than the real world is due to them being measured between two surfaces, if each surface gives an error the error between two surfaces will be twice as large. The map resolution dependent error should therefore be less than $2 * 5 * \sqrt{2}$ cm. The error will also be affected by the accuracy of the SLAM algorithm as the believed pose is used when creating the map. The error from the MPC will not affect the accuracy of the generated map, this is because the SLAM algorithm uses the pose estimate and not the position of the current MPC point. The important thing regarding the map accuracy is that the accuracy does not depend on the distance between points on the map but only the constant ~10 cm. This makes the mapping behave predictable over long distances which is important when planning long paths.

5.3 Future work

The currently used pathing algorithm is A*+, it has an evolution called Phi* that is optimized for replanning the path, this would be better to use since the robot would require less stationary time when replanning the path. It would however require the same obstacle avoidance addition as the A*+ implementation has.

The LIDAR uses an unnecessary amount of processing power, instead of reading all 6500 values per second it would be better if only the data needed by the SLAM algorithm was processed. This would reduce the number of values from 6500 to 1300 per second if the SLAM algorithm was to run at 2Hz.

References

[1] Lidar triangulation image. `www.movimed.com/knowledgebase/what-is-laser-triangulation/`. Accessed: 23.05.2023.

[2] Odrive motor controller. `www.odriverobotics.com`. Accessed: 26.04.2023.

[3] Phi* python code. `github.com/rhidra/phi_star_2d`. Accessed: 01.05.2023.

[4] Ros framework. `www.ros.org/`. Accessed: 26.04.2023.

[5] Rrt* python code. `github.com/rland93/rrtplanner`. Accessed: 01.05.2023.

[6] Scipy minimizing algorithm. `docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#rdd2e1855725e-12`. Accessed: 16.05.2023.

[7] Syed Zeeshan Ahmed, Vincensius Billy Saputra, Saurab Verma, Kun Zhang, and Albertus Hendrawan Adiwahono. Sparse-3d lidar outdoor map-based autonomous vehicle localization. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1614–1619, 2019.

[8] Colin L. Freeman, John H. Harding, David Quigley, and P. Mark Rodger. Structural control of crystal nuclei by an eggshell protein. *Angewandte Chemie International Edition*, 49(30):5135–5137, 2010.

[9] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[10] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[11] Sertac Karaman, Matthew R Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt. In *2011 IEEE international conference on robotics and automation*, pages 1478–1483. IEEE, 2011.

[12] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. *Aaai/iaai*, 593598, 2002.

[13] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *IJCAI*, volume 3, pages 1151–1156, 2003.

[14] Philippe Moutarlier and Raja Chatila. *An experimental system for incremental environment modelling by an autonomous mobile robot.*, volume 139 of *Lecture Notes in Control and Information Sciences. 139*. Springer Berlin Heidelberg, 1990.

[15] Alex Nash, Sven Koenig, and Maxim Likhachev. Incremental phi*: Incremental any-angle path planning on grids. 2009.

[16] Alexandr Stefek, Thuan Van Pham, Vaclav Krivanek, and Khac Lam Pham. Energy comparison of controllers used for a differential drive wheeled mobile robot. *IEEE Access*, 8:170915–170927, 2020.

[17] Mingze Xu, Bo Liu, Jun Zhou, Zelin Zhou, and Zhenjie Weng. Trajectory tracking of wheeled soccer robots based on model predictive control. In *2021 6th International Conference on Control and Robotics Engineering (ICCRE)*, pages 50–54, 2021.

[18] Shao-Wen Yang and Chieh-Chih Wang. On solving mirror reflection in lidar sensing. *IEEE/ASME Transactions on Mechatronics*, 16(2):255–265, 2011.

[19] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.