



# Universiteit Leiden

## Opleiding Informatica

dynSLAM: Robust 2D Lidar SLAM  
in Dynamic Environments

Name:	Elgar R. van der Zande
Studentnr:	s1485873
Date:	August 16, 2022
1st supervisor:	Dr. Erwin M. Bakker
2nd supervisor:	Prof. dr. Michael S. K. Lew

MASTER THESIS

Leiden Institute of Advanced Computer Science (LIACS)  
Leiden University  
Niels Bohrweg 1  
2333 CA Leiden  
The Netherlands

## Abstract

In this paper, dynSLAM is presented. dynSLAM is a SLAM solution for dynamic environments. The SLAM algorithm is grid map based and uses smaller localized maps. These local maps are placed into an overreaching large scale history map. Because the history cells only contain the most recent local map, dynSLAM can generate up-to-date global maps, whereas existing techniques do not remove depreciated information. To enforce a robust mapping experience, loop closure and numerous data conditioning techniques are utilized. Such as, egomotion correction, rejection of far away lidar points, and removal of unrealistic points from the grid map.

The removal of unrealistic points is a novel grid map filter, that aims to remove erroneous points in unobservable areas. The filter first calculates an observable area, then it removes cells that fall outside of this area. This step is typically omitted in traditional SLAM solutions. The end result is that our solution leaves much less artifacts behind walls and objects.

Moreover, dynSLAM is able to plan routes to a destination, set on a mapped area. A mobile robot using dynSLAM, can drive the route autonomously, while avoiding dynamic obstacles. The destination can be set either manually or by the dynSLAM, when on so called exploration mode. The exploration algorithm, attempts to choose destinations in such a manner that the environment is fully explored.

dynSLAMs software uses a client-server model, the server is running on a robot where as the client is running on a computer. The remote computer allows the robot to be controlled remotely. The software is open source and made public by the means of a GitLab repository.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Related Work</b>	<b>6</b>
<b>3</b>	<b>SLAM System Overview</b>	<b>8</b>
3.1	Server . . . . .	10
<b>4</b>	<b>dynSLAM</b>	<b>12</b>
4.1	dynSLAM Components . . . . .	15
4.1.1	SLAM - Preprocessing Filters . . . . .	16
4.1.2	SLAM - Scan Matching . . . . .	19
4.1.3	SLAM - flood fill cleanup . . . . .	21
4.1.4	SLAM - Grid-maps and Loop Closure . . . . .	23
4.1.5	SLAM - Global Map Updates . . . . .	24
4.1.6	Path Finding . . . . .	26
4.1.7	Path Following . . . . .	27
4.1.8	Exploration . . . . .	29
<b>5</b>	<b>Experiments</b>	<b>31</b>
5.1	Normal Driving . . . . .	31
5.1.1	Hallway Dataset . . . . .	31
5.1.2	Livingroom . . . . .	33
5.2	Map Building . . . . .	35
5.2.1	Small Increments . . . . .	35
5.2.2	Loop Closure and Cleanup-pass . . . . .	37
5.3	Distance Accuracy . . . . .	38
5.4	Dynamic Environment . . . . .	40
5.4.1	Walking in the Vicinity of the Robot . . . . .	40
5.4.2	Opening and Closing a Door . . . . .	42
5.5	Route Planning . . . . .	42
5.6	Frontier Finding . . . . .	45
<b>6</b>	<b>Future Improvements</b>	<b>48</b>
6.1	Local Sub-map Storage . . . . .	48
6.2	Ray Length Filter . . . . .	49
6.3	Networking . . . . .	49
6.4	3D Egomotion and Height Correction . . . . .	49
6.5	AI Driver Improvements . . . . .	50

<b>7</b>	<b>Conclusion</b>	<b>52</b>
<b>A</b>	<b>Code Layout</b>	<b>54</b>
A.1	Server . . . . .	54
A.2	Client . . . . .	54
A.2.1	General . . . . .	55
A.2.2	UI . . . . .	55
A.2.3	SLAM . . . . .	55
A.2.4	Auto Piloting and Exploration . . . . .	56
A.3	SLAM Library . . . . .	56
A.4	Networking Protocol . . . . .	58
<b>B</b>	<b>Controls and Indicators</b>	<b>60</b>
B.1	Controls . . . . .	60
B.2	Indicators . . . . .	61



# Chapter 1

## Introduction

SLAM (Simultaneous Localization and Mapping) is a highly researched topic in the robotics community and with good reason. The applications are virtually endless and range from indoor mapping to space exploration. This paper is limited to indoor SLAM and exploration.

The goal of SLAM is to use a mobile robot to create a map of the environment it is currently driving. The initially unknown map is filled during traversal of the environment. The environment is observed using sensors. Current research focuses almost exclusively on visual SLAM (i.e., a camera as a sensor) or on LIDAR SLAM. In this paper, we focus on LIDAR SLAM.

A LIDAR (Light Detection And Ranging) device is a sensor that emits laser pulses. These laser pulses reflect on an object and then shine on a detector in the LIDAR. The time of flight of the pulse is used to determine the distance between the sensor and the object. By emitting multiple rays in many direction or by rotating the LIDAR, a 2D or 3D image of the environment can be obtained.

Observations of from the sensor are then incorporated into a map. A popular way to store the map is to create a grid map. A grid map is similar to a top view image of the environment. The cells (or pixels) of the grid map indicate whether a location is occupied, empty or unobserved. SLAM systems that store their environment in a grid map, are called grid-map-based SLAM systems.

While it may seem trivial to simply match a new observation to the already existing map, in practice many problems arise. Most notably, errors can accumulate and distort large scale structures. This is most notably when a place is revisited after some time. Because of accumulated error, between the two visits at this location, walls and objects no long align between the first and last observation. A solution to this problem is to use loop closure in combination with GraphSLAM.

GraphSLAM is a different way to store the environment in the robot. Instead of merging all the observations into a single grid map, each observation is stored in a node of a graph. The edges of the graph contain the transformation needed to place one node on top of the other. By scan matching consecutive observations, edges can be added to the graph. The main advantage of GraphSLAM is that the location of an observation can change during the SLAM progress (cf. grid map SLAM where the location is fixed after each scan matching).

Additionally, a loop closure scan matcher searches for overlapping observations and places additional constraints (edges) in the graph. The location of the nodes (observations) is then optimized. Intuitively, the optimization process can be thought of as a network of springs, where the springs are the edges of the graph and the nodes are the locations of the observations. The energy favourable position of the system is analogous to the optimal position of the nodes

(observations) [22].

By introducing loop closure, the problem of error accumulation can be mostly removed from a SLAM system. However, loop closing introduces a new problem in the form of wrong loop closure constraints. Wrong loop closure constraints are edges added to the graph that are incorrectly generated because similar areas are added observed and matched. However, in reality the locations are not the same. Without proper handling of wrong loop closure constrains the SLAM map is ruined. Dealing with wrong loop closures is still an active area in SLAM research.

Another open (and seemingly often ignored) issue in SLAM is mapping of dynamic environments. In these kinds of environments, small changes are allowed to happen during the SLAM mapping progress. Typical changes indoors are movement of people and furniture, opening and closing doors etcetera. Existing implementations that do support dynamic SLAM, achieve this by changing the way they update the cell values in a grid map. Dynamic SLAM requires the ability to remove old values when changes occur.

Our dynSLAM is a grid-map-based SLAM system. The SLAM solution is made globally consistent, by implementing loop closure. Moreover, it is designed to operate in dynamic environments and can quickly update its map when changes occur. Our method of updating the grid map is to similar to existing dynamic SLAM solutions, in the sense that we also remove hit and misses from the grid-map. However, we also introduce the novel concept of a history map. The history map is a map of smaller sub maps. Using this map, it is possible to make a selection of only the most recent sub maps. Additionally, by using the history map, the SLAM system can run continuously without filling up because old measurements can be discarded.

Another objective of dynSLAM is that it should be able to correct measurement errors. These errors occur in situations where measurements are distorted due to sudden accelerations of the LIDAR or motion sensor. Another source of error is simply the measurement noise of the LIDAR. This phenomenon results in thick walls because points are measured before and after the actual wall. Misplaced points in a grid map can result in bad scan matches because the scan matcher is given too much freedom when matching a new scan.

Transitional solutions to this problem vary widely. Some SLAM implementations do not use any additional data conditioning. Smoothing the point cloud coming from a LIDAR is a popular approach. Alternatively, not using all data available is also possible. For example, sometimes it is beneficial to only add LIDAR point clouds once the robot has moved a significant distance.

In our dynSLAM approach, many of the measurement errors are corrected by the same systems that allow the incorporation of dynamic environments. Because measurement errors and changes in the environment can be handled in similar fashion. Additionally, we use a data conditioning pipeline that takes the incoming LIDAR data and attempt to correct or remove inconsistent data. The pipeline consist of a filter that removes far away points, does egomotion correction and detects LIDAR observations that are distorted due to large pitch and roll deflections. The lasts to steps in the pipeline are done with the aid of an IMU device.

Moreover, we also designed a novel filter that works on a grid map. The filters purpose is to remove misplaced points on a grid map. It does this by first determining the reachable area. Using this area, it can determine which points should not have been placed. These points are then removed from the map. The resulting maps shows thin lines around walls and objects. Not only does this look better, it also ensures better performance during scan matching.

Pathfinding is another contribution of our work. The pathfinding problem is not directly related to SLAM. However, mobile robot path finding has some additional difficulties compared to normal (graph) based pathfinding, like Dijkstra or A\*. One obvious problem is that the mobile robot has a certain width, and as a result it can not fit trough every opening. Moreover, the SLAM created 2D map can contain small errors, which means that the robot can not fit trough an opening in reality, even though it should be possible according to the map. Driving the robot

though a narrow space is complicated by the fact that the robot controls are not precise. Hence, the algorithm generating the path needs to take these factors into account.

The dynSLAM pathfinder uses a modified A\* pathfinder. The modifications are made to allow some room for error, made during driving. The modifications are made such that there is clearance around the generated path. The path is generated from the global grid map. The adapted A\* algorithm constructs a path of neighboring cells. This path is then converted into a set of waypoints, these way points are connected with straight lines. Having only a few waypoints compared to the dense path generated by the adapted A\* algorithm is beneficial when driving the path, this is because the angle between the robot heading and the waypoint is better defined when the separation distance is large.

Additionally, we show how to make our robot anonymously follow this path without collisions. Collisions are avoided by looking for objects in the vicinity of the robot, using the most recent LIDAR data. A set of simple steering rules determines the action the robot should take to avoid a collision.

Lastly, we touch upon anonymous (frontier) exploration and automatically keeping the map up-to-date. Frontier exploration is a typical exploration techniques encountered in many papers on the subject. Additionally, instead of stopping after all frontiers are visited, we revisit places based on their last observation. The oldest places are revisited first. This has the effect that it keeps the map up-to-date.

The dynSLAM project is mostly built up from scratch using C++. The robot hosts a server to which a remote client can connect. This allows for: remote control of the robot, instructing it to drive to some location or explore the environment. A video feed is available for the human controller, hence the robot can be fully controlled without a visual on the robot. The source code for the project can be found in the following GitLab repository <https://gitlab.com/masterthesis14/dynSLAM>.

Scientific contributions of the projects are:

- a grid-map SLAM library.
- a client and server applications for remotely controlling a robot platform.
- a method for correcting errors and incorporating dynamics in the environment. This works by lowering hit and miss cell values on conflicting measurements.
- a, to our knowledge, novel method for removing misplaced cells on grid maps. This works by determining the reachable area of the robot, and then deciding which measurements are possible.
- a, to our knowledge, novel method for storing local maps in an overreaching history map.
- a method for doing SLAM in dynamic environments.
- a method to construct routes on a SLAM grid map
- a method to have a robot anonymously follow a route
- a method for anonymous exploration to find unvisited locations and update the map.

## Chapter 2

# Related Work

Traditionally, SLAM used an (Extended) Kalman Filter to do pose estimation [5]. A plethora of similar filter variants has also been applied in SLAM [44]. The most frequently encountered variants are information-based filters and unscented Kalman filters [18]. Later methods move away from the Kalman filter-based approach, in favor of grid-maps [3] and GraphSLAM [36]. In our work, we utilize grid-map SLAM. Additionally, GraphSLAM is used as a back-end for large-scale data associations.

Scan matching or data association between scans can be achieved by many different means. Direct landmark extraction had been done in the past in the form of tree detectors [4]. Other methods feature ‘feature detectors’, varying from simple wall and corner detection [17] to context invariant feature detectors [24].

Later SLAM methods do not require finding and associating landmarks. Instead, determining a pose difference between scans is used to reconstruct the agent’s path. Pose transformations can be obtained by matching two or more scans. Scan to scan matching is proposed by [27]. Grid-map matching incorporates multiple past scans into a map and uses this map to align newly obtained scans to [18]. Our solution uses this scan to map approach because of its robustness and speed. Our work also uses grid map scan matching. Our method differs in the way we construct the grid map, our grid map update rules are adapted to work in dynamic environments, hence they can also make correctional changes to the grid map.

When it comes to path planning for mobile robotics, there are four [31] main categories. These are Reactive Computing; Soft Computing; Optimal Control and C-Space search. Reactive computing is a method where a global path or direction that does not incorporate obstacles, is followed. Any obstacles are avoided using reactive manoeuvre [20, 15, 7, 38, 34, 14, 8] or a local optimization [29, 19, 30].

The methods that fall under the Soft Computing class, do not strive to find exact/deterministic solutions. Commonly, they are based on fuzzy logic or biological processes. Genetic algorithm [16, 37, 2, 23] can be used to find a path. Genes representing waypoints are mutated until convergence. Another possible method is to use a Swarm Optimizer [28, 12, 9, 10, 32]. These algorithms are mostly based on the collective behavior of animals. For example, the Ant Colonies Optimizer leaves a pheromone trail to mediate communication between particles. Finally, Fuzzy logic [33, 42, 41] and Machine learning [43, 6] also fall under the Soft Computing category.

Optimal Control divides into two sub-categories. One method is to optimize a path through a smooth cost map, by means of numerical PDE solving [39, 25]. The other sub-category is global optimization. Here, an already existing path, found by other means, is globally optimized [21].

The advantage of OC algorithms is that they find smoother and possibly shorter paths [11], compared to more traditional pathfinding because waypoints are not placed on discrete (i.e., cells, pixels) points.

C-Space search consists of the classical graph-based shortest path algorithms like Dijkstra, A\*, and D\*. Any-Angle [13] algorithms aim to solve the angle restriction caused by neighboring cells (i.e., increments of  $\pm 45$  degrees). Another type of algorithm belonging to this category is the sample-based algorithm. For example Rapid Random Tree and Rapid Deterministic Tree.

Our path-finding method fits into the Any-Angle C-space search algorithms. We use A\* to find the path and then prune unnecessary waypoints. Moreover, we use a system similar to fuzzy logic algorithms to avoid nearby objects.

On the topic of exploration during SLAM, frontier exploitation [35] is the most popular. There are some deviating strategies like random or sample-based destination or wall following [44]. In some cases, multiple strategies are combined [26]. Our exploration method uses a frontier method to explore the map. It then continues to visit as many different places as possible.

Comparable SLAM systems in this field are Cartographer [18] and the work done by Wang et al. [40]. Our SLAM system is inspired by the Cartographer implementation. However, our work deviates from the work of Hess et al. on numerous fronts. Most notably, our scan matcher does away with the complexities associated with maintaining a probability map. Instead, we opted for a much simpler approach. In our experience, this method does not perform noticeably differently compared to Hess et al. Moreover, our storage of local maps differs significantly from Cartographer to allow for dynamic environments.

The work of Wang et al. is an extension of the Cartographer that aims to also incorporate dynamics of the environment. They accomplish this by changing the cell update rules to also allow the removal of cell values.

## Chapter 3

# SLAM System Overview

Before we start the discussion on the our SLAM system, it is beneficial to first introduce an overview of our SLAM system. This allows us to introduce the hardware components of the system, such that we can relate to them further on in the text. Additionally, having a concrete example of a SLAM system layout, aids with the substantiation of certain design choices discussed further on in the text.

One of the goals of the project is to create a remotely controlled robot (see Figure 3.1), that can do SLAM while being controlled by a human or when driving itself autonomously. There are two autonomous driving modes. One is used to let the robot drive itself to a destination set by the user. The second mode is used to let the robot explorer and discover its environment. Moreover, in this mode, the robot will revisit places such that the map stays up to date.

The robot is controlled by a client program (see Figure 3.2) running on a remote pc. The client program connects to the server running on the robot. SLAM and other calculations are carried out on the PC rather than on the robot. Hence, the brain of the robot is located on the remote PC. This choice was mostly done for ease of development because a PC has much more computational resources than the often less powerful processor (e.g, Raspberry PI) located on the robot.

Another advantage of running the complex algorithms on a remote system is that the CPU of the robot consumes less power. Hence, the time the robot can be used on it often limited power source is increased.

The client has three modes it can be operated in. The first, and default, mode is manual control. Here, the user uses a (e.g., Xbox) controller to remotely control the robot. A video feed of the front camera is displayed on the client window along with a top-down view of the LIDAR data. This allows the user to precisely control the robot without colliding. Collisions are also signaled to the user using an onscreen indicator and haptic feedback.

A PWM power limit can also be set to get more precise control over the robot. This limit, maps the trigger range from zero to the set limit. Changing the power limit is necessary for two reasons. First, not every terrain has the same friction coefficient, hence different power is required to maintain the same speed on different terrain. Second, when the batteries lose charge, a higher PWM setting is required to maintain speed.

The second driving mode is an anonymous mode, meaning that the robot will drive itself. In this mode, the user sets a destination and the robot will attempt to drive to this point. The AI plans a route and will attempt to drive there. A path or route consists of several checkpoints, which the robot will attempt to visit in order. During the drive, the path is recalculated periodically. The reason is that we assume the environment to be dynamic, hence the path can also



Figure 3.1: This picture shows the robot we used during our research. The LIDAR is attached on top of the robot. The IMU is placed behind the LIDAR on the small PCB.

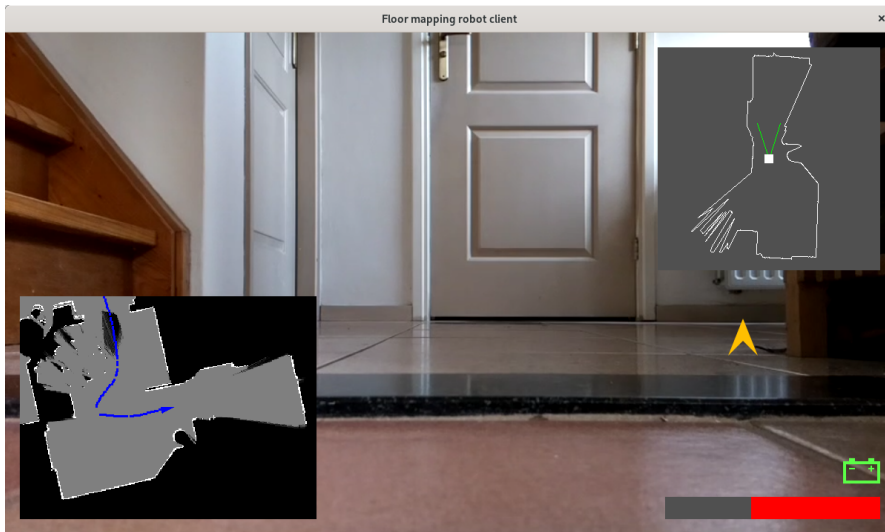


Figure 3.2: This figure shows the main view of the client UI. The overlay shows: on the left minimap of the current map; on the top right, the most recent LIDAR frame; on the bottom right, some status indicators. In this case the driving mode, battery status and current applied power and power limit.

change during a drive. Recalculating the path now and then also proved to be beneficial when the robot misses a checkpoint. Instead of turning and revisiting, the new path will simply start from the current robot location.

The third driving mode is also anonymous. Here, the robot will search the map for locations that will expand the map when visited. For example, the blind spots behind a doorway Figure 4.10. These types of locations are called frontiers. By visiting these frontiers, the map is expanded because new locations are discovered. After all frontiers are discovered, this drive mode goes into an update state, where it attempts to keep the map up-to-date.

If the robot collides when it is in anonymous driving mode, the robot goes into an error state that needs to be resolved by human intervention. Collisions can happen because of multiple reasons. For example, someone can simply step in front of the robot. However, a more likely reason comes from interruptions in the connection causing the robot to drive uncontrolled for a short time<sup>1</sup>.

The slam systems consists of a number of components that each carry out a small task. The combination of these tasks solves the SLAM and exploration problem. In our case, these components are. Preprocessing filters to condition the incoming sensor data. A scan matcher that matches a LIDAR obtained point cloud to the existing grid map. A grid map clean up pass, this pass removed unrealistic points from a grid map. The purpose of this pass is to make the SLAM process more robust. A loop-closure pass, this pass searches for scan matches between measurements taken at different time but at the same place. By adding extra loop-closure constraints to the Graph, the SLAM process is made more robust.

Additionally, there are some components related to exploration. The first component is the path finding component. This component is responsible for finding a path from the robot position to the destination. A path following component, make the robot follow the path. Finally, a exploration component tries to find a set of destinations and chooses one, this point is then visited with the aid of the path finding and following components.

### 3.1 Server

As mentioned before, the server runs on the robot and is responsible for two tasks. The first task is to control the wheel motors. The second task is to read sensor data and relay it to the client program.

The task of controlling the wheel motors is rather straightforward. All it needs to do is unpack the control packages from the network stream and send the control commands to the motor controller. There is, however, a caveat that needs to be dealt with. Namely, if the connection fails or is interrupted<sup>2</sup>, the robot will keep on driving uncontrollably.

The solution to this problem is to implement what we call heartbeats. Heartbeats are nonce messages from the client to the server, purely to tell the server that the connection is still good. Additionally, normal messages are also interpreted as heartbeat messages.

If the server has not received any message within the last 200 milliseconds, the motors are stopped. A new motion command from the client is then required to make the robot move again. To prevent continuous triggering of the heartbeat time, a small debounce time is defined. When the server starts receiving messages again, it needs to do so for at least the debounce period before the motors can be restarted.

---

<sup>1</sup>When the connection is lost for a longer period, the server on the robot will stop all motors to prevent a runaway.

<sup>2</sup>This happens more often than one would think.



The second task of the server is to gather data from the sensors on the robot. There are four sensors available on the platform. The first sensor is the LIDAR mounted on top of the robot. Because LIDAR data is acquired using a blocking IO call to the sensor, the task of reading and sending the LIDAR data is offloaded to a separate thread.

The second sensor is the camera mounted on the front of the robot. This sensor also utilizes a blocking IO call to retrieve data. Hence, the task of gathering data is also moved to a separate thread. Moreover, to access the camera a *raspivid* child process streams the data to the server program. We specify a fairly small resolution and frame rate of 960 by 540 at 20 fps. We do this to reduce the amount of data that needs to be transferred. The *raspivid* program provides us with an h.264 video stream.

The third sensor is the IMU sensor. The server polls the IMU sensor at 100 Hz. The BNO055 IMU, runs in fusion mode. In this mode, a microprocessor in the chip talks to the actual sensors and fuses the data. The data is also integrated over time between polls from the server. This implies that there is no need to worry about polling at the exact rates of the actual sensors.

In the network packages of the LIDAR and IMU data, a timestamp is included. This timestamp is measured in milliseconds and starts when a connection between client and server is established. The timestamp is used by the SLAM algorithm to associate IMU and LIDAR frames to each other.

From the IMU chip, the heading, roll and pitch are obtained and packed into a network packet. To prevent sending too many packets, 10 IMU packets are combined together and then send as one large packet client. Because we want to read data from the IMU at precisely 100 Hz, we also moved the entire IMU task to its own thread.

The last sensor on the robot platform is the battery voltage level. This sensor is not required for SLAM, but it does provide useful information. The battery voltage gives an indication of the battery level, although it is not perfect. This discrepancy, is caused by the load of the battery varying heavily during use. Accelerating and moving the motors under heavy load significantly drops the battery voltage.

To combat this problem of fluctuating voltages and prevent sending the battery level too often, we average the value. The battery level is only sent to the client if the level changes.

This solution poses another problem. That is, we do want to get notified when the battery drops below a certain threshold. Because low voltages can cause the Raspberry PI to reboot. By reporting these short voltage drops, or brownouts, action can be taken to prevent them. Most notably, reducing the applied wheel power.

To implement the brownout system, the main loop of the server frequently checks the battery voltage. If this drops below a certain threshold, a brownout network packet is generated and sent to the client.

# Chapter 4

## dynSLAM

This chapter describes the dynSLAM algorithm in great detail. We start with an overview of how the dynSLAM components are related to each other, and how the data flows between them. In the consecutive subsections, we discuss the design choices for each component in the dynSLAM application.

Figures 4.1, 4.2 and 4.3 show four flowcharts of dynSLAMs SLAM pipeline. The flowcharts are merely split up for clarity. The first (A) flowchart shows the steps taken from a LIDAR observation to the client program. The data is sent over the network and finally stored in a queue at the client side. The second (B) flowchart shows the steps for the IMU data. This process is similar to the LIDAR data stage. In the client, both the LIDAR data and IMU data are stored in a shared memory location. A worker thread can access this data and carries out all the necessary SLAM steps.

The process done by the SLAM worker thread is shown in flowcharts C and D. The flowchart is split up for convenience, the last flowchart shows the finalization step.

Starting with flowchart C, it is clear that the worker thread waits for a point cloud and IMU data to arrive. Once enough new data is available, the point cloud is popped from queue. The point cloud is then fed through a pipeline that attempts to correct or remove unnecessary data. Each step in this pipeline is called a pass. The first pass is the 'PC clamp pass' which removes all points that have a length greater than 4 meters from the origin (at this stage, the LIDAR is positioned at the origin of a point cloud). The reason for this pass is that points measured beyond this distance become erroneous.

Then the pipeline executes a pass that works on the IMU data associated with the current point cloud being processed (i.e., the IMU data measured between the start and end of a point

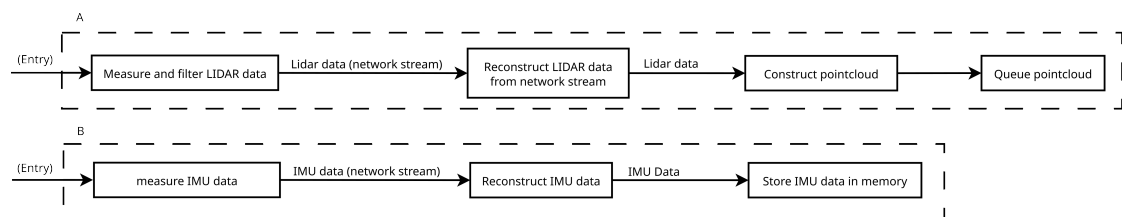


Figure 4.1: These flowcharts, show the transmission of SLAM data from the server to the client. The LIDAR and IMU data are obtained by the sensors on the robot and then sent over the network. The client then unpacks the data. The unpacked data is stored for further processing.

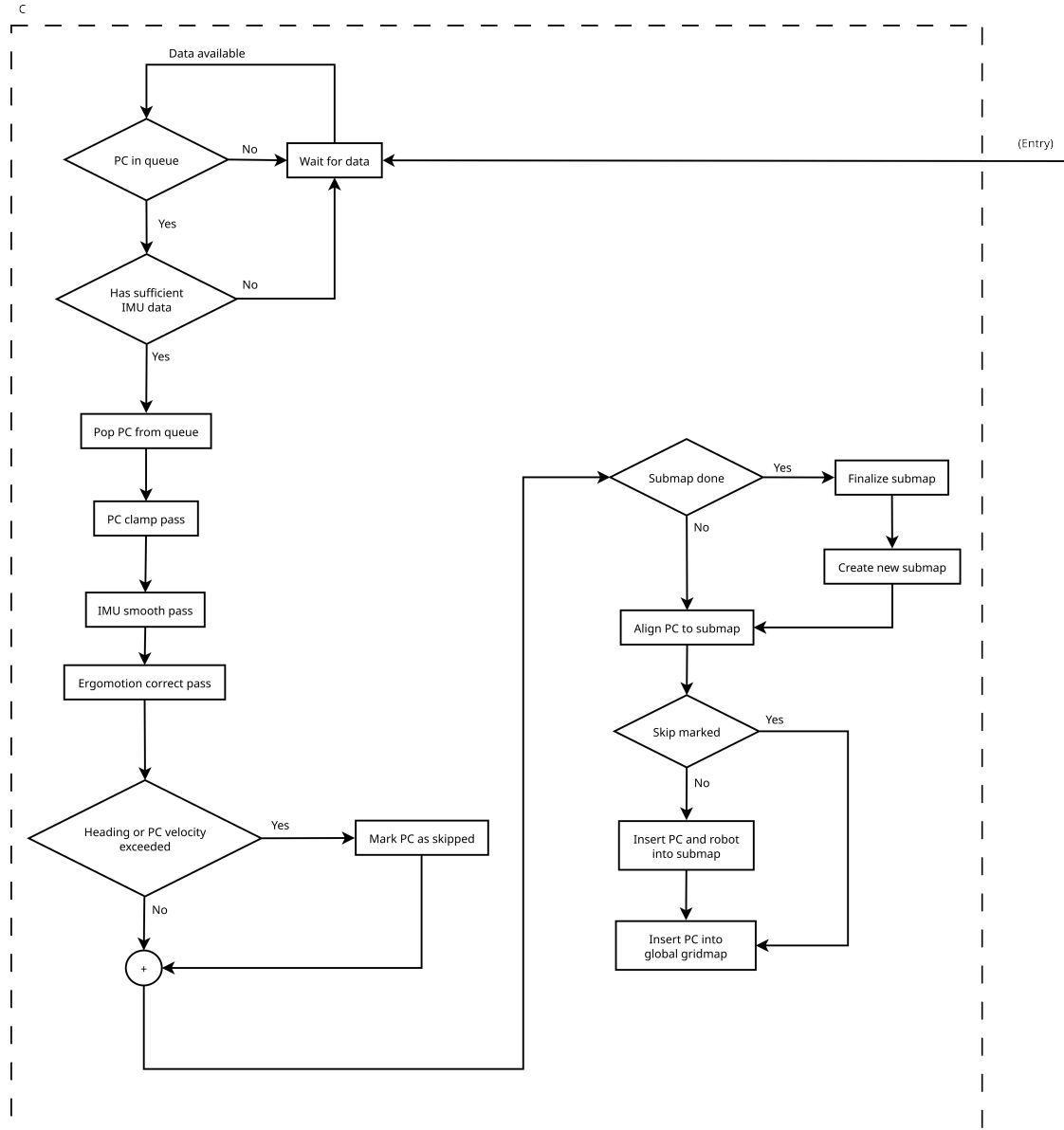


Figure 4.2: Flowchart C, shows the process of inserting a new LIDAR point cloud into a local map.

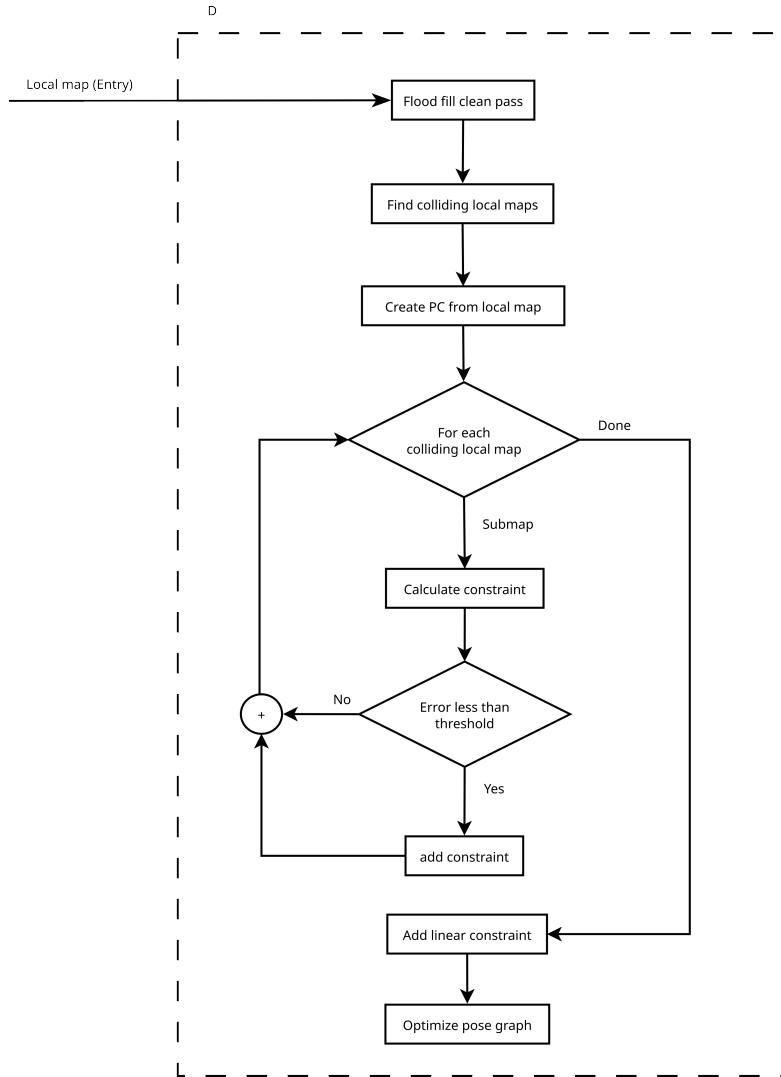


Figure 4.3: Flowchart D, shows the process of local map finalization and loop closure. This flowchart is the expansions of the ‘finalize submap’ block in flowchart C.

cloud). The pass, 'IMU smooth pass' removes peaks in the IMU heading data. It is not known exactly why these peaks exist. Perhaps they are caused by sudden shocks and accelerations of the IMU sensor.

Now that the IMU data and point cloud data are more consistent, an egomotion pass is executed. This pass takes both the point cloud and IMU data and produces a new point cloud. The newly generated point cloud is a point cloud corrected for heading (yaw) velocity. Moreover, the new point cloud also obtains an absolute rotation (i.e., north is always in the same direction).

From the IMU data, we can also calculate the heading, pitch, and roll velocity. If this velocity exceeds a certain threshold value, the point cloud is marked as 'skip'. Point clouds marked as skipped are not added to the local map. The reason is that these measurements are typically distorted, and not adding them tends to leave a better result. Point clouds marked as 'skipped' are however still pushed further along the pipeline, because we can still extract a transform from it. This transformation is necessary because otherwise, we cloud lose track of the robot position if multiple distorted measurements are taken after each other.

The last step in flowchart C shows how the point cloud is inserted into the global grid map. This global grid map is not necessary for our SLAM approach and only serves as a visual map for the user. In fact, efficiently maintaining a global grid map is not an easy task. Further on, we devote an entire subsection to this problem.

When inserting a processed point cloud into a local map, a check for completeness of the local map is made. Local maps are complete if 50 point clouds are added to them. If a local map is complete, it is finalized and the next empty local map is created. The finalization process is described in flowchart D.

The first step in the finalization process is to do a 'flood fill cleanup pass' this pass removes misplaced points behind walls and objects. Then, by finding overlapping bounding boxes, it is possible to find earlier local maps that share parts of the same region. By converting the most recent local map to a point cloud, our point cloud to grid map scan matcher can be reused to find a relative transformation between local maps. These transformations then form loop closure constraints for GraphSLAM [40]. Another constraint is found by calculating the translation from the previous to the most recent local map. This constraint is always present and necessary to prevent the constraint graph from becoming disjunct, in the case that there are no other constants. The final step is to optimize the constraint graph, this is done using the Ceres solver [1].

## 4.1 dynSLAM Components

In this section, we give a more in-depth explanation of the components used by dynSLAM. The , however, relate to SLAM, pathfinding and exploration. The order in which they are listed in this thesis, is also the order in which they where developed. Or at least the order in which development started. During the development, all components where constantly tweaked and improved. Depending on what was the limiting factor, we decided which component to improve.

It was exactly this way of reasoning, and the design goals set at the beginning of the project, that led us to a choice of components to implement. During development, we have build up a rather extensive list of tweaks and components, that can still be added to the project. However, most of these ideas where never implemented, this is because we prioritized on the parts of the software that where performing the worst. We only added new components when we felt like the current implementation was performing reliably.

Finally, at the end of this thesis, we have added a Future Work section. In here, we discuss what we feel like is the next logical step when it comes to extending the project.

The list stated below sums up the algorithmic contributions made in the paper. We discuss the algorithms based on the implementation in the dynSLAM application. The most important reason being that some design choices are directly related to the real world scenario, presented by the choice of hardware and indoor environment.

- preprocessing filters
- point cloud to grid map scan matching and point cloud insertion
- floodfill clean up pass, this pass attempts to remove unrealistic points from a grid map
- loop closure in grid maps
- fast global grid map updates
- path finding
- path following
- exploration

#### 4.1.1 SLAM - Preprocessing Filters

We have already discussed the SLAM pipeline that is used in our work, and introduced the filters that are being used to process the incoming data. This subsection, goes into detail about these filters and explain their precise working principles.

The first notable pass is the IMU smooth pass. This pass takes the newly received IMU data and smooths out peaks in the heading data. The peaks are generally only a single measurement point and have an unrealistically different value. Figure 4.4, shows some example IMU heading data before and after filtering.

The smoothing pass searches through the IMU data and replaces these peaks with the average value of the point before and after the anomaly. To detect peaks reliable, we use the following method. Around each point  $i$ , we calculate the speed relative to the point before and after it. Of the two speeds, we determine the lowest absolute value. This value is compared to some threshold, and when it is higher, the point is corrected. The reason for choosing the lowest value is because it acts as a ‘sort of AND-gate’, requiring both points to be above the threshold. This technique makes sure to correct outlying point and not the point before or after it.

The described technique requires that for two consecutive points  $i$  and  $i + 1$ , it is possible to calculate the angular difference. Because the heading data is obtained in polar coordinates, this can fail when the robot makes an entire turn over the  $\theta$ -zero axis. When this happens, the difference between the two points obtains an extra  $\pm 2\pi$  term.

To solve this problem, the heading angles are adjusted such that  $\angle i + 1 - \angle i$  always results in a correct difference. This is done by counting the total number of rotations ( $n$ ) of the robot. Upon receiving new IMU data,  $2\pi n$  is added to the heading before it is inserted into the IMU data structure. The method that achieves this is stated in Listing 4.1. Note, this method assumes that  $\forall i : |\angle i + 1 - \angle i| < \pi$ .

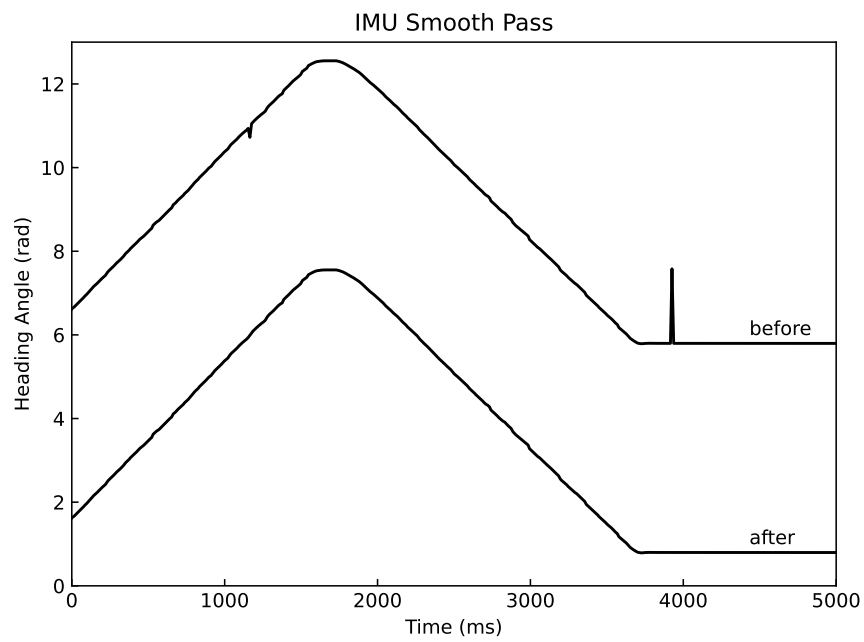


Figure 4.4: This figure shows the effects of the IMU smoothing pass filter. The lines are vertically offset by 5 radians to make them distinguishable. The top line shows raw heading data, the bottom line shows the processed data. Is clear the two peaks are removed from the IMU data after smoothing.

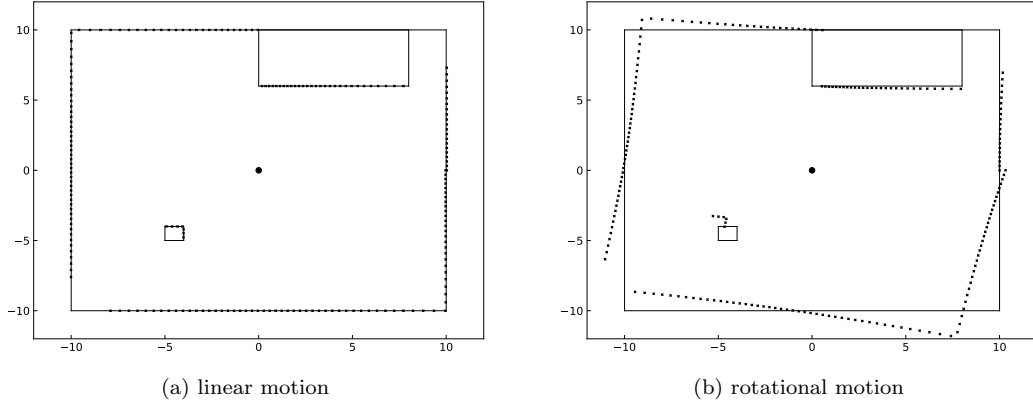


Figure 4.5: This figure shows a simulation of the scan points obtained by a moving or rotation LIDAR. (a) shows the distortion for linear motion. (b) shows the distortion when the LIDAR is rotating. The linear and rotational speed are: 0.7 m/s and 2.5 rad/s respectively. These speeds are easily obtainable with the real life robot.

```

void make_continuous(double headings[N]) {
    int rotations = 0;
    for (size_t i = 1; i < N; i++) {
        double heading0 = heading[i - 1];
        double heading1 = heading[i] + 2 * M_PI * rotations;
        double diff = heading1 - heading0;
        if (diff < -M_PI)
            rotations++;
        else if (diff > M_PI)
            rotations--;

        heading[i] += 2 * M_PI * rotations;
    }
}

```

Listing 4.1: This algorithm shows how to make angular coordinates continuous.

Another important pass is the egomotion correction pass. This pass takes the IMU and point cloud data and corrects the motion of the LIDAR during the measurement. Figure 4.5 shows how a rectangular room is observed by a moving LIDAR.

From Figure 4.5 it is evident that rotational motion needs to be addressed and linear motion can be ignored. There are two reasons that we can ignore linear motion. The first one is that linear speeds are low, hence distortion is small. On other types of robots (e.g., a car) this might not be the case and linear motion also needs correcting. The other reason is that the linear distortion is overshadowed by other measurement noise. Because of this noise, it is nonsensical to choose a grid map cell size that is small enough to store this noise in great detail. Hence, even if linear motion is corrected, in our case at least, the obtained correction is then lost again in the grid map.

Luckily, there is no need to solve the linear motion problem. Because in our case this would



be a difficult problem to solve. Our robot is not equipped with a device that can accurately measure its speed (e.g., wheel encoders). Hence, the only way to obtain some sort of motion value is by doing SLAM and looking at the robot positions between scans. But to do the linear motion correction pass, the latest position of the robot needs to be known. Hence, we now need to make an iterative process to first get an initial guess of the robot's position. Derive a speed and then correct and do everything over again. Moreover, the speeds obtained from SLAM data tend to be spiky so smoothing of the velocity data is required.

Correcting for rotational motion of the LIDAR is done with the aid of the IMU sensor. From the IMU, 100 Hz heading data is available. This gives us around 8 heading points during a single LIDAR scan, because its frequency is around 12 Hz. Equation 4.1 shows how to obtain a global direction ( $\theta^o$ ) from the IMU direction ( $\theta^i$ ) and the angle of a LIDAR point ( $\theta^l$ ). Note,  $\theta^i$  is the global heading of the robot because the IMU is rigidly mounted on the robot platform.

$$\theta^o(t) = \theta^i(t) + \theta^l(t) \quad (4.1)$$

Because the LIDAR and the IMU take discrete samples, only the discrete values  $\theta_i^l$  and  $\theta_i^l$  are available. To calculate the global heading angles  $\theta_i^o$ , it is possible to discretize them using the already existing LIDAR samples  $\theta_i^l$ . However, the IMU  $\theta_i^i$  sample timestamps do not align with the LIDAR timestamps. To solve this problem, it is possible to use linear interpolation and interpolate the value of  $\theta^i(t)$ , for each  $\theta_i^l$ .

Initially, only the starting and stopping timestamp of a LIDAR frame (a full rotation) is known. To calculate the timestamp of each point in the frame, we use the following method. First, the angles are made continuous, in the same manner as discussed in 4.1. We can then calculate the corresponding timestamp for each point using Equation 4.2

$$t_i = t_0 + (t_{n-1} - t_0) \cdot \frac{\theta_i^l - \theta_0^l}{\theta_{n-1}^l - \theta_0^l} \quad (4.2)$$

Here,  $t_i$  is the timestamp for the  $i$ -th measurement.  $t_0$  and  $t_{n-1}$  are the starting and ending time of the LIDAR frame. Likewise,  $\theta_0^l$  and  $\theta_{n-1}^l$  are the first and the last heading direction in the LIDAR frame.

We now need a way to calculate  $\theta_i^i = \theta^i(t_i)$ . To do this, we simply linearly interpolate between the datum point before and after  $t_i$ . Because, the IMU data is stored in an array already sorted by time, we use binary search to quickly find the right offset in the array.

Finally, we carry out the correction step in Equation 4.1. This process produces a new point cloud that has corrected points and global heading values. Meaning that the north always points in the same direction, even between LIDAR frames. This is possible due to the global yaw sensor of the IMU.

#### 4.1.2 SLAM - Scan Matching

The scan matching scheme we utilize is a scan to grid-map based approach. This scan matcher tries to find a transform that matches a point cloud to a grid-map (image) of the current map of the environment. In fact, the scan matcher keeps track of two grid maps to function properly. The first map is the hit (or occupation) map, storing occupied areas. The second map is the miss-map, storing observed empty areas. Both maps contain cells with values ranging from zero to one. Here, zero denotes an unknown state and one denotes a hit or miss state, depending on the map type. Values in between are also valid; however, with our algorithm values converge quickly to one or zero.

The usage of scan-to-map scan matching has some advantages and some disadvantages. The most important advantage is, given a point  $(x, y)$ , finding its matching error in the grid map is a

$\mathcal{O}(1)$  operation. This is because it can simply convert the coordinates  $(x, y)$  to the right indices of the 2D array and do a memory lookup. Hence, determining the total error for a point cloud, has complexity  $\mathcal{O}(n)$  (cf. Scan to scan matching has at least complexity  $\mathcal{O}(n \log(n))$ ).

Another benefit of the scan to map approach is that it incorporates previous measurements. The grid map is extended after each point cloud that is added to it. This makes the process of scan matching more resilient to one-off erroneous measurements (e.g., because the robot drove over a bump).

The first downside of using a grid map is its memory consumption. dynSLAM uses a grid map of 2048 by 2048 cells with a cell size of 2 cm. This roughly spans a 40 m by 40 m area that can be mapped. Because we store cell values with double precision<sup>1</sup> and save a hit and miss map, the total memory consumption is 64 MiB.

Because the memory consumption grows quadratically, this method does not scale well to large areas. A solution would be to divide the map into chunks and only allocate memory where needed.

Another disadvantage is that updating an entire grid map is computationally expensive. It is possible to do SLAM without making large updates to the grid map. However, in our system, there are some operations that require updating large portions of the global map. In the subsection about global map updates, the update process of the global map is discussed in much more detail.

In our case, scan matching is easy. Because we use the IMU to determine the robot heading, only the linear transformation needs to be found. Hence our scan matching problem reduces from three dimensional to two dimensional. Moreover, the search area is a 20 cm square around the previous robot position. Also, our grid map cell size is 2 cm; hence, there are 100 possible transformations. The search area limits the speed at which the robot can travel, in our case, this is 1.25 m/s assuming a LIDAR frequency of 12.5 Hz. Higher speeds are rarely reached because the LIDAR image becomes too unstable at this point.

Because only 100 possible transformations need to be considered, a brute force approach is used. However, if the search area becomes larger or computational resources are limited, a method discussed by Hess et al. [18] can be used to optimize the scan matching process.

To compare scan matches, an error function needs to be defined. The function we use is given by Equation 4.3.

$$e = \frac{1}{n} \sum_{i=0}^{n-1} |1 - \text{hit}(x_i, y_i)| \quad (4.3)$$

Here,  $\text{hit}(x, y)$  is the value of the hit map at location  $(x, y)$ .  $x_i, y_i$  are points in the point cloud we are matching. Note, it is assumed that all points are within the grid map bounds. The actual implementation simply skips points outside of the grid map.

The error function always gives a value between zero and one. Where zero is a perfect match and one is a complete mismatch. A lot can be said about error functions. Like squared error, absolute, or Huber error. Incorporating the miss-map or not. Taking the neighboring cells into account. etcetera. In practice, we found that most of these choices make little to no difference, compared to Equation 4.3. Hence, we ended up using this simple error function.

A new point cloud is inserted into a grid map as follows. First find the transformation that yields the lowest error. In rare cases when all error values are the same, assume no change in transformation. Then we apply the optimized transformation to the point cloud and add it to the grid map.

---

<sup>1</sup>a single byte would probably suffice.

Insertion of a point cloud into a grid map happens as follows. For every point in the point cloud that is also in the bounds of the grid map, update the hit and miss cell as in Equation 4.4.

$$\begin{aligned}\text{hit}'(x, y) &= \min(1, \text{hit}(x, y) + 0.1) \\ \text{miss}'(x, y) &= \max(0, \text{miss}(x, y) - 0.05)\end{aligned}\tag{4.4}$$

Here, the  $\text{hit}'(x, y)$  and  $\text{miss}'(x, y)$  are the updated grid map cells. At places where the LIDAR measures an object, the hit cell values are increased. Simultaneously, miss cells at these locations are decreased by a small amount. Decreasing the miss cells has the effect that changes in the environment are quickly incorporated. Moreover, it also removes outliers, on the condition that the outlier lands in an observable place (i.e., not behind a wall or object).

The choice of the increment and decrement values is based purely on experimental findings. However, the actual values do not seem to matter too much. In essence, larger values make the map converge faster to its final form. However, errors also accumulate faster through the map.

Moreover, we also tried the probability-based grid map update approach of Hess et al. [18]. However, in our case, this did not work better than the much simpler approach presented here. In fact, our approach has the benefit that the values are guaranteed to stay within a zero to one interval.

The second part of inserting the point cloud in the grid map is to add the measured empty (miss) space. This space is obtained by drawing lines from the robot to the points in the point cloud. To draw lines between the robot and the hit points, we use the DDA algorithm. This gives the cells in a straight line between the robot and the hit locations. These cells are updated according to Equation 4.5.

$$\begin{aligned}\text{miss}'(x, y) &= \min(1, \text{miss}(x, y) + 0.1) \\ \text{hit}'(x, y) &= \max(0, \text{hit}(x, y) - 0.05)\end{aligned}\tag{4.5}$$

This update method is similar to the one for the hits. The same reasoning holds.

A shortcoming of only drawing lines, is that long enough lines diverge by more than cell size. As a result, the miss area is not fully covered. A solution to this problem is to use a triangular area, instead of a line. The triangles have vertices  $[(x_i, y_i), (x_{i+1}, y_{i+1}), (x_c, y_c)]$ . Where, as before  $(x_i, y_i)$  are LIDAR measurements and  $(x_c, y_c)$  is the center of the point cloud (or robot position).

In practice, only drawing lines works good enough. In most cases multiple measurements cause the miss area to fill in nicely. Moreover, the maximal length we set for a LIDAR light ray also reduces the above mentioned problem.

### 4.1.3 SLAM - flood fill cleanup

Before it is possible to add a new local map to the graph (for GraphSLAM), a clean-up step is performed. This step attempts to remove unrealistic points on the (local) map. The update rules in the previous subsection ensure that errors and environment changes are quickly removed from the map. However, misplaced points behind a solid object (e.g., a wall, see Figure 4.6) are not removed by this process. We use a (to our knowledge) novel method to find and remove most of these points. The cleanup algorithm uses flood fill for traversing a grid map, as a result, we often refer to this algorithm as the flood fill cleanup algorithm.

The resulting grid map produced by the flood fill cleanup algorithm has thinner walls and fewer erroneous points behind objects. A grid map that has these properties will result in better scan matches. This is because thick walls and objects allow for more freedom in the scan matching



Figure 4.6: This figure shows misplaced points behind a wall. Our flood fill cleanup pass attempts to remove such points.

process than thin walls. As a result, when inserting many point clouds without correcting the grid map, walls and objects may grow to unrealistic sizes.

The derivation of the algorithm starts with the assumption that any point that can be observed should be reachable and is at most one cell thick. Other cells are considered invalid. A reachable cell, is a cell that has a path from some valid robot position to the cell. The path is only allowed to cover miss-cells on the grid map. Checking for every cell for a possible reachable path is computationally intensive. However, we can do smoothing a bit more clever to determine valid cells.

Instead of checking the validity of each cell individually, we calculate a set of cells that contains all reachable cells. The calculation of this set is done by utilizing the flood fill algorithm. During the build-up phase of the local map, the robot positions are stored as well. These robot positions can then be used to initially fill the flood fill queue. Then we iterate over the queue until it is empty. In each iteration, an item (cell) is removed from the queue and added to a visited set to prevent revisiting. Moreover, it is added to the reachable set if it is a hit or miss cell. If the current cell is a miss cell, the neighboring cells that are miss cells and are unvisited are added to the queue.

At some point, the queue is empty, and we are left with a set of reachable cells. We want to decrease the cell values of all nonreachable cells with a certain factor. This factor determines the aggressiveness of the algorithm. In our case, factor is set to 1, meaning that we completely remove all nonreachable cells from the grid map. Algorithm 4.2 shows a simplified version of the just described algorithm. The factor value is denoted  $f$  in the code listing.

```

void floodfill_clean(GridMap &gm, double f) {
    set<Vec2i> queue;
    set<Vec2i> visited;
    set<Vec2i> reachable;

    for (Vec2i &coord: gm.robot_positions)
        queue.insert(coord);

    while (!queue.empty()) {
        Vec2i coord = queue.pop();
        visited.insert(coord);

        if (gm.hit(coord))
            reachable.insert(coord);
        if (gm.miss(coord)) {
            reachable.insert(coord);

            for (Vec2i neig: neighbors(coord))
                if (!visited(coord) && !gm.hit(coord))
                    queue.insert(neig);
        }
    }

    for (Vec2i coord: gm) {
        if (!reachable(coord))
            continue;

        gm.hit(coord) = clamp(gm.hit(coord) - f, 0, 1);
        gm.miss(coord) = clamp(gm.miss(coord) - f, 0, 1);
    }
}

```

Listing 4.2: This code listing shows the steps the flood fill cleanup algorithm takes. It firsts determines a set of reachable cells. Then it subtracts a factor value (f) from every nonreachable cell.

#### 4.1.4 SLAM - Grid-maps and Loop Closure

A downside of grid-map-based SLAM is that all information about how the map was built is lost. Because of this, it becomes impossible to do any form of GraphSLAM and loop closure. A solution to the problem [18], is to create a graph of smaller local maps. We create new local maps every 50 LIDAR frames (every 4 seconds). These local maps are then placed into a graph, the edges between two local maps contain the transformation between the maps.

In the graph, each local map is connected to its predecessor because this transformation is simply known from scan matching (i.e., when creating a new sub-map, the position of the robot is known; hence, the origin of the new local map). Moreover, a loop closing scan matcher is used to find matches between nonconsecutive local maps. Figure 4.7 shows an example of a local map graph.

Before adding a local map to the graph, the flood fill cleanup algorithm is used to remove

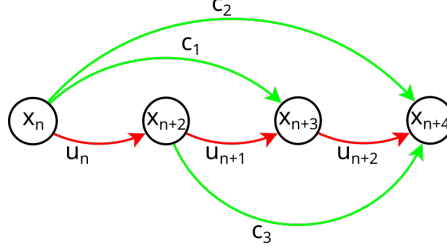


Figure 4.7: This figure shows a GraphSLAM graph. The nodes ( $x_i$ ) represent local maps. The arrows contain relative transformations ( $u_i$  and  $c_i$ ) from one map to the other. The red arrows ( $u_i$ ) represent odometry constraints, obtained from SLAM. The green arrows ( $c_i$ ) contain loop closure constraints, found by scan matching a local map to all previous (overlapping) local maps.

any unrealistic points and reduce the wall thickness. This step is necessary because we want to convert the finalized local map back into a point cloud. For this step to work correctly, it is paramount that object walls are thin and well defined. Moreover, the cleanup step also removes points; hence, it improves the performance of the scan matching process. However, the net performance improvement is offset by the complexity of the flood fill cleanup pass ( $\mathcal{O}(n^2)$ ).

The next task in the loop closure process, is to find possible matches between local maps. This is done by calculating the bounding box of every local map, and checking for collisions between previous local maps and the current one. Any local map that collides with the current local map is considered a loop closure candidate.

Then the current local map is converted into a point cloud. By scan matching the just created point cloud to the loop closure candidates, we find relative transformations between the two maps. If the error of the scan matcher is below a certain threshold, the constraint is added to the graph.

Once all constraints have been added to the graph, it can be optimized using Ceres. This process moves the positions of the local maps such that some error is minimized. The error function is given by Equation 4.6, the summation goes over all edges ( $e_{ij}$ ) in the graph. Moreover,  $x_i$  is the location of local map  $i$ ,  $e_{ij}$  the transformation to go from local map  $i$  to  $j$ .

$$\sum_{\forall e_{ij}} (x_i - x_j - e_{ij})^2 \quad (4.6)$$

#### 4.1.5 SLAM - Global Map Updates

Up to this point, we have only discussed how to keep track of the location of the robot and how to do loop closure. However, when it comes to displaying the mapped areas, we have left the reader in the dark. That is because it is not vital for the SLAM algorithm to maintain a global map throughout the mapping phase. The only thing required is to maintain enough data such that it is possible to generate a global map at any point throughout the mapping process.

In offline SLAM, it is only required to generate a global map after all data is processed. In our case though such a solution is unsatisfactory, because we require a live updating global map in the client. However, it is beneficial to first look at how to generate a global grid map from the SLAM-generated data. After we have established this procedure, we discuss the adaptations needed to make the algorithm live updating.

As described in earlier sections, measurements are subdivided into local maps. These local maps consist of 50 LIDAR scans. Each local map position in the global map, can be updated

during a loop closure step. Along with each local map, we also store the point clouds that build up the map.

Constructing a global grid map from the local maps is done by inserting point clouds into the global map. These point clouds are inserted based on age, starting with the oldest point clouds first. Each point cloud is associated with a local map, which determines the position of the point cloud in the global map. Insertion of point clouds into the global map happens via the already discussed method.

After all point clouds are inserted, a final flood fill clean pass is executed to remove unwanted measurements. Note, that it is not needed to run the clean-up pass during the insertion of the point clouds into the global grid map. This is because during this phase no scan matches are carried out; hence, there is no need to have a nice clean grid map during the construction phase.

There are however two problems with the just described solution. The first one is that the number of stored point clouds increases linearly with the run time of the robot. Hence, the number of point clouds that need to be inserted also increases linearly with time. As a result, the time it takes to build a global grid map becomes larger as time moves on. The second problem with the approach is that we assume the environment to be dynamic. Hence it does not make sense to incorporate old local maps that have changed over time.

We solve these two problems using a single solution. The solution works as follows, a history grid map with a cell size of 50 cm is constructed. The cells of this grid map store indices of the most recent local map that was observed from this location. The default value of this map is a special value, indicating that there is no observation in this cell.

Using this history map, we can make a selection of active local maps. Active local maps are maps that have their index value stored in the history map. By inserting only these active local maps in the global map, we obtain the most recent global grid map. Insertion of the active local maps happens as discussed before.

Because we want to present the user with a real-time 2D map, we need to constantly update the global grid map. However, the method presented above is too slow to run at a high frame rate. Moreover, it would be a waste of computational resources to constantly render the entire global map. Conveniently, new LIDAR observations can simply be added to the old map, without redrawing it entirely.

Redrawing the global grid-map is only required after a loop closure step. Because after this step a large portion of the map may suddenly change. However, after a loop closure step, often two cases apply. Either only the last local map changed position, or the changes are local only.

The first case where only the most recent local maps changed position, can be solved easily. Simply keep an old version of the global map. This version is the global map without any point clouds from the most recent local map on it. This way, the global map can be constructed from the old global map by inserting the latest local map into it. Because there are only 50 point clouds in a local map, this operation is quite fast. However, a flood fill cleanup pass is still required on the global map after this operation.

The second case where there are only local changes is solved by calculating a bounding box around the changes. The bounding box is a simple rectangle. Only the bounded area is then updated on the global map. This happens as follows. First, the area in the bounding box is cleared. Then all active local sub-maps are found in the bounding box. This is done by looking for intersections with the update box and the bounding boxes of the local maps. Then all local maps within the bounding box are redrawn onto the global map. Finally, a flood fill clean phase cleans the global map.

### 4.1.6 Path Finding

In this subsection, we discuss the method for finding a path from the robot to some destination. The main idea of our method is to find a path that only covers empty cells on the global map. We use A\* and a path pruning technique to find such a path. Figure 5.9 shows an example of a path.

Normal A\* does not account for the width of the robot. As a result, the path will lay too close to objects, which causes the robot to collide. The solution for this problem is simply to ignore any points that have a hit cell in their vicinity. In our case, we consider a cell valid if it has a coordinate within the bounds of the grid-map, and has no hit cells in a square around it. The square has sides of 80 cm. Note, that the robot has sides of roughly 22 cm. This may seem like a lot of headroom. However, during testing, this number seemed to produce the most reliable course. The choice for this parameter is a fine balance between driving speed and headroom around the robot. This required headroom, is a result of inaccuracies in the map and robot position. Moreover, there is always a small delay between reality and the map. Having a little extra headroom helps to prevent collisions, and allows some time to correct for errors.

Another problem with the path generated by A\* is that it consists of many cells. This is an unwanted result, because for points close to the robot, the angle between the robot heading and the waypoint is poorly defined. Hence, it is beneficial to have as few cells in the path as possible, reducing the time the robot spends close to a waypoint. As a result, we want to remove as many unneeded waypoints as possible.

Our solution is to remove any unnecessary points from the dense path. Ideally, we want to remove all points such that the path consists of a minimal amount of waypoints, and the lines between the waypoints only cross valid cells. Finding this exact path is quite a difficult challenge since there are  $\mathcal{O}(2^n)$  possible paths to consider.

However, we can settle for a heuristic approach where the solution is not perfect but still satisfactory. Binary search is used to quickly find the furthest point away from some cell, such that the straight line in between the two points is still valid. These two points are then added to the pruned path. Moreover, we run the algorithm until there are no line sections left. The algorithm starts at the beginning of the path. The code of the algorithm is stated in code Listing 4.3.



```

int begin = 0;
while (begin < path.size()) {
    int l = begin;
    int r = path.size() - 1;
    while (l <= r) {
        int m = (l + r) / 2;

        // check of the vector between l and m is valid
        Vec2i x = path[begin];
        Vec2i y = path[m];
        Vec2i d = y - x;
        double steps = max(abs(d[0]), abs(d[1]));
        bool valid = true;
        for (int step = 0; step < steps; step++) {
            Vec2i v = x + (d * step / steps);

            if (!valid(gm, v)) {
                valid = false;
                break;
            }
        }

        if (valid)
            l = m + 1;
        else
            r = m - 1;
    }

    // add the waypoint
    route.push_back(path[r]);
    begin = r + 1;
}

```

Listing 4.3: This algorithm shows the pruning process of the A\* path. Note the A\* path is placed in path, while the output path is stored in route.

#### 4.1.7 Path Following

Now that we have defined a way to generate a path, it is time to have the robot drive it autonomously. We first give the minimal implementation to do path following and then present problems and adaptations of the algorithm.

The path following algorithm runs on the client machine. When the autopilot is activated, the user controls are inhibited. Instead, motion controls are calculated by the autopilot algorithm and sent to the robot.

The autopilot always tries to drive to the first waypoint in the route. If the robot is within a 5 cm radius of the first waypoint, the waypoint is removed from the path. Any other waypoints are removed from the front of the path, until there are no more waypoints within the 5 cm radius.

At this point, we are either done following the route (i.e., no more waypoints are left in the route), or we have a new target waypoint at least 5 cm away from the robot position. The next

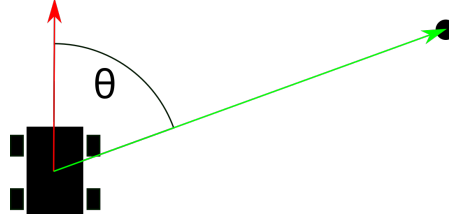


Figure 4.8: This figure shows how the heading difference is calculated. The red vector shows the heading of the robot. The green arrow shows the vector from the robot to the next waypoint. The difference angle between the two vectors ( $\theta$ ) needs to be reduced to some small value before the robot can start driving in a straight line.

step is to calculate the difference angle between the robot heading and the target, Figure 4.8 graphically illustrates this process.

Based on the difference angle, the robot either does an in-place turning manoeuvre or it drives in a straight line. If the absolute value of the difference angle is greater than 0.2 radians, the robot turns towards the waypoint. In the other case, the robot to drive forwards towards the waypoint. Note, the difference angle is repeatedly checked, to decided whether to drive or turn.

The proposed method works; although, there are some problems with it. These problems stem from inaccuracy in knowledge of the robot's position and the time delay between reality and global map updates. As a result, the robot can exhibit oscillatory behavior. A solution, modulate the applied wheel power. By decreasing the wheel power, the turning action takes longer. As a result, the information has more time to travel through the delayed system, and overreaction is prevented. Currently, turning operations are performed at 20% of the total power.

Driving in a straight line poses less of a problem. However, overshooting the target is possible. When this happens, the robot is too far passed the target for it to be removed from the list. As a result, the robot needs to turn 180 degrees, to reach the target again. This scene is awkward to look at and should be prevented as much as possible. We lessen this problem in two ways. The first method is to reduce the wheel power (and thus speed) when the robot gets close to a target. Moreover, the route is also recalculated now and then. As a result, any targets the robot has passed by do not pose a problem anymore. Recalculation should be done at an interval more or less equal to the time it takes the robot to turn around. In our case, this time is every 2 seconds.

It may seem computationally wasteful to recalculate the route now and then. However, recalculation is required anyway to incorporate dynamics of the environment into the route. For example, without recalculations, stepping in front of the robot would simply result in the AI driver waiting until the blockade is canceled.

We use the global map for finding a route to the destination. Moreover, the map is also used to calculate the drive controls. However, only using the global map to generate these control commands has some drawbacks. The main issue is that the global map always lags behind reality somewhat.

There is however more recent LIDAR data available. This data is the latest LIDAR point cloud, where each point is a reflection of a nearby object (see the top right overlay in Figure 3.2). Because this data has much less latency than the global grid map, it is of good use when avoiding collisions.

We use the following method to correct the robot motion based on the most recent LIDAR data. Around the robot, we identify six segments as in Figure 4.9. Upton arrival of new LIDAR data, the number of points that fall into each segment are counted. When a motor control command is issued and the robot drives a straight line, the motion controls are adjusted such

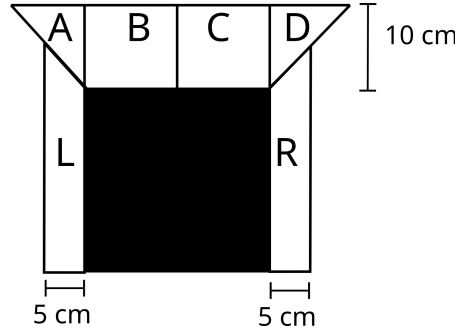


Figure 4.9: This figure depicts the six segments around the robot (black square) used for direct correction of the control input. The driver AI aims to keep the number of LIDAR point equal in the left and right bins. As a result, it keeps a safe distance from objects.

that the number of points in opposing segments are equalized. Moreover, segments dead ahead (B and C in Figure 4.9) of the robot are prioritized.

Note that when there is a large object in front of the robot, the just discussed method is not able to navigate the robot around it. In this case, the robot is stopped by the collision detection system. This system also uses the most recent LIDAR data, to determine if a collision is happening. The method is a simple check for points within a square the size of the robot. If there is a collision, the autopilot stops, and manual intervention is required to get the robot moving again.

In principle, collisions should never happen. However, in practice, there are some cases where they do occur. The most prevalent reason is when lag occurs, causing the robot to overshoot and end up too close to an object. Another cause is when an object is deliberately placed in front of the robot.

#### 4.1.8 Exploration

The next logical step is to make the robot fully autonomous, with the goal to map an entire room or building. To do this, we need a way to generate possible places of interest. Our method does this based on two criteria. The first, and most important method is to find frontiers. These frontiers are points on the map that separate observed and unobserved areas. A clear example is an area observed through a doorway (Figure 4.10). Secondly, after all reachable frontiers have been explored, the history map is used to derive other places of interest. The choice of which history cell to visit next, is based on whether a cell is empty or the age of the local map it is pointing to.

Frontiers are identified as follows. We use flood fill from the position of the robot, and traverse all neighboring cells if they are observed (i.e., hit or miss). Moreover, for each cell, we check if it is classified as a frontier cell. A cell is a frontier cell if it is only a miss cell and has at least one unobserved neighboring cell. All frontier cells are placed in a frontier set, during the flood fill exploitation of the map.

After the creation of the frontier cell set, it is time to cluster these neighboring cells together. Clustering is a matter of doing flood fill over frontier cells only. We start with a randomly chosen frontier cell in the set. Any neighboring frontier cells are added to the cluster. Moreover, we remove all visited frontier cells from the frontier set, to prevent revisiting them. We keep clustering until the frontier set is empty.

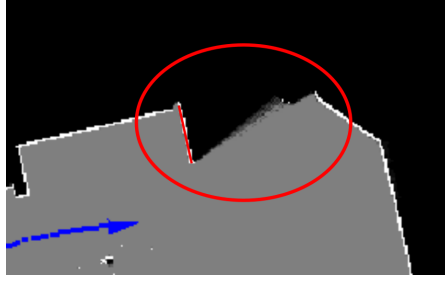


Figure 4.10: This figure shows a frontier. The area behind the door (red line) is not visible to the robot. Frontiers are recognized by the transition from miss cells to unknown area (without a hit cell border).

Any cluster that has more than 20 cells is converted into a (frontier type) point of interest. A point of interest is found by calculating the rectangular bounding box around all cells in the frontier cluster.

The next method to find points of interest is to divide the global map up into smaller (square) regions with a size of 50 cm. These regions are derived from the history map, discussed in the global grid map section. The goal of this exploration step is to visit each reachable history map cell at least once. The idea behind this step is to make the global map more robust and complete by adding the maximal number of local maps to it. Moreover, by visiting each reachable history cell, we make sure that all objects are observed from many directions. Eliminating shadows cast by small objects (e.g., legs of chair), because these shadows are generally too small to be picked up by the frontier search algorithm.

The search for these reachable history cells is once again done by a flood fill based algorithm. The search starts from the robot location. Then it expands the search over miss cells to obtain a set of reachable cells. The next step is to convert the cells into larger squares, that have the size and place of the history map cells. Moreover, the history map is used to check if there is already an observation from this history cell location. For any history cell that is empty, we add a point of interest to the places of interest set.

Finally, we propose to continuously add reachable history cells. By doing this, the robot will never run out of places to visit. Visiting reachable history cells should be done by visiting the oldest ones first. As a result, the (global) map will always stay up to date. This last method of finding places of interest is not completely implemented at the time of writing this text.

After the search for frontiers and unvisited cells, priorities need to be assigned to places of interest. To determine which point of interest needs to be visited first. We first sort these points of interest based on their type (i.e., frontier, reachable cell, revisit cell. From high to low priority). We then further sort them by distance from the robot's position. Finally, the robot visits the first reachable place of interest in the queue (i.e., there exists a route from the robot to the place of interest.).

## Chapter 5

# Experiments

In this section, we describe several experiments and show their results. Unfortunately, using a well-established data set along with our software proved difficult. Because dynSLAM was never build to support general datasets. As a result, a mock-up server would need to be build to stream the data into the client application. Moreover, the dynSLAM is fine tuned for our robot layout. Finding a well known dataset that matches these properties is difficult. Because of this, we choose to create our own datasets. We have created two datasets of static environments. These will be referred to as ‘Hallway’ and ‘Living Room’.

The robot platform we use is the MonsterBorg (see Figure 3.1) in combination with the ThunderBorg motor driver board. The sensors we use are the RPLIDAR A3 and a BNO055 inertial measurement unit (IMU).

A Raspberry PI is responsible for controlling the motors and collecting the sensor data. The Raspberry PI also runs a server that makes it possible to control the robot and retrieve sensor data remotely. WiFi is used to connect the robot to the local network.

### 5.1 Normal Driving

In this section, we drive around in a static environment and show the progression of the map being built. For this test, we use two datasets, obtained at different places. The first dataset is referred to as the hallway the second is referred to as the living room.

This experiment aims to show the SLAM performance of our dynSLAM algorithm in a static environment, under well behaved conditions. That is, the robot was driven around in a smooth manner, and jerky movement are avoided as much as possible. Moreover, the terrain the robot drives is smooth, but there are some bumps in the form of doorsteps and alike. The floor is representative for typical indoor environments.

#### 5.1.1 Hallway Dataset

The hallway dataset consists of a hallway and two rooms. The robot starts in the hallway, explores the two rooms, and then returns to the starting location. The hallway dataset contains 852 LIDAR frames. As a result, the measurement took around 70 seconds. The final map obtained is shown in Figure 5.1.

We annotated the result of Figure 5.1, to highlight some notable features in the map. The red letter A is placed on a seemingly unexplored area. However, in reality, there is a wall as indicated by the red line. This behavior is caused by a mirror placed against the wall. Note that

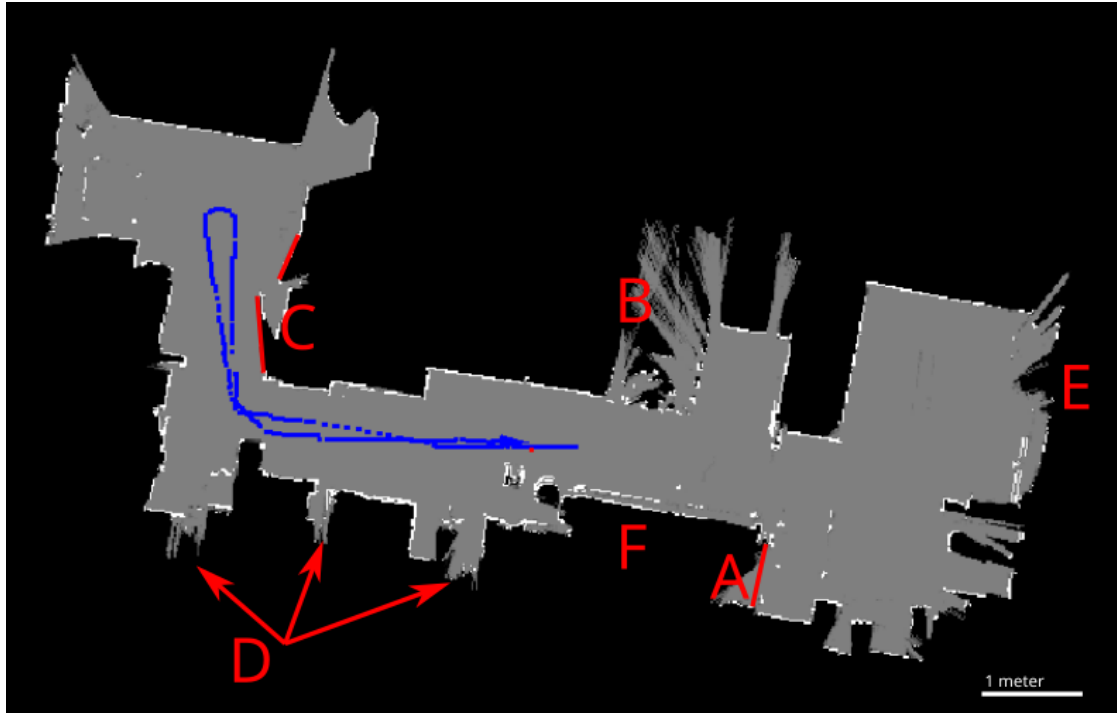


Figure 5.1: This figure shows the final map obtained on the hallway dataset. Note the blue trajectory of the robot is incomplete as only the most recent robot positions are shown. Annotation A shows the effect of a mirror on the map, instead of the wall denoted by the red line. Annotations B and D show objects that partially block the view of the LIDAR. Annotation C, shows two doors, indicated by the red lines. Annotation E shows a wrongly mapped wall, cause by the robot driving too fast over a doorstep when exiting the room. Annotation F shows a wrong scan match, after the robot returns to the same place.

we do not claim to support the mapping of reflective objects. When the ghost region intersects with another existing area, it can easily ruin the SLAM process.

The second annotation in Figure 5.1, is denoted B. The area around B seems like an erroneous observation; however, this is correct. The observation is a fence guarding a staircase. The fence posts show up as small white stripes, indicating a wall or object. Objects in the stairwell, prevents the light rays from reaching the wall on the other side of the stairs. This causes a chaotic look on the map.

Annotation C, here it looks like an incorrectly mapped wall. However, in practice, there are two doors, as indicated by the red lines. The area is mapped correctly. Other points that look like errors are shown by D. These observations are correct and caused by objects on the floor. Some light rays manage to shine in-between two objects.

The wall at E is mapped wrongly in the end result. During the mapping process, the room is first mapped correctly. However, when exiting the room, the robot drives over a doorstep. This tilts LIDAR and causes it to obtain the wrong data. This frame is then inserted into the map and throws the SLAM algorithm off track. Note however that further scan matches are calculated correctly, even after this hiccup. We have discussed the use of IMU roll and pitch data to prevent adding these non-horizontal measurements to the map. In this case, the system was not able to prevent the addition of the frame. To make the system reject these particular frames. However, at some point the threshold needs to be lowered in such a drastic manner that the overall quality of the map start to suffer from lack of LIDAR frames.

Another problem with the map is denoted by F. At first, the wall is mapped correctly. However, when the robot returns, it looks like a wrong loop closure is made, where the wall is placed a few centimeters too low (relative to the bottom of the image). The old wall still slightly persists on the map after the loop closure, resulting in a double wall on the map. The other side, has no double wall. This is because the filter algorithm discussed in earlier sections, removes the outside wall. Moreover, if more scans were taken at this location, the double inside wall will disappear because of the newly obtained data containing empty cells at that location.

The hallway dataset is rather difficult for the SLAM algorithm. The main reason is that there are many items on the floor, cluttering the view of the robot. Moreover, in many locations of the dataset, a slight variation of the measurement height causes a large change in measured distance. For example, a radiator that is mounted slightly below the normal measurement plane of the LIDAR. When the robot makes a small dip forwards, the light rays shine under the radiator. This causes the measured distance to increase by the thickness of the radiator because we now observe the wall behind it.

### 5.1.2 Livingroom

Figure 5.2 shows the final map obtained from the Livingroom dataset. The dataset contains 1450 LIDAR frames and has a capture time of 120 seconds. The dataset contains one large room, and two smaller rooms attached to each other. The robot starts and stop in the large room. The leftmost portion of the map was unreachable by the robot, due to problems with WiFi reception in that area.

We once more annotated the map, however, because in this data set the floor is less cluttered, the end result is cleaner compared to the Hallway dataset. As a result, there are fewer annotations.

The first annotation, A, shows two shadows on the map. This is because the robot was not able to look at these two objects from the top left. In the two boxed shaped areas, some fill-in can be observed. Obviously, this should not have happened. However, because there is no fully connected line around the object, the cleanup algorithm is not able to remove the occupied space

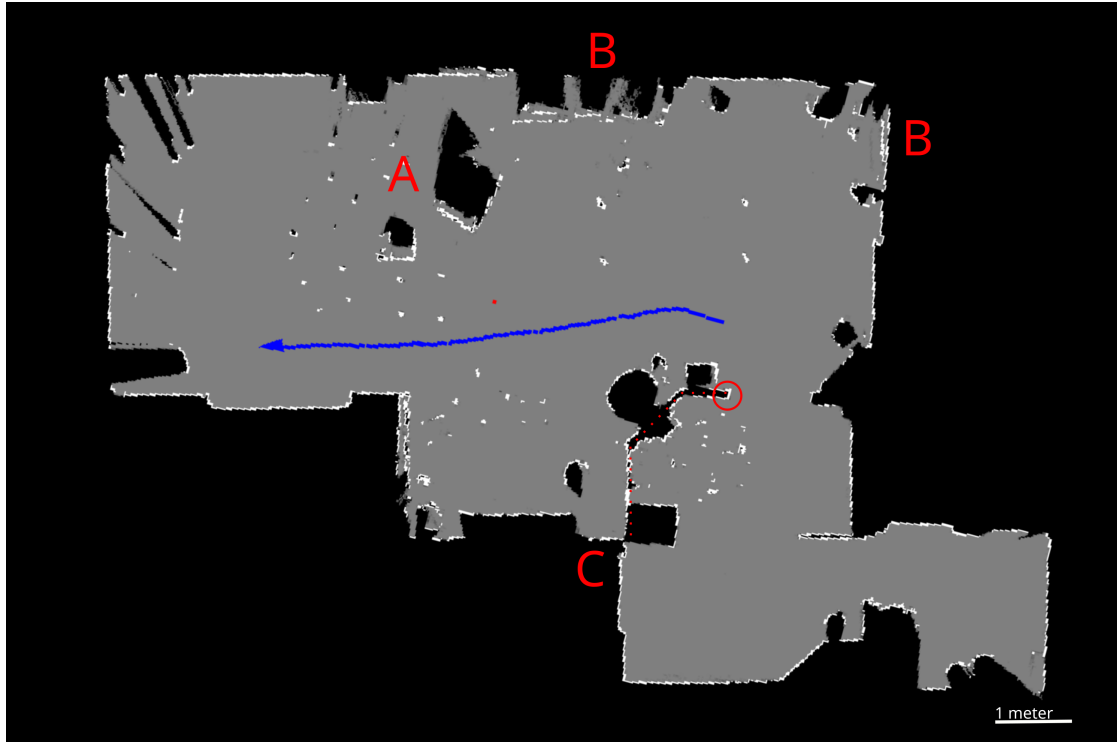


Figure 5.2: This figure shows the final map obtained on the Livingroom dataset. Note the blue trajectory of the robot is incomplete as only the most recent robot positions are shown. Annotation A shows two rectangle shaped objects that are only viewed from one side. Annotation B shows to locations where multiple objects are mapped at different heights, resulting in a chaotic look on the map. Annotation C show a mapping error where a wall collapses into a single line.



drawn within the objects. This is because there may still be a path ‘into’ the object as far as the algorithm is concerned.

The area denoted by the B’s shows distortions. The reason is that there are many distance variations when the measurement height changes slightly. This is caused by two radiators and an open closet. Just as in the previous hallway dataset, such a situation proves to be difficult for dynSLAM. Especially when the measurement was taken over a large distance, because a small variation in pitch or roll can cause the measurement of an other object.

Finally, annotation C shows two consecutive rooms, separated by a fairly long wall. This kind of situation is difficult for a SLAM algorithm. There is a chance that the two sides of the wall will overlap when small matching errors accumulate. And in fact, this also happened in our case. In the beginning, the wall is mapped with the correct size, as indicated by the red circle. The red dotted line shows the continuation of the wall. Eventually though, the front and back of the wall collapsed into a single line.

The wall collapsing problem is not trivially solved apart from improving the accuracy of the scan matches (both scan to map and loop closure scan matches). In the case that there is an actual loop in the map, loop closure can help solve this problem. As the pose graph gains a circular dependency (cf. in our case, it is a treelike structure, with only very small loops).

Moreover, when the two walls do collapse to a single line, the problem becomes even more difficult to restore later in the process. This is because the loop closure constraint scanner now matches a new local map to both the correct wall and the wrong one at the same time. As a result wrong loop closure constraints are added to the graph, forcing also the placement of new local maps in the wrong location.

## 5.2 Map Building

In this section, we aim to show the behavior of the map building process. We start by showing the different map stages with small steps 30 LIDAR frame steps, while the robot is driven around. This experiment aims to show how new data is incorporated into the (global) map.

In the following subsection, we show the effects of a loop closure step and floodfill cleanup step. The situation we have chosen a situation where the robot just regains vision on an already observed area. We show the map before and after loop closure and floodfill clean up.

### 5.2.1 Small Increments

In this section, we show the process of map building. In particular, we show how mapping looks (see Figure 5.3) like on the Livingroom dataset by showing intermediate maps for every 30 LIDAR frames added. The first sub-image shows the map after 70 LIDAR frames, because the robot is stationary before this point. After 70 LIDAR frames have been obtained, the robot starts to move through the environment, as depicted by the blue trail. The current robot position is indicated by the blue triangle.

While most of the mapping progress shown in Figure 5.3 is rather straightforward, the transition from frame 130 to 160 shows some noteworthy behavior. It can be seen that areas around the robot that were previously marked as empty space, are now back to the unknown state. This is a result of how local maps are stored. As noted in the Design chapter, we make use of a history map to create a set of active local maps. Because local maps are created every 50 LIDAR frames, a newly obtained sub-map can fall into the same history cell. When this happens, it overrides the previous local map in the history cell. If the new map contains less detail, this will show up as in the last map in Figure 5.3.

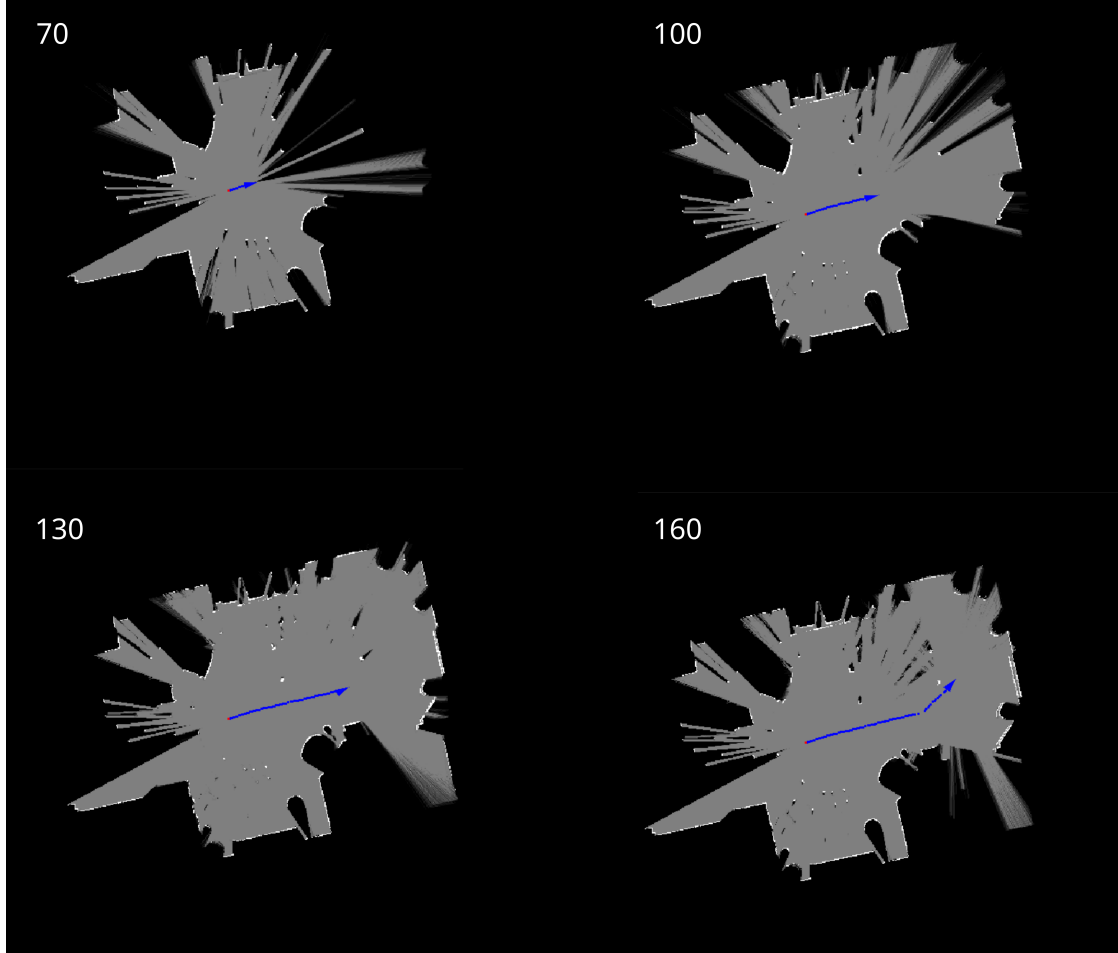


Figure 5.3: This figure shows the beginning of the creation of the Livingroom dataset map. Each separate map shows the advancement made after adding 30 LIDAR frames. The white text denotes the total number of LIDAR frames added to the map so far. The blue trail and triangle show the path taken by the robot and current position of the robot.

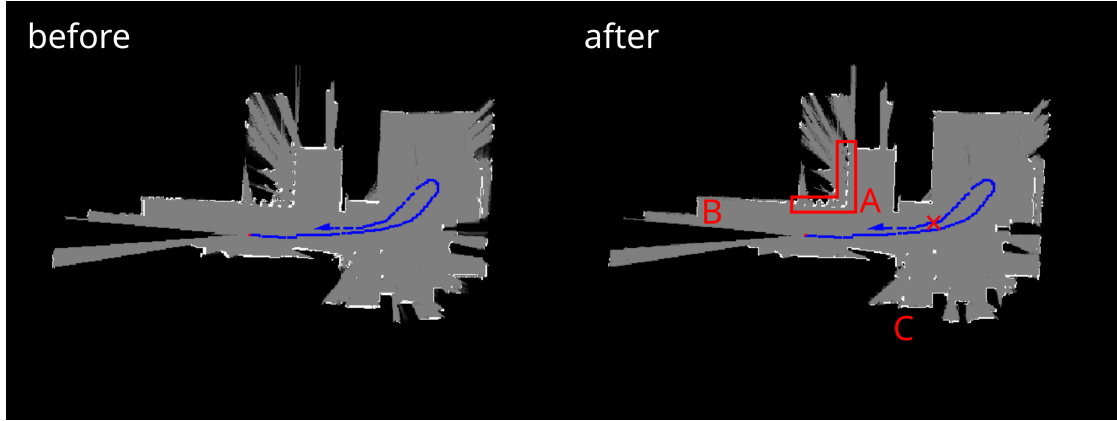


Figure 5.4: This figure shows the effects of the loop closing step and the flood fill cleanup pass. The left map shows a map before loop closing and the flood fill cleanup pass. The right map shows the same map but after these loop closing and flood fill cleanup. The red cross indicates the starting point of the most recent created local map. The blue path shows the trail of the robot. The blue triangle shows the current robot position. Annotations A and B, show how loop closing can recover fine details. Annotation C, shows how the cleanup pass removes thick lines around objects and wall.

The problem just discussed, shows up on more occasions throughout the experiments. It is most notable when the ray length is limited because less data is included. In our experiments, we use a 4 meter ray length, which is short compared to traditional work. The reason for choosing a short maximal ray length is that they are affected less by the pitching and tilting motions of the robot. In our Future Work section, we propose a solution that prevents parts of the map disappearing if a local map is replaced.

### 5.2.2 Loop Closure and Cleanup-pass

In this subsection, we explore the loop closing and flood fill cleanup pass, discussed in the Design section. We use the Hallway dataset, and analyze the map around LIDAR frame 500. Figure 5.4 shows the result of the loop-closing and flood fill cleanup pass.

We choose to analyze this location and time, because the robot does a loop closing observation at this step. Moreover, the chosen situation benefits from loop closing, because the robot reenters a location that has been observed before. Because the robot was previously visiting another room, it did not have a visual on the hallway. Hence, there is a possibility that measurement errors accumulate, resulting in a misaligned placement of a local map on reentry.

The local map frame starts at the red cross and ends at the current robot position. Because the robot has just traveled around in a loop inside of the rightmost room, the view of the area indicated by A is blocked. The frame created between the red cross and the current robot position, reobserves the area indicated by A.

In Figure 5.4, annotation A denotes the area that benefits from loop closing. In the before map, we see duplicated bars inside the L-shaped red marking. The map after the loop closing and clean up step shows that these misalignment errors are significantly reduced. The bars after loop closing and clean up pass now consist of a single rectangular area, which is true to the bars in reality.

Also note, that the two bars in the corner are nicely placed perpendicular to each other. This

is an especially good result, as complex-shaped items tend to converge into a large blob. Once this has happened, it is a difficult situation to get out of, because the scan matcher is given a large amount of freedom.

Because we only use a set of active (or most recent) maps, the our SLAM algorithm may resolve the problem. The reason is that we discard old results, hence also the errors made in the previous maps. Note that the errors need to be small compared to large scale structures, for this type of correction to work. Large scale errors will manifest throughout the loop closure constraint graph, and thus remain present.

Another example of the loss of detail problem described above, can also be seen in Figure 5.4. For example, looking up from the red cross a closet leg is shown. After loop closing, there is an empty area in this leg, this is correct because it is impossible to observe the inside of the leg. However, When looking down, can see what can see the algorithm fail. Here, there are 4 chair legs arranged in a square. The inside should be empty/undefined as well however, this is not the case.

Another interesting result shown from loop closing is denoted by annotation B in Figure 5.4. From the before and after, it is evident that the double walling is resolved by loop closing. However, the wall is further away then the robot should be able to observe, because we limit the maximal length of a ray that can be observed. The double wall by B is resolved by the graph optimization process that tries to optimize the total error between local maps.

Finally, annotation C, shows how the cleanup pass removes large hit cell borders from the map. This is important because it gives the scan matcher less freedom over how new scans are placed on the existing map. Hence, improving the SLAM algorithm performance.

In conclusion, we show that loop closure correctly removes accumulated errors after we re-observe a known location. Moreover, the result shows that after loop closure, small details in the map are recovered. Moreover, the final cleanup pass also removes thick lines around walls and objects. Comparing the before and after map, shows that these steps clean up the map significantly.

### 5.3 Distance Accuracy

In this section, we evaluate the accuracy of the distances measured. We use a small-scale object and a large-scale object as a reference. The small-scale object is a cardboard box we placed on the floor. The large scale object is a table, we use the distance between the legs to compare it to the real world distance. The Accuracy of the distances is limited by the cell size of the grid map. In our case, this size is 2 cm.

Figure 5.5, shows the map obtained by driving around a box with dimensions of 33 cm by 42 cm. We rotated the map such that the sides of the box are horizontal and vertical. The rotation angle required was  $45^\circ$ . Moreover, The map that is shown here, is taken directly after a loop-closure step and flood fill cleanup step.

Because it is difficult to determine the exact size of the box from the map, we choose to fit two boxes at both extremes. The small red box in Figure 5.5, has dimensions 28 cm by 35 cm. These dimensions are substantially smaller than the actual ones. As a result, the small box is around 15% too small. Or, in absolute terms, the dimensions are 5 cm and 7 cm off.

The large red box in Figure 5.5, has dimensions of 34 cm and 42 cm. This measurement almost exactly agrees with the expected value. The 1 cm deviation is smaller than the grid size, and thus an acceptable error. Because the cardboard box is a small object, an off by one grid cell error translates into a relatively large error.

To explore how errors manifest in the larger object, we look at the distance between two table

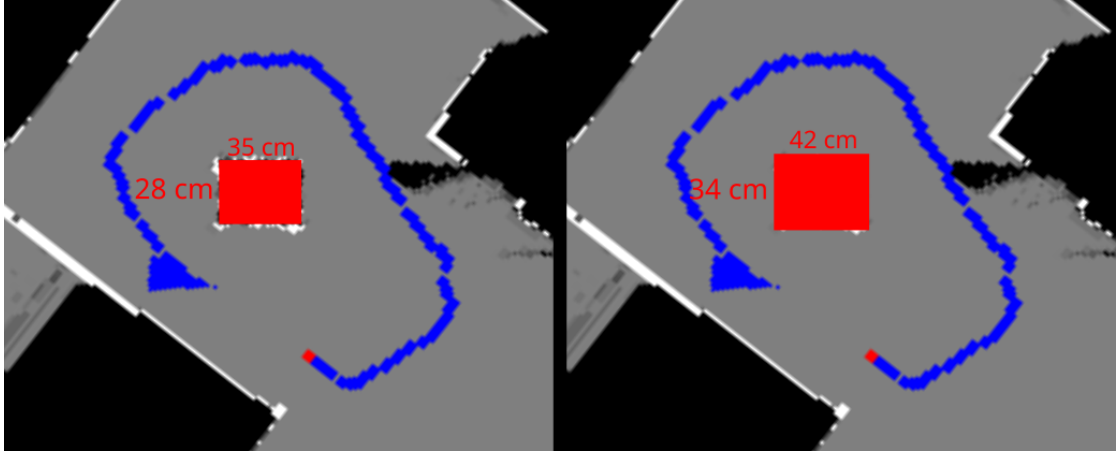


Figure 5.5: This figure shows a map obtained by driving around a cardboard box, as indicated by the blue trail. Because it is difficult to determine the exact transition between empty space and the box, we have drawn two boxes. The red box left shows the minimal size possible size derived from the map. Right the maximal size derived from the map. The actual dimensions of the box are 33 cm by 42 cm

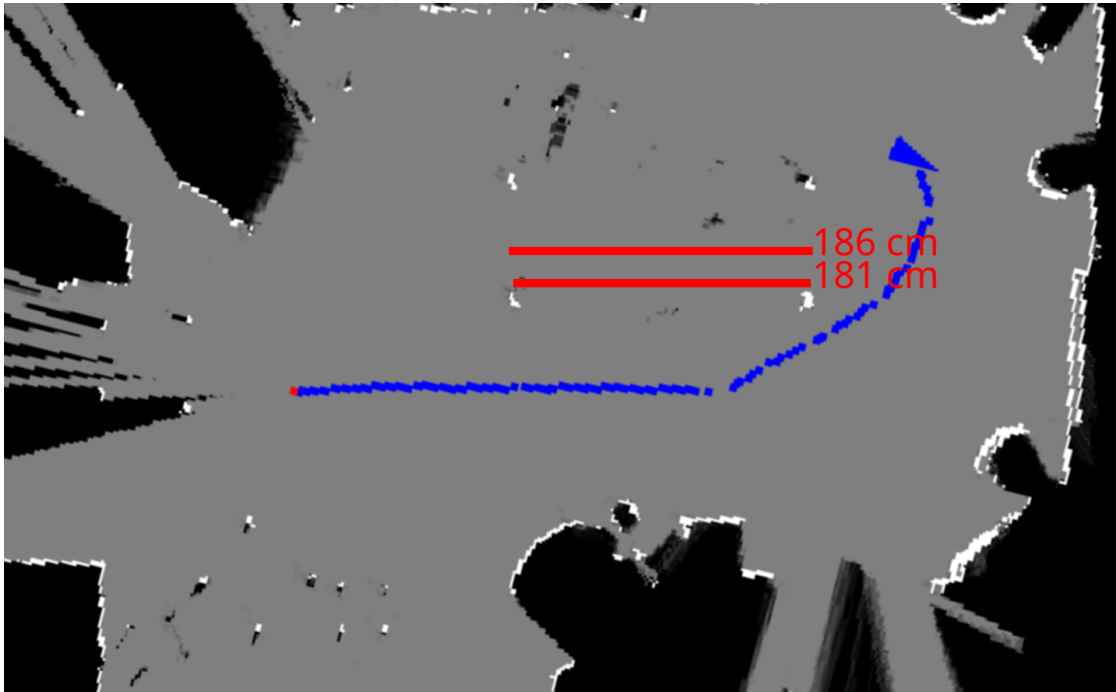


Figure 5.6: This figure shows a map obtained by driving past a table, as indicated by the blue trail. We annotated the map with two bars, representing the maximal and minimal determined distance between the outside of the legs. In reality, the distance is 187 cm.

legs. We use the table in the hallway dataset. Figure 5.6 shows the table in the map, we measure the distance between outside of the two lower table legs. Because it is difficult to measure and determine the exact location of the legs in the map, we have drawn two bars. The upper bar denotes the longest distance, where we feel confident that this is the maximal distance measured between the two legs. The lower bar denotes the minimal distance between the legs. The length of the upper bar is 186 cm and the length of the lower bar is 181 cm. The longer bar is 1 cm short, resulting in an error of less than 1%. The shorter bar is 6 cm short, resulting in a 3% error.

Notably, the bar measuring the longest distance between the legs is also shorter than the expected real value. Since the difference is so small and less than the grid cell size, it is difficult to pinpoint a cause. The error in the short bar can not entirely be explained by the resolution of the cell size alone. The maximal error obtained by this resolution is 4 cm, one cell size on each side. However, the absolute difference is bigger. Once again, just as with the cardboard box, it is difficult to find an exact cause for this error.

In conclusion, it is difficult to determine the exact border between free space and an object. Because of this, we measure the objects in two ways. One where we can confidently say the object is no smaller and one where the object is not bigger. From the results, we see that the biggest measurements are close to the actual size of the objects. The measurements denoting the minimal size, are typically two or three grid cell sizes off.

This result is also to be expected from the way our algorithm works. Because misplaced hits on the outside of an object are almost immediately corrected by the next measurement. Erroneous points placed inside of an object are less easily corrected, because they can not be observed. Hence, they need to be removed by reasoning about the possibility whether a measurement is possible. This task is done by the flood fill cleanup algorithm. However, it seems that this yields less exact results.

## 5.4 Dynamic Environment

To show how the map changes when the environment changes during the mapping process, we did two experiments. In the first experiment, someone walks past the robot. This experiment was chosen because people moving past the robot when it is active is a common occurrence in real life scenarios.

The second experiment is to open and close a door during a SLAM session. In this experiment, a large portion of the map suddenly disappears and reappears.

### 5.4.1 Walking in the Vicinity of the Robot

Figure 5.7, shows a sequence of maps, with a person walking past the robot. The person's position is annotated with a red circle. From the timestamps in the figure, it is evident that the walk took around 4 seconds (discarding the starting and stopping map where the person was standing still). The distance walked is around 3.5 meters. This gives an average walking speed of 3 km/h.

Even though the person is walking at moderate speed in a SLAM application, the map shows nicely defined dots representing two legs. Apart from the 25.7 s mark, there are little to no stripes on the maps. This is a result of our aggressive cell updating rules (equations 4.4 and 4.5). From these equations, it is clear that a cell can be fully defined in 10 LIDAR frames. Had we used lower update values, one-off errors would have less influence on the mapping process. The choice is a trade between response to dynamics and robustness.

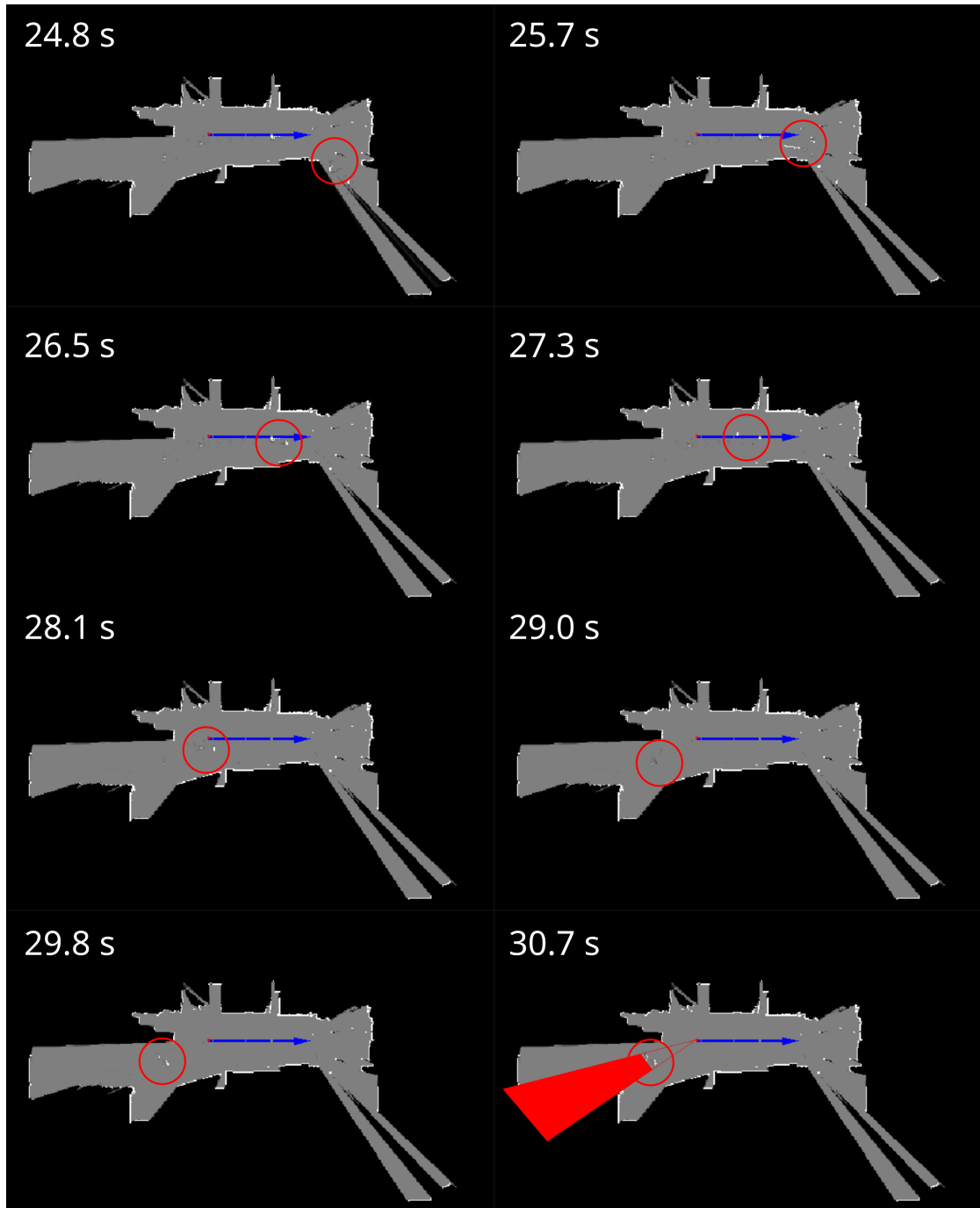


Figure 5.7: This figure shows consecutive maps separated around 0.7 seconds apart. In the sequence of maps, someone is walking past the robot. The red circles indicate the position of the person. The white numbers in the top left corner indicate the timestamps of the map. The red cone shows a ‘shadow’ cast by the person standing, any changes that occur can not be observed by the robot anymore. Before the start of the experiment, the robot was driven a small distance, as indicated by the blue trail of the robot.

The map at the 30.7 seconds mark, shows the person standing in a stationary position. This map shows an error that dynSLAM solution makes when drawing newly appeared objects. Because the environment is already known before the new object appears and the way cells are updated, only the front view is drawn in the map. To obtain the correct shape of the new object, the robot needs to drive around it. This ensures views from all sides are included in the map.

Moreover, we have annotated a red cone on the last map. Everything that changes in the red area, can no longer be seen in by the robot, hence it does not receive updates. Correctly handling this error is difficult, to say the least. One possible way is to detect new objects and remove everything in its ‘shadow’ from the perspective of the robot. The downside of this approach is that it quickly removes too much data from the map. As there is not a good way of knowing how long the shadow should be. As a result, it may even go through walls and remove parts of an entirely different room.

### 5.4.2 Opening and Closing a Door

In this subsection, we explore how our SLAM implementation handles opening and closing a door. Figure 5.8 shows the results of this experiment. The position of the door is denoted by a red circle on the maps. As shown in the map at timestamp 30.3s, the robot starts in a room and drives through the door. We then drive a small section through the other room to map it correctly.

In figure 5.8 at timestamp 36.0s, we annotated the location where the door is closed. In this map, the door that appeared open in the previous map is now closing. The process of closing the door leaves a fan out of partially observed doors. It is worth noting that fan out crosses a local sub-map boundary. This resulted in part of the fanout being updated already and the other part is still present for some time. The closed position of the door is faintly starting to show up. The map at timestamp 40.0s shows the fully closed door after the update process has stabilized.

The bottom row of figure 5.8 shows the act of opening the door again. Before we reopen the door, we drive the robot back and forwards for a couple of meters, as shown by the blue path. Just as the closing step, the reopened door is quickly incorporated into the map. The only error made in the mapping process is that only the front view (as seen from the robot) is drawn on the map (i.e., the door has no thickness). This is because the only one side of the door is observed. We already discussed the difficulties associated with this phenomenon in the previous subsection.

When closing a door or obscuring vision in some way, there are two possibilities of how the changes propagate through the map. Either the closed-off area remains present on the map or it is removed over time. The difference originates from whether there is a local map present in the closed-off area. In the case that there is no local map in the closed-off area, the only knowledge about area is saved in local maps outside. When these local maps are replaced, the closed-off area is reverted into unknown data (i.e., black). In the future work, we discuss a method that will always preserve the closed-off area. This method, stores local sub-maps based on their coverage, rather than starting location.

Because we started the robot in the room that is closed-off when the door is shut, there is a local map in the room. As expected, the room does not disappear when we close the door and drive the robot to update the local maps.

## 5.5 Route Planning

In this section, we show a path generated by our route generation algorithm, see Figure 5.9. We used the living room dataset because it is the largest dataset we have captured. Moreover, it contains multiple rooms and has a reasonable amount of ground clutter.



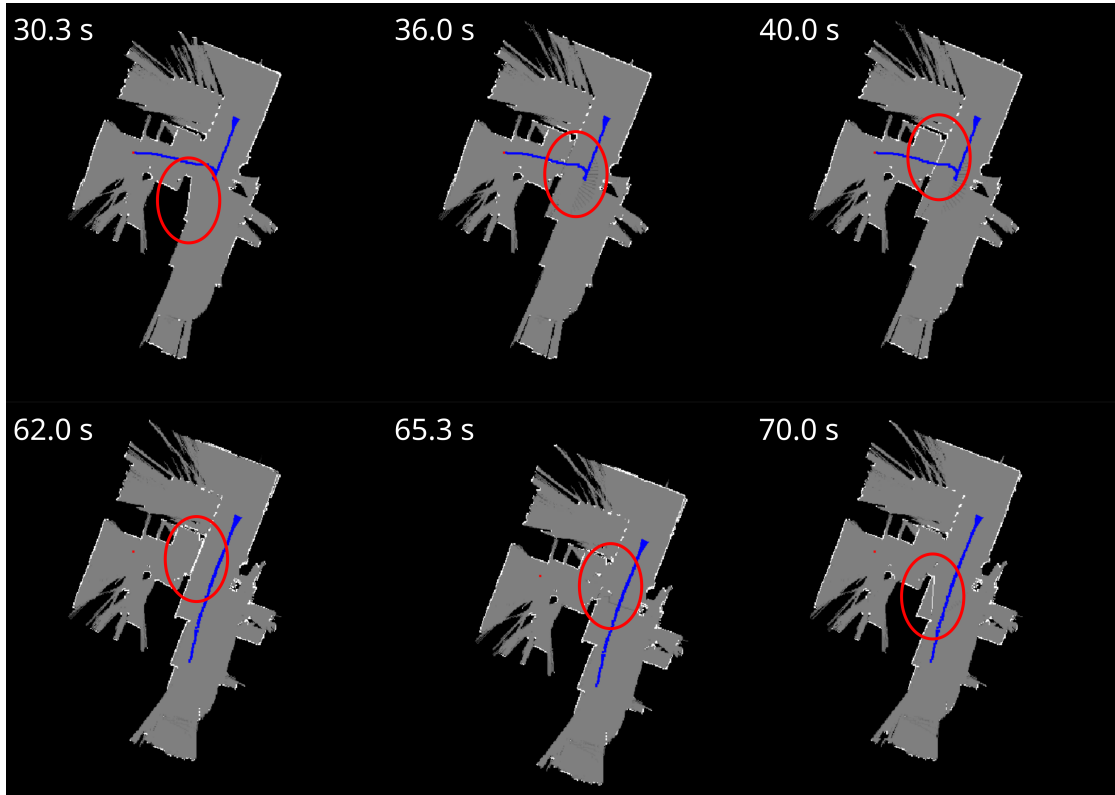


Figure 5.8: This figure shows the effects of opening and closing a door to a room. The robot initially starts in the room that is later closed-off. The blue trail shows where the robot was driven to map the environment. At the 36.0s timestamp, we close the door, the robot is then driven through the hallway and back, to update its local maps. Finally, at the 65.3s timestamp, the door is reopened.

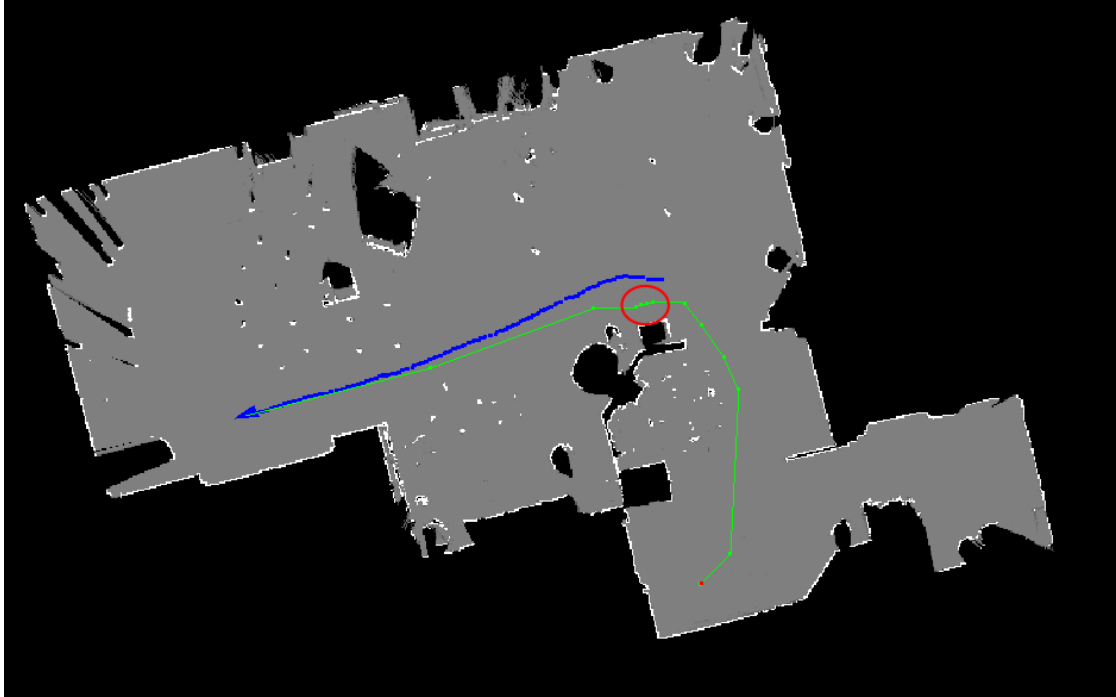


Figure 5.9: This figure shows a route generated by our method, on the living room dataset. The blue trail shows the path taken by the robot to map the environment. Note that the blue path is not complete, as only as it only shows the most recent robot locations. The green points show the waypoints on the path, the green lines indicated the predicted path the robot takes to get to the next waypoint. The path starts at the robot position, and ends in at the red dot. The total length of the path is 10.4m.

The path starts at the robot’s current robot position (i.e., the blue arrow), and ends at the red dot. We measured the route to be 10.4 meters. Moreover, the route consists of 12 segments. The waypoints on the route are indicated by the green dots.

The generated route, for the most part, is nicely laid out with enough clearance to walls and objects. In Figure 5.9 we annotated a part of the route with a red circle. In this part of the route, waypoints are placed close together. Placing waypoints close together, sometimes poses a problem for the autopilot. This comes from the fact that the direction vector (i.e., the difference between the next waypoint and the robot position) is ill-defined when its magnitude is comparable to the cell size. This causes the angle of the vector to change drastically for small movements of the robot. This results in jerky movement of the robot, when traversing this part of the route.

During development and testing, we noticed that waypoints are often placed close together when there is a curve in the route placed close to some object (just as in the annotation in Figure 5.9). The reason for this behavior is that the dense path generated by A\* places the path as close to the object as is allowed. Pruning the waypoints in the curve is then not possible, because it would violate the clearance constraint.

One possible solution is to make the clearance constraint for A\* slightly larger compared to the clearance constraint when pruning waypoints. The downside to this approach is that the robot requires more space to manoeuvre through the environment.

## 5.6 Frontier Finding

In this section, we discuss the performance of the frontier finding algorithm. The exploration part of dynSLAM is underdeveloped due to time limitations. However, we can still show the result of the frontier search algorithm. Figure 5.10 depicts the frontiers found on the partially mapped living room dataset.

Figure 5.10 shows the frontiers found on a partially explored living room dataset. Each frontier is indicated by a green box. The green box is the bounding box around a frontier. Small frontier areas are filtered, hence not all frontier cells are included in a frontier. It is clear that all of the frontiers found, will provide new insight of the environment, if an observation is done in the vicinity of a frontier.

Looking at Figure 5.10, we see that a route (green path) is planned to a frontier. This frontier was chosen because it is the closest reachable frontier. The destination is the center of the frontier, indicated by the (unconnected) green dot. The actual route is planned to a destination close by. This happens because when the AI searches for a route, cells in the vicinity of the destination are also accepted. As a result, routes can be planned to destinations that are otherwise considered unreachable due to clearance constraints. This method proves especially useful when planning a route to a frontier because frontiers are typically found in constricted areas. However, visiting the frontier close by is typically good enough to remove it from the map.

A problem that can be seen in Figure 5.10, frontiers are sometimes fairly small. An example is indicated with a red circle in the figure. Currently, the minimal size of a frontier is determined by the minimal number of frontier cells that need to be in a cluster. However, in practice this constraint seems to be a little too relaxed. A simple fix for this problem is to also require a minimal area and/or minimal side lengths.

Finally, there are overlapping frontiers in Figure 5.10. One simple solution is to merge overlapping frontiers into a single larger frontier. This can be done by substituting the overlapping bounding boxes with a single bounding box that encompasses the bounding boxes it replaces. However, one problem with this approach is that frontiers grow so big, that they do not represent



Figure 5.10: This figure shows frontiers on a partially mapped living room. The green boxes are bounding boxes denoting places of interest. The blue trail shows path taken by the robot to map the room. The green lines, show a path to a frontier. This selected frontier is chosen to be visited next by the autopilot, it is clear that an observation at the location of this frontier, yields new insight of the environment.

a single point of interest anymore.

In conclusion, generating frontier bounding boxes is a simple problem. Filtering and merging them in a useful way is a more difficult task. Our method does work, however for practical use, it still needs further fine tuning and performance (as in speed) improvements.

## Chapter 6

# Future Improvements

In this chapter, we discuss the limitations of dynSLAM. Most of these problems were either discovered during testing, or we were unable to implement a solution due to time limitations. The list of improvements is vast and too large to fully discuss in this section. Instead, we only discuss improvements that are currently holding back our SLAM solution. The order in which we discuss the topics is the order of importance, with the most pressing topics being presented first.

Moreover, we specifically do not discuss optimization problems. There is a large number of optimization improvements that can be made in our software, both in memory usage and speed. However, we never investigated the performance of our application. Moreover, we never made any claims towards these properties. Instead, we focussed on other metrics of our SLAM solution (such as accuracy, quality of the map, etc.).

### 6.1 Local Sub-map Storage

Currently, local maps are tracked by placing a reference to them in the history map. This map stores pointers to the most recent local map at that position. The cell size of the history map must be much larger than the cell size of the local maps. In our case, the history cell size is 50 cm and the normal cell grid map cell size is 2 cm.

The criteria for adding a pointer in the history map is simply that the local map has its origin at that location. However, during our experiments, we noticed a problem with this approach. When the robot is moving through a new location in a single line. The history map pointer is often replaced in the same history cell. However, because the robot is moving, it happens quite often that some part of the environment is obscured and not visible anymore. However, in the previous local map, this data was present. When the new local map is added, this obscured part disappears from the map.

To solve this problem, we suggest the following method. Instead of only saving local maps at their origin in the history map, save them in multiple history cells. The criteria for storing/replacing a history map pointer is that the local map covers a significant area. For example, store a pointer to a local map in a history map cell, when it fills that cell with more than 75%.

Another advantage of this approach is that it quickly fills the history map. Making the generated global map more robust. Currently, the filled cells of the history map correspond to the path the robot has traveled (because the origin of a local map is the robot's position when it is created). Hence, in many cases, the history map remains sparsely filled.

## 6.2 Ray Length Filter

We have discussed the problem caused by adding points far from the LIDAR. In essence, the longer the light ray, the larger pitch, and roll deflection errors become. As a result, there is a trade-off between visible distance and accuracy.

In our current implementation, any LIDAR rays longer than the set threshold (4 meter) are simply removed. The downside of this approach is that it leaves holes when the robot looks at an object far away.

Our proposed solution is to only add the empty cells up until the point where the ray reaches its max length. Note that this is different from reducing the length of the ray to the threshold value, since that would also add a hit cell at the end.

## 6.3 Networking

The network stack we currently use tends to behave inadequately when the WiFi connection is bad. This behavior results in stuttering and sometimes leaves the robot uncontrollable for considerable time. The reason is that TCP is used as a communication protocol over WiFi. Moreover, all data is packed into the same network stream. As a result, when the connection hangs for some time, the video buffer fills up quickly. When the connection is restored, video data is sent all at once to the client, inhibiting all other control commands.

Another problem with our current implementation is that it can theoretically still block on network calls. This happens because, as soon as there is network data available, the applications (client and server) expect to be able to read at least one full network message. However, this is not guaranteed. In practice though, this never actually poses a problem as far as we can tell.

The solution to the video buffer filling up and the possibility of blocking IO. Is to make use of both UDP and TCP connections. Important messages, such as control and sensor data are still sent using the TCP connection. Low priority data, in our case the video stream is sent using UDP. The advantage is that, if the connection is lost the video packets are simply lost instead of retransmitted.

It should not be a problem when video data is lost or corrupted during transmission. We found the video decoding library we use can handle data corruption adequately. In cases where the video connection is completely lost, the client can simply request the server to restart its video stream. This request is then sent using the TCP protocol, to make sure it is delivered.

To solve the possibility of a network call blocking the main loop, we propose the following. As of now, the `select` function call is used to find out if a `recv` call will block or not. The program expects to receive an entire network message. However, it is possible that the network packet is not available in its entirety. Hence, blocking the main loop. By using non-blocking IO and a receive message buffer, the process of receiving a network packet can be split over multiple main loop iterations.

The suggested network improvements do not influence the outcome of the SLAM algorithm. However, it does make the process of capturing data smoothly, because there are fewer hiccups. Moreover, networking problems only occur when the WiFi connection is bad. If this is not the case, the current networking solution functions adequately.

## 6.4 3D Egomotion and Height Correction

Our current egomotion correction technique acts on 2D data. The point cloud obtained from the LIDAR contains solely 2D points. Correction of this data is done using only the yaw component

of the IMU. This works on the assumption that the ground and the LIDAR observations are parallel. This assumption holds as long as the robot does not tilt or roll, or at least only by a small amount.

In practice, we notice some errors due to light rays not measuring at a constant height. The behavior can be improved by taking the full IMU orientation vector into account. Linear interpolation between orientation vectors is possible by representing them as quaternions. Interpolating the quaternion makes it possible to convert the LIDAR datum points from 2D into 3D space. By settings a maximal deviation from the  $z = 0$  plane, it is possible to tell if we need to remove a point from the point cloud. Finally, the 3D points are projected back onto the  $z = 0$  plane to obtain the 2D datum points again.

Ideally, the 3D corrections and the extended ray length filter are combined. In this case, we do not remove the entire ray from the LIDAR frame. Rather, its length is reduced to the point where the ray intersects the upper or lower  $z$  threshold. When inserting this point into the (local) map, no hit cell is inserted at the end.

## 6.5 AI Driver Improvements

Currently, the AI driver is either rotating or driving in a straight line, while it is active. It would be better if small-angle corrections are made during driving. For large angle differences (e.g.,  $|\theta| > \pi/2$ ), stopping and rotating is still a better option. Another interesting idea to make the AI drive even more smooth, is to fit a curve (e.g., the Bezier curve) to the waypoints. This way, the heading of the robot is constantly adjusted (cf. our current solution only changes the robot heading at waypoint locations).

We already discussed the problems associated with the robot overshooting a waypoint. Our current solution is to simply recalculate the route now and then. However, it would be better if there was a robust way to select the next waypoint in front of the robot. The first solution that comes to mind is simply selecting the waypoint which minimizes total path length  $l$ , as in Equation 6.1.

$$\arg \min_i l = |r - p_i| + \sum_{n=i}^{N-1} |p_{n+1} - p_n| \quad (6.1)$$

Where  $l$  is the total path length from the robot to the destination.  $i$  is the waypoint to visit next.  $r$  is the current robot position.  $p_i$  is the  $i$ -th waypoint.

However, this method for finding the next waypoint has a problem. The problem arises when the next waypoint found is obscured by an object. In this case, the AI driver would try to drive into the object and come to a stop. A solution to this problem, is to also check whether the line between the robot and the next waypoint violates the clearance constraints.

Currently, the AI driver does not have speed control. Instead, it blindly varies the PWM duty cycle of the motor wheel. However, this poses problems on ground types with high friction (e.g., carpet). In this case, the robot may get stuck during turning manures because the duty cycle is too low. Simply increasing the value is not an option, because this causes the robot to turn too fast on smooth ground. Moreover, the state of the battery charge also greatly affects the duty cycle and speed relationship.

Two possible solutions come to mind. The first one is the use of a feedback loop (e.g., PID controller) and the speed obtained from the SLAM process. However, there are some hurdles to overcome, mostly due to time delays. First, there is the time delay due to SLAM being slow because of the 12 Hz LIDAR. Moreover, due to the relatively large cell size, the time derivative of the position function is not smooth at slow speeds. A smoothing filter solves this but also introduces additional delay.



The second method is to explore the relationship between the PWM duty cycle and robot speed. By isolating the relevant factors, presumably battery voltage, and ground friction coefficients, a mathematical model can be constructed. Based on the previous behavior of the robot, the friction coefficients can be calculated and substituted in the model. Then, based on the assumption that the relevant model parameters are not changing rapidly. The model can be used to directly map a speed to a PWM duty cycle.

Alternatively, use motors equipped with encoders and obtain a speed that way.

## Chapter 7

# Conclusion

In this thesis, we presented dynSLAM our SLAM implementation. dynSLAM is split into two applications. A server running on the robot platform and a client running on a pc. The server is used as a dumb device and the client does all the work.

We show that our grid map based SLAM solution is capable of mapping dynamic environments. The method used to update the map when changes occur, can add and also remove points to the grid map. Moreover, only the most recent local maps are used to stitch a global map together. To find the most recent local maps, we use a history map. This history map contains pointers to local maps and has a cell size of 50 cm. The pointer in this map is updated when new local maps are obtained at that location.

In our SLAM solution, we force robustness by first filtering the incoming raw data. This step removes garbage data before it propagates through the SLAM system. IMU data is used to correct the LIDAR data for the movement of the robot platform during measurements. This step ensures that LIDAR scans are consistent between measurements.

Moreover, we correct measurement errors behind objects and reduce the hit cell border. This step is done using our (to our knowledge novel) cleanup pass. The advantage is cleaner-looking maps, but most importantly, it also reduces the freedom given to the scan matcher (i.e., matching against a thin line versus a thick line. The latter one having much more freedom in the matching process).

We also discuss the process of pathfinding and following. Moreover, we discussed anonymous exploration using frontiers and history cells. However, the implementation in its current state leaves room for improvement.

In the experimental section of this text, we show the performance of the SLAM algorithm, the route planner, and the frontier finding algorithm. The experiments show that our solution is indeed capable of mapping both static as well as dynamic environments. The dynamic environments consist of: a person walking by and a door being closed and reopened. Essentially, a room is added or removed as far as the SLAM algorithm is concerned, when opening and closing doors.

Unfortunately, we were unable to make a sensible comparison of dynSLAM and Cartographer [18], the current state of the art in the field. We did attempt to run Cartographer on our own datasets, however this proved more difficult than expected, because of Cartographer is not really suited to consume the data dynSLAM expects. This would have resulted in an unfair comparison for Cartographer, and we feel like the comparison would not have been scientific. Hence, it is also not included in the text.

We did however experiment with Cartographer on the publicly available datasets. Moreover,

the authors also publish a paper that contains experimental result of Cartographer. When we compare these results to our dynSLAM, we feel that dynSLAM has the potential to be competitive in the field. Especially the quality and accuracy of our maps seem comparable. Areas where dynSLAM lacks most are: performance (as in speed of the algorithm) and memory consumption and the limited maximal size of the global map.

The current state of development is as follows. SLAM, path planning, and path following can be considered working. However, improvements to performance and the suggested adjustments from the *Future Work* section can still be made. The state of the anonymous exploration implementation is currently unfinished and can only be considered a proof of concept. We do propose a possible implementation. However, at the time of writing, we did not have enough time to implement and test it ourselves.

As mentioned in the above paragraph, the performance (i.e., speed of the algorithm) of the SLAM algorithm can still be improved. This work did not focus specifically on creating a high-performance SLAM algorithm. As a result, the client currently, runs stable on a modest desktop PC. However, the client on embedded hardware (e.g., a raspberry PI) proves troublesome, due to memory and speed limitations. One area where optimization is possible is our scan matcher. This can be optimized, for example, with a similar approach as the branch and bound technique discussed by [18] et al. Moreover, many optimizations can be made in the handling of grid map data. However, profiling needs to be done to find where the points of constriction are exactly.

# Appendix A

## Code Layout

The entire project is written in C++. This choice was made because it provides a good performance vs versatility trade-off. Moreover, we try to use as few external libraries as possible to make it easy to compile the project on different machines. When we do use external libraries, they are well-known and supported libraries. If this is not the case, the source code of the library is provided in the project folder.

The project is split up into two programs and three libraries. The two programs are the server and client, which run on the robot and a controlling pc respectively. The libraries make it possible to reuse parts of our code in both the client and the server. Or, in the case of the SLAM library to provide easy isolation of this part of the project. The three libraries are, a logging library; a networking library; and the SLAM library.

### A.1 Server

Since the server does very little work itself, the application is relatively small and only counts eight C++ source files. The main loop of the application can be found in the `main.cc` file. This loop is responsible for handling all incoming network packets. Moreover, the main loop is also responsible for sending battery information to the client.

In the `main.cc` file, we also start three other threads to handle different data streams. These three streams are video, LIDAR, and IMU data. These threads push their results directly to the network stack. Apart from synchronizing network calls, these threads do not interact with the main loop.

The files `bno055.cc`; `rpcamera.cc`; `rplidar.cc` and `thunder.borg.cc` provide a convenient interface to access the hardware. The BNO055 and ThunderBog class access the hardware directly via i2c. The camera is accessed using the `raspivid` application. Communication happens via UNIX pipes. The LIDAR is accessed via the SDK provided by SLAMTEC.

Finally, there are two files `battery.cc` and `heartbeat.cc`. These files abstract the state of the battery and the heartbeat hysteresis process.

### A.2 Client

The client is a much more complex application, containing 28 source files. We will group the files by their task, and discuss their most important details.

### A.2.1 General

Initialization of the application happens mostly in the `main.cc` files. Of most importance are the creation of the `IOController` and `Program` objects. The classes are implemented in the `io_controller.cc` and `program.cc` files respectively.

The `IOController` object is responsible for handling all of the IO. In our case, IO comes either from the network (i.e., the robot) or from the human controller. These events are abstracted by the `IOController` and send to the `Program` object.

The `Program` object handles the entire state of the application. All of the further subsystems discussed in the following subsections are placed in the `Program` class. Hence, a new component should be to this class.

The file `steering_controller.cc`, contains the class `SteeringController`. This class is responsible for converting control input from either a (physical) controller or the autopilot to motor controls. Moreover, this class also manages power limits set by the user.

`streamer.cc` contains the `Streamer` class. This class is responsible for converting the video data stream coming from the network (i.e., an h264 video stream) to frames in memory. We use `libav` to do the decoding of the data stream.

### A.2.2 UI

The UI system of the client uses widgets to display the components on the screen. Each widget inherits the `widget` base class, located in `widget.h`. The files ending in `_widget.cc` implement a widget for use in the UI. Most of the names of the widgets are self-explanatory. If this is not the case, a quick look at the associated header file will reveal its purpose.

Currently, there are two components that the widgets can use. These components are common properties shared between widgets. There is a component to allow a widget to store a child widget. This is common practice in widget-based UI systems. Because it allows for nested placement of widgets. The other component is for widgets that populate an SDL texture. For example, the streamer widget copies video frames to a texture.

Some files end with `_view.cc`. These files create and maintain the widgets associated with a certain view. In our application, there are two views. The main view is the screen that is presented by default to the user. The other view is presented when the user presses the ‘change view’ button on the controller. The alternative view shows a fullscreen global map. Moreover, in this view, it is possible to set the destination for the auto driving feature.

Finally, the top-level widget needs to be rendered to the screen. This task is handled in the `Graphics` class, located in `graphics.cc`. The class initializes an SDL window and renderer. On each draw call, the window size is calculated and set correctly. Then it calls the draw command of the top-level widget, passing a renderer and a draw location. Each widget calculates the position and size of its children, and then recursively calls the draw command.

Widget sizes are calculated bottom-up through the widget tree. Each widget class has a `request_size` method that returns the required size of the widget. This method also takes a hint size, the widget uses this size to calculate its size. Based on the size of the child widgets, a parent widget can calculate its size.

### A.2.3 SLAM

While most of the SLAM code is located in the isolated SLAM library of the project, some application-specific code can be found in the client as well. The code is located in the `Floormapper` class, located in the `floormapper.cc` files. This class manages the worker thread responsible for

inserting IMU and LIDAR data. Moreover, cell sizes and other parameters concerning SLAM are also defined here.

In the worker method, the top-level SLAM choices are defined. For example which processing passes should be used. These passes are implemented in the SLAM library, however, it is up to the user to make a decision on which ones to use. Moreover, parameters for these passes are also defined here. The order and purpose of the used passes are discussed in great detail in the design section.

The `Floormapper` class also keeps track of the robot history map (i.e., the positions of local maps). Moreover, it also transforms data into the right format for the SLAM library to use.

## A.2.4 Auto Piloting and Exploration

Auto piloting of the robot is divided into two source files. The implementation of the route planner can be found in the `route_planner.cc` file. The route planner simply returns a list of waypoints, given a destination and a global map.

The code in `exploration_searcher.cc` implements the search engine responsible for searching the map for places of frontiers and other places of interest. For performance reasons, we added a worker thread that does the computation-intensive tasks.

Perhaps more interesting is the code in `auto_pilot.cc`. This file, contains the implementation of the `AutoPilot` class. This class is responsible for generating the motor control input to send to the robot. These control inputs are dependent on the current mode of the autopilot. Currently, there are two modes: navigation and exploration. In navigation mode, the robot simply drives to a user set destination. In exploration mode, destinations are set according to the results of the `ExplorationSearcher`.

The `AutoPilot` class also implements collision detection and collision avoidance systems. Moreover, it also decides which waypoint is visited and when the route has been completed.

## A.3 SLAM Library

The SLAM library is located in `lib/slam`, there are 11 C++ source files. In this section, we discuss the most important ones. We start with an explanation of the top-level class and work our way down to the classes executing smaller tasks.

The all-encompassing class is the `Graph` class located in the `graph.cc` source file. This class should be included in a program implementing the SLAM library. The purpose of this class is to do the following tasks. First, it determines when to do graph optimizations. Second, it draws global maps. While drawing maps seems simple enough, the difficulty comes from keeping track of dirty areas. We have already discussed how we prevent redrawing of the entire map every update.

The interface of this class is rather simple. The only necessary operation to do SLAM is to push point clouds. The result can then either be obtained by retrieving global grid maps or it can be saved to disk. The point clouds that are being pushed to the `Graph` class are not raw point clouds. Instead, the user is expected to filter a point cloud using the provided passes. This way, the choice is up to the user as to which passes are suited for a given task.

As mentioned, the `Graph` class is responsible for starting the graph optimization process. In our case, this is hardcoded to every 50 LIDAR frames. The actual optimization process is carried out by the external Ceres library. In `ceres_solver.cc`, an adapter function is implemented that converts the internal data structures to a suitable Ceres problem. After optimization, the results are handed back to the `Graph` class. This class then adopts the newly obtained changes and marks updated areas of the global map for redrawing.

The **Graph** class uses other classes to store the global grid map, the local maps, and point clouds. The **GridMap** class, located in `grid_map.cc`, implements the grid map structure and associated functions. The **GridMap** class, manages three internal maps. Of most importance are the hit and miss map. The purpose of these two maps is discussed in the Design chapter. The last map that is stored in this class, is the annotation map. The annotation map serves mostly a debug purpose. It can freely be used to draw on the map, without interfering with the SLAM system. Finally, the **GridMap** also maintains the robot positions of the inserted point clouds (LIDAR frames). The robot positions serve both a functional and a visual purpose. Functionally, the robot positions are often used by exploration searches to mark valid (initial) robot poses. Visually, the robot positions are also drawn on the map, this shows up as the traveled path of the robot.

Because of the intrinsic discrete nature of grid maps, and the continuous nature of nature; the **GridMap** class provides two coordinate systems, that can be used to access the underlying data. The most obvious method is to access the cell data using continuous vectors. Here, a floating-point 2D vector is used to access the grid maps. The units of this vector are in meters. When looking up this vector in a (grid) map, the obtained value is simply the value of the cell that contains the vector value. For example, the  $i^{th}$  cell contains the continuous values from  $[d \cdot i, d \cdot (i + 1))$ , where  $d$  is the cell size. In actuality, this example is extended for two dimensions.

The other coordinate system that can be used to access the maps are integer typed 2D vectors. Here the units of the vector are simply the cell indices. Moreover, the  $[0, 0]$  vector denotes the center cell<sup>1</sup>. It may seem unnecessary to include both these coordinate types to access the same data. Having just the continuous coordinate system would suffice. However, in practice, the ability to iterate the cells using integer coordinates is a nice to have feature. It prevents unnecessary coordinate conversions and prevents rounding errors from occurring. All of the flood fill based algorithms use the indices-based vectors to access the cells directly. Moreover, screen updates also access the maps this way. However, insertion of point cloud data uses the floating-point vectors.

Similar to the (grid) maps in **GridMap**, there is also a general-purpose **Map** class, located in `map.cc`. The data stored in this map is arbitrary. The access methods are similar to the ones discussed for the **GridMap**. The **Map** class is used by the client to store the history maps indices.<sup>2</sup>

The **LocalMap** class contains the information needed to update and maintain the local maps in the graph. A **GridMap** instance stores the local grid map. Moreover, the initial pose and the current pose (i.e., the pose of the robot position after the last point cloud insertion). By storing the initial pose, the first point cloud can be inserted in the center of the grid map. By doing this the local grid maps can have a small size. In fact, given that (i) we filter any data points further away than 4 meters in a LIDAR scan. (ii) The maximal speed of the robot (at which we can do SLAM reliably at least) is around 1 m/s. (iii) A local map takes around 4.2 seconds to make (50 frames at 12 Hz). We can conclude that the dimensions of a local map do not need to exceed 16.4 meters.

The **IMUData** class, located in `imu_data.cc` stores all of the accumulated IMU data throughout a SLAM session. Accessing IMU datum frames is done using its index. The first IMU datum has an index of zero, the second an index of 1, and so on. However, many times it is more useful to find an IMU datum associated with a timestamp. To do this, the class implements a binary search method that can be used to find the datum point before and after a certain time.

One problem with this approach is that accessing a single datum point is still  $\mathcal{O}(\log n)$ . However, many times we want to access the datum points consecutively, starting somewhere in

<sup>1</sup>The upper right cell of the four cells that can be considered center cells, to be precise.

<sup>2</sup>The **GridMap** class should have used the **Map** class to store its internal grid maps. However, due to time constraints, this was never corrected.

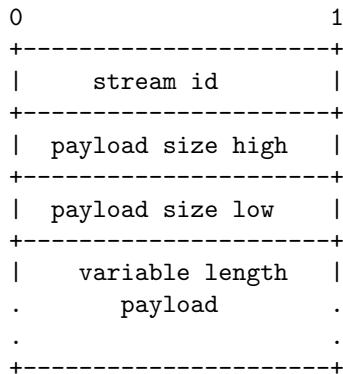
the middle of the array. To solve this, the binary search algorithm also takes a hint argument. This hint is set to the previously found index. By first checking the neighbors of the hint index, only finding the first element in a linear sequence has complexity  $\mathcal{O}(\log n)$  the rest of the elements have a constant time lookup.

Most of the time we are not interested in the datum frame before or after a timestamp, but rather the linear interpolation between these two at the given timestamp. The `IMUData` class, implements just that. This linear interpolation method uses the binary search algorithm described in the previous paragraph. As a result, linear interpolation of consecutive elements is a fast operation.

The `IMUdata` class also implements the smoothing pass for the IMU data. Because this pass requires previous data to function, it can not easily be implemented as a stateless function as is the case for the other passes. The implementation of these passes can be found in `passes.cc`. As a reminder these passes are used to: sanitize incoming raw LIDAR and IMU data; Scanmatch point clouds to a grid map; Insert point clouds into a grid map and clean up grid maps.

## A.4 Networking Protocol

The communication between the robot and the client happens via WiFi. The connection can be made over the local network or the internet. The connection between server and client is a non-secure TCP connection. In the `lib/network` folder, we have implemented a networking library. This library builds upon the TCP connection. We have multiple data streams, that share a common TCP connection. To keep the streams apart, we pack them into smaller packets and associate a network type (stream id) with each packet. Each packet also contains an unsigned 16-bit integer field containing the size of the payload. The diagram below shows the structure of a network packet.



Integers larger than one byte, are sent using the network byte order (i.e., big-endian). Moreover, we also have to send floating-point numbers. To prevent having to deal with byte ordering of floats, they are always convert them to integers first. We can do this because we know in advance what minimal precision is required. The float is multiplied with some big number (e.g., given a float with unit meter, we can multiply by 1000 to obtain 1 mm precision) on the sending side. On the receiving side, the number is divided by this number to obtain the (more or less) original floating-point number. The integer that is sent over the network is once again network order encoded in the message.

For timing reasons, we want the methods of the `Network` class to end as quickly as possible. Moreover, because the `Network` instance is shared between multiple threads, a blocking method



would affect all threads attempting to also access the object. When receiving data, the problem is solved by using the UNIX `select` system call to make sure we can do a non-blocking `recv`. This method leaves a small possibility that the call will block. The way this happens is as follows. The `select` system call returns, indicating that a non blocking `recv` can be done. However, not all bytes of the packet may have arrived at this time. In time case, the `Network::recv_pkt` methods can block. In practice, we have not noticed any problems. This is because the network packages have a relatively small size. As a result, blocking generally will not take too long. Moreover, all of the time-sensitive processes run on the robot. Because the robot is sending much more data than it is receiving, the problem crops up less often.

To prevent blocking of the `Network::send_pkt`, we use an internal send buffer in the `Network` class. During a call to `send_pkt`, the packet is appended to the end of the buffer. Then, we try to send as much of the data from the buffer as we can without blocking.
















## Appendix B

# Controls and Indicators

### B.1 Controls

Button	Main View	Map View
	Steer	Steer
	Pan	Pan
	Center on robot	Center on robot
	Forwards	Forwards
	Reverse	Reverse
	Increase power limit	Increase power limit
	Decrease power limit	Decrease power limit
	Switch view	Switch view
	Increase minimap size	
	Decrease minimap size	
	Zoom in	Zoom in
	Zoom out	Zoom out
	Toggle AP	Set destination
	Switch AP state	Reset destination

## B.2 Indicators

Indicator	Description
	Autopilot active
	Autopilot nothing to do
	Autopilot stopped
	Navigation mode active
	Navigation mode no destination
	Navigation mode no route possible
	Exploration mode active
	Battery good
	Battery almost empty
	Recharge battery
	Battery brownout detected
	Driving close to object
	Collision detected
	Excessive yaw angle detected
	Excessive pitch/roll angle detected

# Bibliography

- [1] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 3 2022.
- [2] Maram Alajlan, Anis Koubâa, Imen Châari, Hachemi Bennaceur, and Adel Ammar. Global path planning for mobile robots in large-scale grid environments using genetic algorithms. In *2013 International Conference on Individual and Collective Behaviors in Robotics (ICBR)*, number 1, pages 1–8, 2013.
- [3] Syahrul Fajar Andriawan Eka Wijaya, Didik Setyo Purnomo, Eko Budi Utomo, and Muhammad Akbaryan Anandito. Research Study of Occupancy Grid map Mapping Method on Hector SLAM Technique. *IES 2019 - International Electronics Symposium: The Role of Techno-Intelligence in Creating an Open Energy System Towards Energy Democracy, Proceedings*, pages 238–241, 2019.
- [4] Josep Aulinas, Xavier Lladó, Joaquim Salvi, and Yvan R. Petillot. SLAM based selective submap joining for the Victoria Park Dataset. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, 7(PART 1):557–562, 2010.
- [5] Josep Aulinas, Yvan Petillot, Joaquim Salvi, and Xavier Lladó. The SLAM problem: A survey. *Frontiers in Artificial Intelligence and Applications*, 184(1):363–371, 2008.
- [6] Tamir Blum, William Jones, and Kazuya Yoshida. PPMC Training Algorithm: A Deep Learning Based Path Planner and Motion Controller. *2020 International Conference on Artificial Intelligence in Information and Communication, ICAIIC 2020*, pages 193–198, 2020.
- [7] Johann Borenstein and Yoram Koren. The vector field histogram-fast obstacle avoidance for mobile robots - Robotics and Automation, IEEE Transactions on. *IEEE Transactions on Robotics*, 7(3):278–288, 1991.
- [8] Oliver Brock and Oussama Khatib. High-speed navigation using the global dynamic window approach. *Proceedings 1999 IEEE international conference on robotics and automation*, 1:341–346, 1999.
- [9] Jingang Cao. Robot Global Path Planning Based on an Improved Ant Colony Algorithm. *Journal of Computer and Communications*, 4, 2016.
- [10] Honglei Che, Zongzhi Wu, Rongxue Kang, and Chao Yun. Global path planning for explosion-proof robot based on improved ant colony optimization. *Proceedings of 2016 Asia-Pacific Conference on Intelligent Robot Systems, ACIRS 2016*, pages 36–40, 2016.

- [11] Hsun Chiang Chia, Jui Chiang Po, Jerry Chien Chih Fei, and Sin Liu Jin. A comparative study of implementing fast marching method and A\* search for mobile robot path planning in grid environment: Effect of map resolution. *Proceedings of IEEE Workshop on Advanced Robotics and its Social Impacts, ARSO*, 2007.
- [12] Yee Zi Cong and S. G. Ponnambalam. Mobile Robot Path Planning using Ant Colony Optimization. *IEEE/ASME International Conference on Advanced Intelligent Mechatronics.*, 2009.
- [13] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta\*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.
- [14] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1):23–33, 1997.
- [15] SS S Ge and YJ J Cui. Dynamic Motion Planning for Mobile. *Electrical Engineering*, 13(Med):207–222, 2002.
- [16] Woong Gie Han, Seung Min Baek, and Tae Yong Kuc. Genetic algorithm based path planning and dynamic obstacle avoidance of mobile robots. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 3:2747–2751, 1997.
- [17] C. Harris and M. Stephens. A Combined Corner and Edge Detector. In *Alvey vision conference*, pages 23.1–23.6, 2013.
- [18] Soonhac Hong, Udo Frese, Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, Wolfram Burgard, Edwin E.B. Olson, Karl Granström, Thomas B. Schön, Juan I. Nieto, Fabio T. Ramos, Josep Aulinas, Yvan Petillot, Joaquim Salvi, Xavier Lladó, Joao Machado Santos, David Portugal, Rui P. Rocha, Shaofeng Wu, Jingyu Lin, R. E. Kalman, Tomohito Takubo, Takuya Kaminade, Yasushi Mae, Kenichi Ohara, Tatsuo Arai, Shoudong Huang, Gamini Dissanayake, Magnus Linderöth, Kristian Soltesz, Anders Robertsson, Rolf Johansson, Tong Tao, Yalou Huang, Fengchi Sun, Tingting Wang, Brian Yamauchi, Zhongli Wang, Yan Chen, Yue Mei, Kuo Yang, Baigen Cai, Wolfgang Hess, Damon Kohler, Holger Rapp, Daniel Andor, Joan Vallvé Navarro, Juan Andrade-Cetto, Joan Solà, Sebastian Thrun, Wolfram Burgard, Dieter Fox, Giorgio Grisetti, Rainer Kummerle, Hauke Strasdat, Kurt Konolige, Cyrill Stachniss, Wolfram Burgard, Edwin E.B. Olson, Kurt Konolige, Giorgio Grisetti, Rainer Kümmerle, Wolfram Burgard, Benson Limketkai, Regis Vincent, F. Lu, E. Milios, Guolai Jiang, Lei Yin, Guodong Liu, Weina Xi, Yongsheng Ou, Muhammad Sualeh, Gon Woo Kim, Luca Carlone, Andrea Censi, Frank Dellaert, Baichuan Huang, Jun Zhao, Jingbin Liu, Niko Sünderhauf, Peter Protzel, Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, John J. Leonard, Hugh Durrant-Whyte, Tim Bailey, Arnaud Doucet, Nando De Freitas, and Stuart Russent. Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms. *Proceedings - IEEE International Conference on Robotics and Automation*, 2016-June(1):1309–1332, may 2018.
- [19] M. Khatib, H. Jaouni, R. Chatila, and J. P. Laumond. Dynamic path modification for car-like nonholonomic mobile robots. *Proceedings - IEEE International Conference on Robotics and Automation*, 4(April):2920–2925, 1997.
- [20] Oussama Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots Abstract. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 5, pages 396–404, 1990.

- [21] Dmitriy Kogan and Richard M Murray. Optimization-Based Navigation for the DARPA Grand Challenge. *System*, 2006.
- [22] Kurt Konolige, Giorgio Grisetti, Rainer Kümmerle, Wolfram Burgard, Benson Limketkai, and Regis Vincent. Efficient sparse pose adjustment for 2D mapping. *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, (June 2014):22–29, 2010.
- [23] Chaymaa Lamini, Said Benhlila, and Ali Elbekri. Genetic algorithm based approach for autonomous mobile robot path planning. *Procedia Computer Science*, 127:180–189, 2018.
- [24] Yangming Li and Edwin B. Olson. Structure tensors for general purpose LIDAR feature extraction. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1869–1874, 2011.
- [25] Yuanchang Liu and Richard Bucknall. The angle guidance path planning algorithms for unmanned surface vehicle formations by using the fast marching method. *Applied Ocean Research*, 59:327–344, 2016.
- [26] Iker Lluvia and Elena Lazkano. Active Mapping and Robot Exploration : A Survey. *Sensors*, pages 1–26, 2021.
- [27] Feng Lu and Evangelos Milios. Robot Pose Estimation in Unknown Environments by Matching 2D Range Scans. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 18(3):249–275, 1997.
- [28] Thi Thoa Mac, Cosmin Copot, Duc Trung Tran, and Robin De Keyser. A hierarchical global path planning approach for mobile robots based on multi-objective particle swarm optimization. *Applied Soft Computing Journal*, 59:68–76, 2017.
- [29] S. Quinlan and O Khatib. Elastic bands: connecting path planning and control. *Proceedings IEEE International Conference on Robotics and Automation*, 2:802–807, 1993.
- [30] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. Trajectory modification considering dynamic constraints of autonomous robots. *7th German Conference on Robotics, ROBOTIK 2012*, pages 74–79, 2012.
- [31] José Ricardo Sánchez-Ibáñez, Carlos J Pérez-Del-Pulgar, and Alfonso García-Cerezo. Path Planning for Autonomous Mobile Robots: A Review. *Sensors (Basel)*, 21(23):7898, 2021.
- [32] Mbl Saraswathi, Gunji Bala Murali, and B. B.V.L. Deepak. Optimal Path Planning of Mobile Robot Using Hybrid Cuckoo Search-Bat Algorithm. *Procedia Computer Science*, 133:510–517, 2018.
- [33] Homayoun Seraji and Ayanna Howard. Behavior-based robot navigation on challenging terrain: A fuzzy logic approach. *IEEE Transactions on Robotics and Automation*, 18(3):308–321, 2002.
- [34] Ioan Sucan, Sachin Chitta, Paolo Fiorini, and Zvi Shiller. Motion Planning in Dynamic Environments Using Velocity Obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.
- [35] Tong Tao, Yalou Huang, Fengchi Sun, and Tingting Wang. Motion planning for SLAM based on frontier exploration. *Proceedings of the 2007 IEEE International Conference on Mechatronics and Automation, ICMA 2007*, (60605021):2120–2125, 2007.

- [36] Sebastian Thrun. The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures. *The International Journal of Robotics Research*, 1998.
- [37] Adem Tuncer and Mehmet Yildirim. Dynamic path planning of mobile robots with improved genetic algorithm. *Computers and Electrical Engineering*, 38(6):1564–1572, 2012.
- [38] I. Ulrich and J. Borenstein. VFH+: Reliable obstacle avoidance for fast mobile robots. *Proceedings - IEEE International Conference on Robotics and Automation*, 2(May):1572–1577, 1998.
- [39] Alberto Valero-Gomez, Javier V. Gomez, Santiago Garrido, and Luis Moreno. The Path to Efficiency: Fast Marching Method for Safer, More Efficient Mobile Robot Trajectories. *IEEE Robotics and Automation Magazine*, 20(4):111–120, 2013.
- [40] Zhongli Wang, Yan Chen, Yue Mei, Kuo Yang, and Baigen Cai. IMU-assisted 2D SLAM method for low-texture and dynamic environments. *Applied Sciences (Switzerland)*, 8(12), 2018.
- [41] Yupei Yan and Yangmin Li. Mobile Robot Autonomous Path Planning Based on Fuzzy Logic and Filter Smoothing in Dynamic Environment. In *World Congress on Intelligent Control and Automation (WCICA)*, pages 1479–1484, 2016.
- [42] Panagiotis G. Zavlangas and Spyros G. Tzafestas. Motion control for mobile robot obstacle avoidance and navigation: A fuzzy logic-based approach. *Systems Analysis Modelling Simulation*, 43(12):1625–1637, 2003.
- [43] Yinyan Zhang, Shuai Li, and Hongliang Guo. A type of biased consensus-based distributed neural network for path planning. *Nonlinear Dynamics*, 89(3):1803–1815, 2017.
- [44] Qin Zou, Qin Sun, Long Chen, Bu Nie, and Qingquan Li. A Comparative Analysis of LiDAR SLAM-Based Indoor Navigation for Autonomous Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–15, 2021.