This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

# Acknowledgements

I would like to thank my supervisor Gerald Steinbauer for his support during this master thesis and constructive discussions on this work even in stressful periods.

Further I would also like to thank my colleagues from the TEDUSAR team as well as from the Autonomous Intelligent System laboratory for their support and interesting discussions.

Finally I would like to thank my family for supporting my during my entire study and my whole life.

Michael Stradner
Graz, Austria, September 2017

# Abstract

This master thesis describes a robot system which is able to generate a three-dimensional map of its environment, classify the terrain in terms of traversability, and explore unknown areas autonomously. The mapping is achieved using a stop-and-go procedure where the robot stops, rotates a 2D laser scanner to make a 3D measurement of its environment and moves on to the next place to explore. The 3D measurement is matched into the existing graph-based map using an ICP algorithm. The ICP registration is also needed to correct odometry errors of the 6D localization of the robot. Additionally an ICP-based loop-closure procedure is applied and to achieve a globally consistent map, a graph optimization is used.

A local part of the map is afterwards converted into a 3d occupancy grid and the surface of this grid is classified into traversable, obstacle, and cliff cells. An inflation area around cliffs and obstacles is added to prevent the robot from collisions as well as going over cliffs. In the next step frontier areas of this 3D grid are classified. A frontier voxel is a traversable voxel which is adjacent to currently unknown voxels. During the terrain-classification a navigation-graph is built which is later used to apply an A*-algorithm to find the path to a selected frontier.

After choosing a suitable frontier according to the distance and orientation of the robot, a path to that frontier is calculated. The robot is then able to navigate to this area and map it. Because of the 3D terrain classification the robot is able to plan paths for example over or under bridges as well as across multi-level buildings.

This system has been implemented using the ROS framework, the libpointmatcher library and the $g^2o$ library and was evaluated in a simulation as well as on a real rover.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

This thesis presents an implementation of an autonoumous robot that is capable to map, navigate, and explore three-dimensional spaces.

Due to increased computing power, cheaper sensors and optimized algorithms emerged during the last decades, autonoumous robot systems find more and more the way into our society. They are used for example in logistics to automate and increse the productivity in the logistics supply chain, but are also gaining increased importance in a lot of other areas such as surveilance or inspections. Some logistics companies are also working on robot systems for the last-mile delivery to it's customers. At the beginning of autonomous robots the majority of them was designed to work indoors because of it's reduced complexity in regard to the environment. But with the increased computing power as well as some other factors, the fields in which robots can be used increased.

In the automotive industry, autonomous cars are currently a highly researched topic. These cars sense their environment with multiple sensors, fuse them together to get more reliable representations and use it to navigate, detect collisions, and avoid them. This way they can help to prevent accidents by reducing human-made errors. But also the usage of robots for disaster response teams gets more and more attention. While most of these robots are currently remotely operated, there is a tendency to also automate this and let them explore unstructured environments by their own.

## 1.1 Motivation

Many robots today use a 3d sensor, for example a laser scanner or a stereoscopic camera, to build a representation of the environment. By using such a sensor a high detailed map of the environment can be aquired for navigation or surveying purposes. While in some indoor environments a 2D map is adequate for some tasks, especially in rough terrain there is a realization without a 3D sensor input hardly possible.

However, the navigation is in most cases still calculated either in 2 or 2.5 dimensional space. While a heightmap (2.5D) as a representation of the terrain is sufficient for many environments, there are still many cases where a full 3D navigation capability would be needed for example to plan above or beyond bridges or across multiple floors in buildings. With a fully-fledged 3D navigation, an autonomous robot is able to plan on multi-leveled environments. Then it is also possible to plan over or under bridges or plan a path across multiple floors in a building. Figure 1.1 shows the robot Wowbagger from the Graz University of Technology during a RoboCup Rescue competition where "victims" should be autonomously detected in a simulated disaster area.

## 1.2 Goals and Challenges

The goals and challenges of this thesis are described as follows. The goal is to develop an robot system that autonomously explores it's environment and creates a three-dimensional map of it. The environment therefore could be indoor as well as outdoor on rough terrain such as a stone pit. However, the environment must provide enough features so that a matching algortithm can register two consecutive laser scans with each other. A reliable method to continously append new measurements in an existing map is needed to achieve a consistent representation of the explored area which is used to safely navigate the robot from one place to another. Therefore only environments that

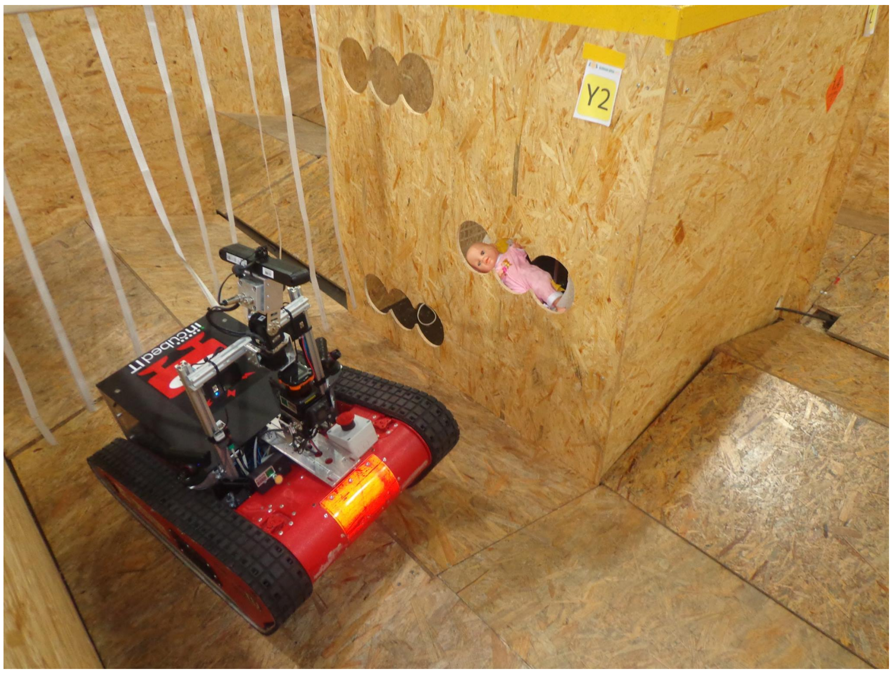

Figure 1.1: Illustration of the robot Wowbagger during a RoboCup Rescue event.

aren't completely flat can be mapped, which normally shouldn't be a problem. The mapped environment gets then classified in traversable areas, obstacles, potential cliffs as well as frontiers and a path to the nearest frontier in respect to the distance and angle from the robot is then choosen as a goal and a path to it is planned. Below is a more detailed summary of the goals of each robot module.

**Mapping**
The mapping is achieved using a stop-and-go procedere, where the robot stops, rotates a 2D laser scanner to get a 3D point cloud, matches this aquired point cloud to the current accumulated map, adds this scan to the current graph-based map, conducts a pose-graph optimization to get a globally consistent map and updates the robot position inside this map, which may be a bit off due to odometry errors.

**Terrain classification**
After a new scan has been inserted into the map, a local area of this map gets converted into an 3D grid map and a terrain classification is applied to. The difficulty with a terrain classification in the 3D space is to evaluate the terrain within resonable time to be able to use it in real-time. Starting from the robot position the terrain gets classified into traversable areas, obstacles, and potential cliffs. Afterwards a robot inflation layer is added and finally the frontier voxels for the exploration goals are chosen. A frontier voxel is a traversable voxel that is adjacent to a currently unknown voxel, that is neither free nor occupied. Utilizing only traversable voxels this way, only areas are chosen to be explored to which the robot can actually navigate to. During this classification phases a graph is created which contains the traversable voxels which are later used for path planning. One of the main challenges of the 3D terrain classification is to keep the memory and time requirements for the terrain classification inside reasonable bounds to achieve a real-time capable system.

**Navigation**
Now a frontier voxel is choosen in respect to the distance and angle between the robot and the frontiers. The aim is to choose a frontier that is near the robot and, if possible, is somehow facing near the robot, so the robot doesn't move in a zig-zag-pattern. The navigation graph

created in the classification step is now used to perform a A* search between the robot position and the chosen frontier and executed by the path-following module.

## 1.3 Contribution

In this thesis a robot system is described and evaluated which creates a three-dimensional representation of its surrounding area using a pose-graph. It can be used in an indoor as well as in an outdoor environment. For navigation purposes this map or a part of it is converted into a 3D grid map. Taking the position of the robot inside this map into account the cells representing the surface of this grid get classified in terms of traversability. This is necessary to safely plan paths to regions of currently unexplored areas of this map and avoiding obstacles or other barriers like to steep terrain. The represented system is able explore and map an area in real time and is able to plan paths across multi-level environments like over or under bridges as well as inside multiple-floor buildings.

## 1.4 Outline

This thesis is arranged in the following order. In chapter 2 the problem formalization is outlined while chapter 3 covers the related research in this area. The prerequisites like the used frameworks and algorithms are covered in chapter 4, while chapter 5 discusses the concept and architecture for solving the three-dimensional navigation and exploration problem. In chapter 6 specific implementation details are highlighted and the evaluation results of this work are presented in chapter 7. Chapter 8 discusses the outcome of this thesis and presents possible future improvements.

# 2 Problem Formulation

We consider a three-dimensional world $W \subset \mathbb{R}^3$ in which a robot explores the environment. The robot is defined in this world by a 6D pose where $Pose := \langle x, y, z, \phi, \theta, \psi \rangle$ where $x$, $y$ and $z$ denotes the translational part and $\phi, \theta, \psi$ the rotational part.

Now let $\chi$ be the space of 6D poses where $\chi \cong \mathbb{R}^3 \times SO(3)$ and $C \subset \chi$ the set of all possible and resonable robot configurations considering the kinematic constraints of the robot in the given environment. Moreover $q \in C$ denotes a specific robot configuration. A reasonable robot configuration is a configuration which the robot is able to reach in a stable manner while driving with its wheels along the surface of the terrain considering the specific robot parameters such as center of mass and maximal terrain inclination. Next we transfer the world $W$ into a 3D gridmap and denote the search space $S = \{w \in W | surf(w)\}$ where we assume that $surf(w)$ returns all surface voxels. Let $D(q)$ be the surface voxels that are visible given the robot configuration $q$. The goal is to find a sequence $Q = \langle q_1, q_2, ..., q_n \rangle$ with $q_i \in \mathbb{R}^3 \times SO(3)$ so that $S \setminus \bigcup\limits_{i=1}^{n} D(q_i) = \emptyset$ with the constraint $\underset{Q}{\text{minimize}} |Q|$.

Let $Q_{free} \subseteq C$ be the set of all collision free robot configurations and a path $P_{i,i+1} : [0,1] \longrightarrow closure(Q_{free}) \setminus Q_{free}$ between two robot configurations $q_i$ and $q_{i+1}$ where $P_{i,i+1}(0) = q_i$ and $P_{i,i+1}(1) = q_{i+1}$. Therefore the path meets the requirement that the robot moves along the surface.

During the robot exploration of the world a map of the environment should be built. Given a set of measurements $d = \{u_1, z_1, u_2, z_2, ..., u_j, z_j\}$ where $u_i$ represents the odometry measurement and $z_i$ the corresponding sensor measurement of the 3D laser scanner with $u_i \in \mathbb{R}^3 \times SO(3)$

and $z_i = \{l_1, l_2, ..., l_o\}$. A laser measurements $z_i$ consist of a set of points $l_k \in \mathbb{R}^3$.

We assume that the world and therefore the 3D map can be represented in a 3D grid. A grid map $m$ partitions the space into many grid cells $m_i$ so $m = \{m_i\}$ , with each cell storing a value if it is free or occupied. The goal is to find the most likely map $m^* = \arg\max_m p(m|z_1, ..., z_j, u_1, ..., u_j)$ with $p(m|z_1, ..., z_j, u_1, ..., u_j) = \prod_i p(m_i|z_1, ..., z_j, u_1, ..., u_j)$ by assuming that there exists an inverse sensor model with $inverse\_sensor\_model(m_i, z_j, u_j) = log\frac{p(m_i|z_j,u_j)}{1-p(m_i|z_j,u_j)}$.

# 3 Related Research

This chapter gives an overview of the relevant literature related to this thesis and is grouped into three different sections. The first section covers techniques for simultaneous localization and mapping (SLAM) while the second section discusses the navigation approches in the three-dimensional space. In the last section some literature regarding exploration procedures are outlined.

## 3.1 SLAM

A fundamental skill for autonomously exploring robots is to create a map of its environment and simultaneously localize itself inside it, which is called simultanious localization and mapping (SLAM)[1]. When the robot moves and explores its surroundings, new measurements lead to an update of the current map while it also needs to keep track of of its own position relative to this map. Several SLAM methods exist, one of them is for example graph-based SLAM[2] which is also used in this thesis. The map is thereby represented as a series of nodes connected with edges between them. These nodes store sensor measurements of the environment while the edges represent relations between them. The graph itself represents a least-squares optimization problem of an error function which is then optimized to gain a consistent map. So during the construction of the graph, the optimization step calculates the most consistent representation of it. Two common frameworks for this kind of graph optimization are for example $g^2o$[2] and GTSAM[3]. The $g^2o$ library performs a nonlinear least squares problem on a graph or hypergraph while the GTSAM library is based on factor graphs.

Zhang and Singh [4] splitted the SLAM problem into two algorithms, one is used to estimate the odometry of the lidar at a high frequency but low fidelity to use a spinning lidar during movement. The algorithm undistorted the received scans which were deformed by the movement. The other algorithm is used for the matching of the undistorted point clouds and registering them to the map. This way it is possible to map the environment in real-time while using a constantly rotating lidar even during movement.

Leingartner et al. [5] evaluated different mapping approaches in a large-scale disaster environment. The compared 2D as well as different 3D mapping techniques with each other in terms of map quality and acceptance for first responders like fire brigades.

For building large-scale 3D maps, Sprickerhof et al. [6] presented a loop closure technique in three-dimensional space called explicit loop closure heuristic. Instead of traditionally building the graph and later optimizing it, they separated the last scan that led to the loop closure from the previous ones and registers that scan explicitly. Nüchter et al. [7] presented also a 6D SLAM system for mapping outdoor environments and to benchmark their created map. They used a 2D map from a land registry office and fused it with 3D data captured with an airplane. That way they created a precise reference map for their algorithm.

Dube et al. [8] used kind of a different approach. They segment the pointcloud into clusters and extract specific descriptors to classify for example trees, parts of buildings or cars which are recorded. If the robot visits the same place again and these descriptors match with the previously recorded ones they detect a loop-closure.

SLAM is also possible without a explicit 3D sensor but just using cameras which is called visual SLAM. For example Taketomi et al. [9] reviewed different common visual SLAM algorithms in their work. One of them is for example ElasticFusion[10] which uses a RGB-D camera to capture consistent surfel-based maps and doesn't use a pose-graph. A different approach uses LSD-SLAM[11] where only a monocular camera is needed and a direct image alignment is applied instead of a keypoint-based approach.

## 3.2 Navigation

Planning a path on a given map and follow its trajectory is one of the key abilities for a mobile robot. Most robot systems require an already existing map of the environment for the path planning process. The necessary map could be created by driving the robot manually over a remote control or via defined waypoints and store the map for later usage. But the map could also be created iteratively on runtime for example by an autonomous exploration algorithm where unexplored areas are detected and the navigation system is used to drive the robot to the edge of the known map so that the onboard sensors can perceive new areas. In general the different navigation approaches can be subdivided into three different groups: 2D, 2,5D, and 3D navigation.

In the 2D navigation approach the perceived world is down-projected into a two-dimensional map or the robot only drives on flat ground with obstacles high enough so that they are recognized by a 2D laser scanner or other sensors. The benefit of this is a fast navigation capability which doesn't require much computational resources. The *move_base* package for example provides such an open source system in the ROS ecosystem which uses an grid map and the A$^*$-algorithm to plan a global path between the current robot position and a goal. To follow the trajectory of this path, a local planner which uses the Dynamic Window Approach (DWA)[12] is used. In certain environments it is enough to create a 2D map using SLAM and navigate on it. One drawback for this approach is that a 2D map is limited to only one area or floor of a building. To be able to navigate on multiple floors of a building, Pratkanis et al [13] used mulitple 2D maps for different floors as well as for indoor and outdoor areas. These maps are connected with each other and a transition to the relevant map is made when the robot enters a specific area. This way he was able to send a PR2 robot from one floor to another using an elevator.

GMapping[14] is for example a popular used software module in the ROS environment for SLAM in 2D. For map creation it uses a Rao-Blackwellized particle filter where each particle carries an own map of the environment and represents a robot trajectory. In the beginning the initial pose of the robot needs to be known. The resampling of the particles is done using a proposal distribution which depends on the odometry measurements as well as the recent laser measurements and in the end a occupancy grid map is generated.

To handle large areas which should be mapped and used for navigation, Lassnig [15] used a pose-graph in conjunction with a roadmap with a graph topology for high-level planning. The high-level plan is then refined into connected local grid maps which he used for the navigation task. That way it was possible to send an robot autonomously across a university campus for transportation tasks.

Another problem with navigating on 2D maps is detecting obstacles of different shape on different heights. A typical office environment contains for example chairs and tables and a robot is only equipped with an horizontal-placed 2D laser scanner, only a height-slice of the environment that the laser actually sees is used to detect obstacle. When a robot is higher than for example a table but the laser scanner only sees legs of the table it could happen that a simple 2D navigation system plans a path under the table which leads to a collision. Marder-Eppstein et al. [16] used a tilting laser scanner in combination with a voxel grid where the obstacles are represented in 3D and projected down to a 2D map to avoid them. This approach can be grouped into the 2,5D navigation but it is still limited to only one floor. Maier et al. [17] used a depth camera mounted on a humanoid robot called Nao to avoid obstacles in a similar way. They used an 3D grid map in which the three-dimension environment is represented. Actually the used two 3D maps, one representing the static environment while the other was used to avoid dynamic obstacles. Because planning on a 3D map is very time consuming and needs still a lot of computation power they trimmed down the 3D map to the size of the robot and made a projection of it into a 2D map.

To be able to navigate across rough terrain, Fankhauser et al. [18] used a local elevation map for this task which takes the uncertainties of the range measurements into account as well as the state estimation of the robot. When the robot moves, the data in the map is updated according to the uncertainty estimates of the robots pose-estimation to better compensate the drift between the robot and the local elevation map.

A different navigation approach was used by Pütz et al. [19] where he used meshes to navigate on them. They used the 3D pointclouds of the environment to reconstruct a mesh of the surroundings containing a graph representing which triangle-mesh-cell are connected with each other. Afterwards they estimated the roughness and height differences of the terrain on this mesh and used it to plan a path. That way they can plan through multi-level environments and for example can even used for robots that can climb walls to navigate inside a tube system for inspections. Their work is available as an open-source ROS package and we tried to get it running on our computer but we didn't succeed to get it to work properly.

Another approach to for 3D navigation was proposed by Colas et al. [20] where he used tensor voting on point clouds to extract primitives like planes, spheres etc which are used to evaluate the traversability. Their method is sensitive to the quality of the map representation and on the density of the points which needs some parameter tuning.

Hertle and Dornhege [21] used an 3D grid map in which they classified the terrain of the surface cells for path planning. However, they used a full 3D representation of the environment and didn't built the map incrementally by sensor input. In this thesis we used a similar approach for path planning in which we convert our map into an 3D grid map where we classify it in terms of traversability to evaluate the next area the robot should navigate to and explore.

## 3.3 Exploration

To map new areas an exploration algorithm is needed which places navigation goals inside an existing map so that new unexplored areas are mapped when or while the robot reaches this goal. One publicly available package in the ROS environment for exploring 2D space is called *frontier_exploration* and is based on the algorithm based on frontiers presented by Yamauchi [22]. He used an evidence grid to probabilistically classify cells into the category open, unknown or occupied. Frontier cells are open cells that are adjacent to unknown cells. He grouped them to frontier regions and let the robot navigate to the closest frontier region. Doing so, the robot perceives new areas.

Another approach was presented by Maurer and Steinbauer [23] where they used a risk-aware exploration where a trade-off between minimizing the risk for collision and dangerous objects like heat signatures and maximizing the information about the area is considered.

A frontier-void-based approach was presented by Dornhege and Kleiner [24] where a robot platform with an 5-DOF manipulator was used to detect victims in a simulated disaster area. This manipulator was equipped with 3D sensors and their algorithm determined the minimal sequence of sensor viewpoints to efficiently explore the search space. For choosing exploration areas they combined unexplored 3D volumes together with frontiers which are boundaries between known and unknown space.

Senarathne and Wang [25] presented an approach for the three dimensional case where instead of free-space frontiers they used surface-frontiers. The used a 3D grid map to represent the environment and classified frontiers only on the surface cells of this map.

# 4 Prerequisites

The aim of this chapter is to present the basic prerequisites to understand this thesis. At first the Robotic Operating System is introduced, followed by graph-based SLAM. Afterwards the ICP-algorithm is explained along with the Ocotmap library.

## 4.1 Robot Operating System

The Robot Operating System [26] in short ROS is an open source meta-operating system. It requires an operating system such as Linux and offers a lot of features which makes developing and running robotics software much easier. It features for example a message-passing-interface, a package managemnt system, or libraries and tools for running code across multiple machines. The most common programming languages in ROS are C++ and Python but it also supports many others. A big benefit is also that it is supported by a big community and offers a lot of packages. A package is a kind of "container" which can include nodes, configuration files, ROS-independant libraries, datasets etc. In ROS a system consists of a set of nodes where each node is a process which performs some individual computations. The all have unique names which identifies them and they are communicating via messages to achieve a common goal. This messages are distributed over topics which are uniquely named buses where nodes can publish or subscribe to it. However, nodes aren't aware if other nodes also subsribe or publish to a specific topic. So if a node publishes a mesage on a topic the nodes which subscribed to it will receive it. In some cases a synchronized communication is needed which could be done by using a service. A service consists of a defined request and reply message. A node can offer a service under a specific name and a client (or other node) can invoke it by sending a request message. The call is blocking so after sending the request, the client has to wait for the reply to proceed. However, for some long lasting task it is desireable to receive some feedback for a request while the task or request is still performing or maybe the client wants to cancel the request. For such a scenario the actionlib libary can be used.

Because robots usually contain a lot of different coordinate frames that can change over time, for example a moving arm or rotating lidar, ROS offers a package called *tf* which keeps track of them and allows operations like requesting or publishing transforms between these frames. Another often used package in ROS is called *move_base* which is a 2D path planning and path execution module consisting of a global and local planner. With it a robot can for example plan a path around obstacles when the robot location, a map and/or sensor readings and a goal is given.

ROS also supports the Gazebo simulator which can be used to simulate a robot in a three-dimensional world. There exists for example a lot of different predefined plugins such as sensors like a laser scanner. Another plugin simulates for example a skid-steer drive.

## 4.2 Graph-Based SLAM

The simulataneous localization and mapping (SLAM) problem can also be represented by a graph. Every node in the graph represents a pose of the robot which are connected by edges to other nodes. Each edges represents a spatial constraint between the two connected nodes. When the robot visits a specific place a second time, it should be recognized and a loop-closure performed. An edge is inserted between the current node and the node of that similar position. Each edge stores an uncertainty matrix with it and during graph optimization, the algorithm tries to find a node configuration that minimzes the edge error by minimizing the sum of the least squared errors.

## 4.3 ICP

The Iterative Closest Point (ICP) algorithm [27] is used to calculate a transformation that minimizes the difference between two pointclouds. One pointcloud is the reference and is kept in a fixed position while the other one, the source, is transformed in a way to best fit to the target cloud. The ICP-algorithm iteratively adjusts the transformation from the source to the target to minimize for example the sum of the squared error of the euclidian distance between the points. One requirement for the ICP-algorithm is that the target- and source-pointcloud are roughly pre-aligned. Otherwise the algorithm would get stuck in a local minima which is not the global one. In this work we used the libpointmatcher library [28] for matching two different pointclouds together. The result is the transformation between them as well as some other parameters like a match ratio ranging from 0.0 to 1.0 indicating how good the pointclouds match with each other.

## 4.4 Octomap

The Octomap library [29] provides a 3D occupancy grid based on an octree (Figure 4.1). An Octree is a tree data structure where each node either has 8 direct children or none. With an Octomap a three-dimensional volumetric map can be represented. Each node in an octree describes a volumetric cube called a voxel. This voxel can be split into 8 smaller voxels until a defined minimal voxel size is reached which is called the resolution of the octree or Octomap. A voxel in the Octomap can either be occupied, free or unknown. To accommodate dynamically changing environments the state of a voxel is modelled probabilistically. This way a new sensor reading can update the state of a voxel. The Octomap itself also supports multi-resolution queries so that the octree is only traversed up to a specific tree-depth to get a coarse map. For example a high level planner uses the coarse map for some calculations and chooses a region of interest in which the local planner uses fine-grained map for calculations. The Octomap libary

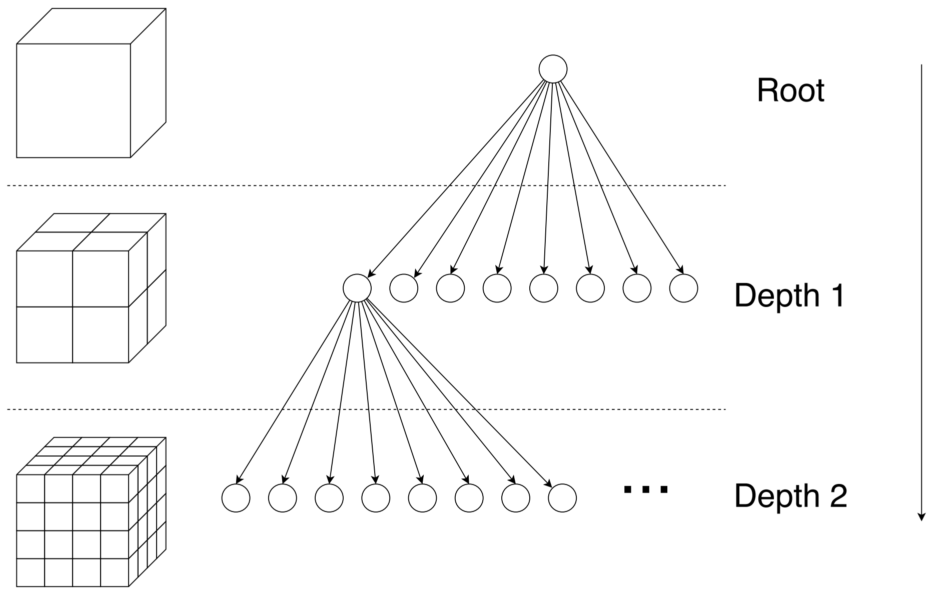

Figure 4.1: Illustration of an octree. Adapted from [30]

also supports to store a color information for each voxel which leads to a colored Octomap. A big advantage the Octomap in comparison to a simple 3D grid is that it is much more memory efficient. However, the downside is that it takes more computing power.

# 5 Concept

## 5.1 Overview

This chapter covers the concept of the developed autonomous exploration system. The used simulation framework with its environment is illustrated as well as the used SLAM approach. Afterwards the algorithm to analyse and classify the terrain is presented which is then used to a chosen frontier and plan a path to it. The frontier calculation is also part of the terrain analysis and provides a list of frontiers, one frontier is then chosen according to a cost function with respect to the robot distance and orientation. The aim of this chapter is to provide the basic concept of this thesis and to provide the basis for the implementation which is presented in the next chapter.

Figure 5.1 gives a short overview of the presented system. The robot perceives its environment and its own state over a set of sensors like a lidar, IMU or wheel odometry. This data is processed and a map, which is based on a graph structure, is generated from it. A part or even the whole map is sent to the module that classifies the terrain on a global level where a navigation goal and a path to it is generated and sent to the navigation stack. The module handling the local terrain classification receives only a local map and generates a costmap in which obstacles are marked. This costmap is also sent to the navigation stack that is now able to drive to the goal by sending commands to the drive system so the robot can move to the goal located in the environment.

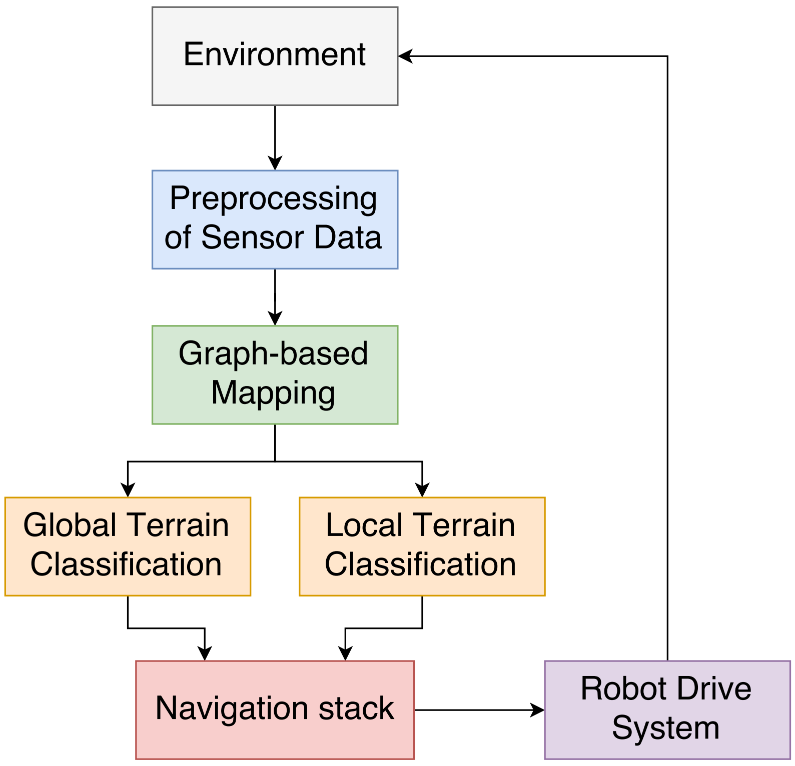
Environment
Preprocessing of Sensor Data
Graph-based Mapping
Global Terrain Classification
Local Terrain Classification
Navigation stack
Robot Drive System

Figure 5.1: Schematic overview of the software system

## 5.2 Simulation of the Environment and Task

The first step was to define in which environment the robot should be used and what kind of tasks it should be able to perform. The robot should be able to operate indoors as well as outdoors in rough terrain, like a stone pit or moon surface. Therefore, the Husky plattform (figure 5.2 and 5.3) from Clearpath Robotics has been selected which should be suitable for the given environment. In the next step sensors had to be selected. Because the goal was to classify rough terrain in three dimensions the sensor to measure the environment has to provide a high resolution and shouldn't be very prone to measurement errors. Therefore a sweeping 2D laser scanner (figure 5.6) has been selected. They are relatively insensitive to changes in the illumination of the environment in contrast to stereo cameras which also provide a lower resolution of the measurements. If a 2D laser scanner is sweeped slow enough, it provides a much higher 3d resolution and is much cheaper compared to an integrated 3D laser scanner. For 3D localisation it is essential to include an Inertial Measurement Unit (IMU) which provides the necessary orientation measurements of the robot. The data from the IMU are fused with the odometry from the wheel encoders to increase the reliability of the overall odometry.

In the next step the robot was modeled in the Unified Robot Description Format (URDF)[31]. A URDF file is an XML Format which is used to represent a robot model in the ROS framework. To simulate this robot in different environments the Gazebo Simulation[32] was used. Gazebo is a three-dimensional simulation environment which can be used in combination with ROS framework and features an realistic physical simulation. This way different robots can easily be simulated in different environments for testing purposes without investing in expensive hardware like robot platforms and sensors. This simulator was also used for example in the DARPA Robotics challenge to simulate a humanoid robot which should perform semi-autonomous tasks in a rescue mission. It is plugin-based and a lot of ROS-plugins which simulate specific sensors can be used as well as some predefined simulation environments. But it is also possible to create an own

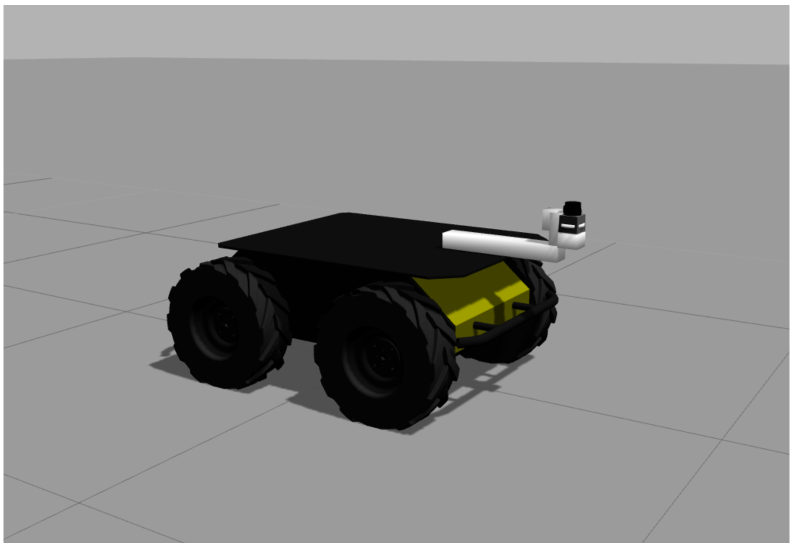

Figure 5.2: Illustration of the simulated husky robot

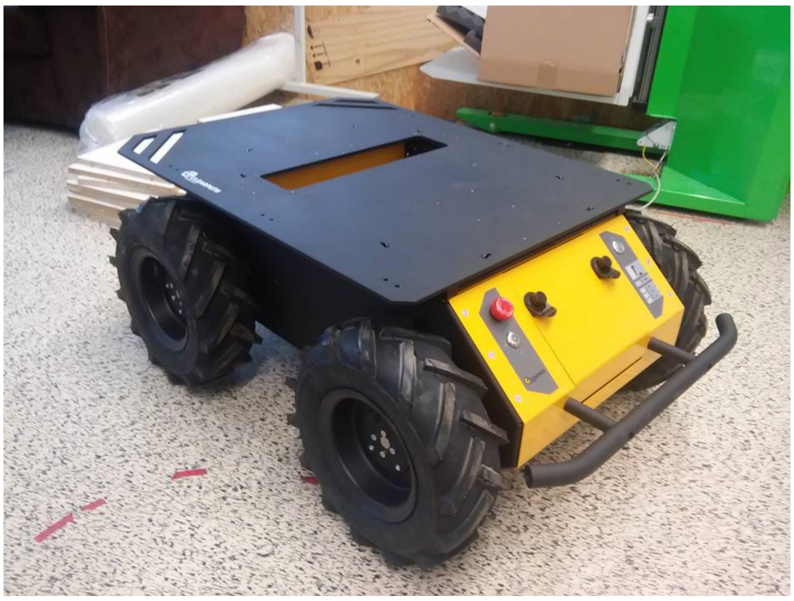

Figure 5.3: Illustration of the real husky robot

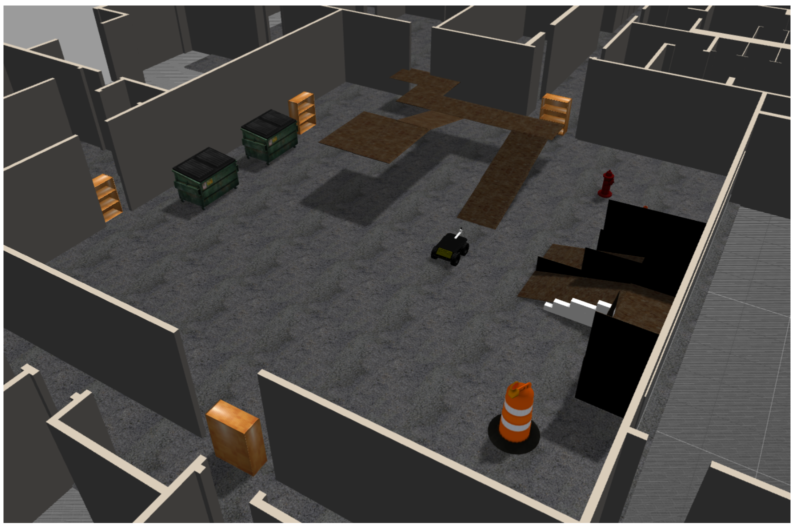

Figure 5.4: Illustration of the simulated ramp world in Gazebo

world.

To evaluate the proposed mapping and navigation system of this thesis two different environments were created. One is more or less an urban setting with lots of detailed features such as hydrants, ramps, and dumpsters (figure 5.4) to benchmark the mapping system relating to correctness of the overall map. That way it can be seen if any severe mapping errors occur in terms of accurately matching point clouds and insert them into a map. The navigation system however finds in this two traversable layers to navigate to due to the big ramp in the middle. The second created Gazebo world is a moon environment (see Figure 5.5) which features a rough terrain and a limited number of external features which makes the mapping process challenging due to a low amount of features which leads to a lower accurency for the pointcloud matching process and therefore for the overall mapping process. Additionally the terrain classification can also be evaluated on rough and flat terrain.

Both worlds are limited in size by walls or other obstacles in order to limit the size of the exploration area for the robot.

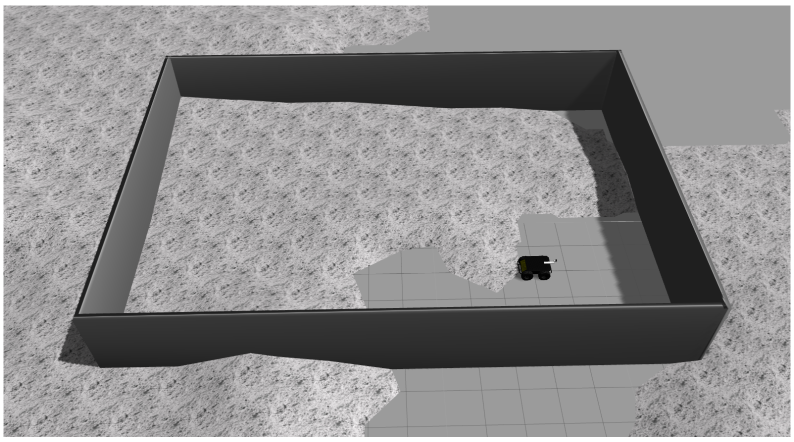

Figure 5.5: Illustration of the simulated rough terrain world in Gazebo

## 5.3 Preprocessing

Before the actual mapping procedure is executed, some preprocessing on the sensor data is required. The laser range measurement of the used 2D laser scanner, a Hukuyo UTM 30LX, is preprocessed in a filter chain to limit the minimal and maximal range of the the measurement as well some readings that might be caused by a veiling effect that occurs when edges of the environment are scanned. This is necessary because we rotate the laser and sometimes some parts of the robot are captured by the laser scanner. By setting a minimal range this effect can be avoided. There exists also an own node that just publishes the 2D laser measurements on a different topic when the laser scanner is horizontal which is used by the navigation stack called *move_base* to avoid dynamical changes in the environment. To create a three-dimensional scan of the environment, a pointcloud is created in which each 2D laser measurement is captured and added to it while the laser scanner is rotating. The endposition for the laser movement is always the horizontal line in respect to the robot base so the laser measurements can be used afterwards also for the navigation purposes.

A problem is the localization of the robot in six dimensions (x, y, z, roll,

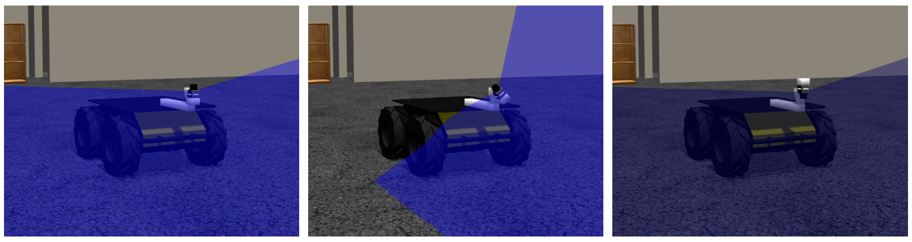
Figure 5.6: Illustration of the laser scanner alignment during one complete sweep

pitch, yaw) inside the environment. To get a more precise estimation of the current robot position, the odometry of the wheel encoders are fused with the IMU measurements. The wheel encoders translate the rotation of the wheels into linear and angular movements. In the short term they are relatively accurate but in the long term the errors add up to a hugh drift. Especially turning in place results in big estimation errors due to the slipping of the wheels for skid-steering robot. The used inertial measurement unit (IMU) can measures the linear acceleration as well as the rotation rate in three-dimensional space but also drifts over time. To improve the overall odometry an extended Kalman filter (EKF)[33] is used which combines the orientation of the IMU with the linear and angular velocities from the wheel encoders.

## 5.4 3D Mapping

This section describes the creation of the graph which is used for the mapping procedure. All stored coordinates in the graph are in reference to the map frame. Each vertex of the graph stores a point cloud of the respective position with it. However, the stored sensor pointclouds are not transformed to the map frame but keep their frame. The first incoming 3D point cloud gets directly stored in the graph as the initial node and the transform between the map and odometry frame stays the same. Additionally after adding this vertex to the graph the position of that vertex is stored in the variable $oldLastVertexEstimate$ (see algorithm 1 for details) which is later used to detect changes in position of the last stored vertex for example after a possible loop closure. This is needed because a loop closure could change the position of the last vertex and if so, this needs to be considered when the transformation between a new measurement/vertex and the last one gets calculated. An illustration of the graph after a creating a map can be seen in figure 5.9 where the white lines on the ground represent the connected edges of the graph. For every additional received 3D point cloud after a laser sweep the following workflow is executed. The current point cloud is stored in the variable *currentPC* and the transformation between the odometry frame and the frame of the point cloud is calculated using the tf-tree in the variable *transOdomPC*. Afterwards all the point clouds of the vertex inside a preconfigured distance to the last stored vertex in the graph are merged and stored in the variable *areaPC*. Because the position of the last vertex could have changed due to a loop closure in the graph, the transformation between the position of the last stored vertex between now and the last cycle is calculated. This is then used to compute the transformation for the current point cloud *transCurrPC* without ICP correction by combining the transformation from before with the estimate of the last stored vertex and the relative odometry between the current and last laser-sweeping-position. Now the current point cloud can be transformed into the map frame by this transformation and is stored as *currentPCMap*. Because now both point clouds, *areaPC* and *currentPCMap* are in the map frame these two can now be matched using the ICP alorithm [28]. After the ICP match the transformation between these two point clouds, *odomICP*, and the covariant matrix *covICP*, are returned. Then a new transformation called *odomCurrentPcICP* is computed by adding *odomICP* to *transCurrPC* which represents the transformation for the current point cloud to the map frame with ICP correction applied to it. Now one can add a new vertex to the graph using the current point cloud as data and *odomCurrentPcICP* as position. Afterwards a check for loop closures using the *findContraints*-function done and the resulting graph gets optimized by the g$^2$o library. In the last step I need to update the transformation between the odom and map frame to correct the odometry errors. This is done by calculating the difference between the position of the current added vertex and *transOdomPC*.

**Algorithm 1:** Graph-based Mapping

```
Data: $num\_3d\_scans$ ... number of received 3D point clouds from laser
      sweeps since the start of the mapping process
Data: $odom$ ... current odometry of the robot
Data: $currentPC$ ... current point cloud created by a sweep at this position
Data: $odom\_frame$ ... odometry frame
Data: $map\_frame$ ... map frame
Data: $graph\_dist$ ... distance in which point clouds of vertices should be
      fused into one big point cloud
1  begin
2      if $num\_3d\_scans = 1$ then
3          $graph.addVertex(currentPC, odom)$
4          $oldLastVertexEstimate \longleftarrow graph.lastVertex.estimate$
5          $lastOdom \longleftarrow odom$
6      end
7      if $num\_3d\_scans > 1$ then
8          $transOdomPC \longleftarrow getTransform(odom\_frame, currentPC.frame)$
9          $areaPC \longleftarrow graph.getMapInDistance(graph\_dist)$
10         $relativeOdom \longleftarrow lastOdom * odom.inverse$
11         $lastVertexLCCorr \longleftarrow$
           $oldLastVertexEstimate * graph.lastVertex.estimate$
12         $transCurrPC \longleftarrow graph.lastVertex.estimate * lastVertexLCCorr$
13         $transCurrPC \longleftarrow transCurrPC * relativeOdom$
14         $currentPCMap \longleftarrow doTransform(currentPC, transCurrPC)$
15         $numICPtries \longleftarrow 3$
16         $\{odomICP, covICP, match\_ratio\} \longleftarrow$
           $matchICP(areaPC, currentPCMap)$
17         $odomCurrentPcICP \longleftarrow transICP * transCurrPC$
18         $graph.addVertex(currentPC, odomCurrentPcICP, covICP.inverse)$
19         $oldLastVertexEstimate \longleftarrow graph.lastVertex.estimate$
20         $graph.findConstraints()$
21         $transformOdomToMap \longleftarrow$
           $graph.lastVertex.estimate * transOdomPC.inverse$
22         $transformMapToOdom \longleftarrow transformOdomToMap.inverse$
23         $sendTransform(map\_frame, odom\_frame, transformMapToOdom)$
24     end
25 end
```

The loop closure detection in the *findConstraints*-function2 tries to match possible and suitable candidates in the area of the last vertex. This is done by selecting vertices in a defined Euclidian distance but aren't directly connected to the last vertex, adding graph related neighbours to it and group them into sets of vertices using the k-nearest-neighbour-algorithm. This is done in a similar way as described in the master thesis by Lassnig[15]. For each set we now calculate the closest vertex to it in reference to the last vertex and check if they maintain a defined minimal graph distance. If so, the pointclouds of both vertexes are transformed into the *map* frame and matched using the ICP algorithm. If the returned match ratio is above a specific threshold, the distance between these 2 vertexes is calculated and an edge is added between them. The match ratio returned from the ICP algorithm is ranging from 0.0 to 1.0 and gives an estimation how good the two point clouds match with each other.

For the matching process between two overlapping point clouds the iterative closest point algorithm (ICP), in particular the ICP libary called libpointmatcher[28], was used. The matching process of this libary can be customized by defining a filter chain and therefore optimizing the matching accurency for specific tasks and usecases. At first the reading point cloud as well as the reference point cloud, which position is fixed during the matching, are processed according to the defined data filters. The used filter chain is illustrated in figure 5.7. We needed these filters for example to remove sensor noise, reduce the density of the point cloud, calculate the normal vector for each point considering their neighbours or removing outliers. In our case the data filters for both point clouds are the same but it is also possible to define different filters for each point cloud. Then the matcher matches each point in the reading point cloud to the closest point in reference point cloud. To prevent outliers from distorting the matching process, two outlier filters are applied. For the error minimization step, point-to-plane error with a returned covariance matrix has been chosen. In the last step the transformation checkers are used to stop the ICP-loop from further iterations. The criteria for stopping the loop is either a reached threshold of maximal iterations or the relative transformations of the two point clouds between the iterations is small enough so that

**Algorithm 2:** Loop closure detection

```
Data: $graph$ ... graph before loop closure
Data: $min\_dist$ ... minimal graph distance to find loop closure candidates
Data: $max\_dist$ ... maximal euclidian distance to find loop closure candidates
Data: $last\_vertex$ ... last vertex added in the mapping algorithm[1]
Data: $min\_dist\_match$ ... minimal number of past vertexes since last_vertex
Data: $match\_success$ ... minimal ICP match ratio for a successful match
Result: $graph$ ... graph after loop closure
1  begin
2      $graph.optimize()$
3      $vertex\_set \longleftarrow graph.findVerticesInRange(min\_dist, max\_dist)$
4      $vertex\_set \longleftarrow graph.addNeighbours(vertex\_set, min\_dist, max\_dist)$
5      $vertex\_groups \longleftarrow graph.findGroups(vertex\_set)$
6      foreach $vertex\_set \in vertex\_groups$ do
7          $closest\_v \longleftarrow graph.findClosestVertex(vertex\_set)$
8          if $last\_vertex.Id - closest\_v.Id > min\_dist\_match$ then
9              $closest\_v\_pc \longleftarrow closest\_v.getPointCloud()$
10             $closest\_v\_estimate \longleftarrow closest\_v.getEstimate()$
11             $closest\_pc\_map \longleftarrow doTransform(closest\_v\_pc, closest\_v\_estimate)$
12             $last\_v\_pc \longleftarrow last\_v.getPointCloud()$
13             $last\_v\_estimate \longleftarrow last\_v.getEstimate()$
14             $last\_pc\_map \longleftarrow doTransform(lastt\_v\_pc, last\_v\_estimate)$
15             $\{icp\_correct, covICP, match\_ratio\} \longleftarrow$
               $matchICP(closest\_pc\_map, last\_pc\_map)$
16             if $match\_ratio > match\_success$ then
17                 $delta\_estimate \longleftarrow$
                   $closest\_v.getEstimate().inverse * last\_v.getEstimate()$
18                 $estimate\_icp \longleftarrow icp\_correct * delta\_estimate$
19                 $graph.addEdge(closest\_v, last\_v, estimate\_icp, covICP.inverse)$
20                 $graph.optimize()$
21             end
22         end
23     end
24 end
```

further iterations won't improve the match anymore.

## 5.5 3D Planning and Terrain Analysis

After adding the point cloud from a 3D laser scan sweep to the pose-graph which represents the 3D map, the navigation process is triggered. The navigation stacks transforms the created point cloud into an octomap [29], which is a 3D occupancy grid. An octomap describes a 3D map as a collection of voxels, which are cubes, down to a predefined lowest level of resolution. Each voxel can represent either free or occupied space, while missing voxels mean unknown space. By knowing these three states for a voxel we are able to plan three-dimensional paths on octomaps as well as define frontier voxels, which are goals for the exploration process.

The octomap server, which handles the octomap creation, receives the point cloud of the current area of the robot. After receiving the point cloud it crops the edges of the point cloud so that a point cloud cuboid of a specific size emerges because only a more dense area is suitable for the terrain analysis. In the beginning of the mapping process there is no map of the current position of the robot available and during the first laser sweeps the ground directly underneath can't be mapped because it is a blind spot for the laser scanner. We presume that the robot always starts its mapping and exploration process from a flat area and therefore we add a small square plane into the octomap which is positioned directly under the robot. This way, in the beginning there is always a voxel-plane right beneath the robot from which the terrain analysis can start. The first step in terrain classification process is to identify the starting voxel beneath the robot. In case there is no voxel directly under the robot and within a given tolerance in height (z-axis), the closest voxel beneath the robot center is chosen as a starting voxel. In some cases this case can occur, when resolution of the octomap is quite low and the density of the point cloud was not high enough for a continous octomap surface. Therefore some "holes" in the octomap surface occur and are only filled, if the density of the point cloud is

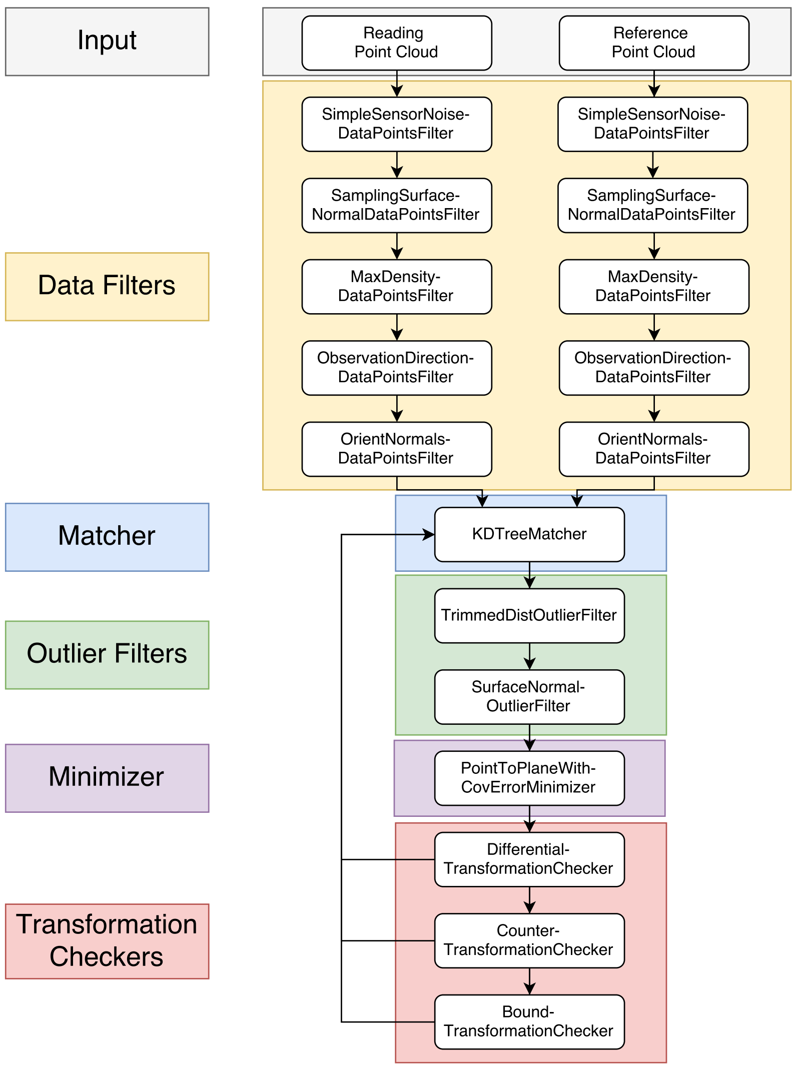
Input
Reading Point Cloud
Reference Point Cloud
Data Filters
SimpleSensorNoise-DataPointsFilter
SimpleSensorNoise-DataPointsFilter
SamplingSurface-NormalDataPointsFilter
SamplingSurface-NormalDataPointsFilter
MaxDensity-DataPointsFilter
MaxDensity-DataPointsFilter
ObservationDirection-DataPointsFilter
ObservationDirection-DataPointsFilter
OrientNormals-DataPointsFilter
OrientNormals-DataPointsFilter
Matcher
KDTreeMatcher
Outlier Filters
TrimmedDistOutlierFilter
SurfaceNormal-OutlierFilter
Minimizer
PointToPlaneWith-CovErrorMinimizer
Transformation Checkers
Differential-TransformationChecker
Counter-TransformationChecker
Bound-TransformationChecker

Figure 5.7: Illustration of the used ICP chain

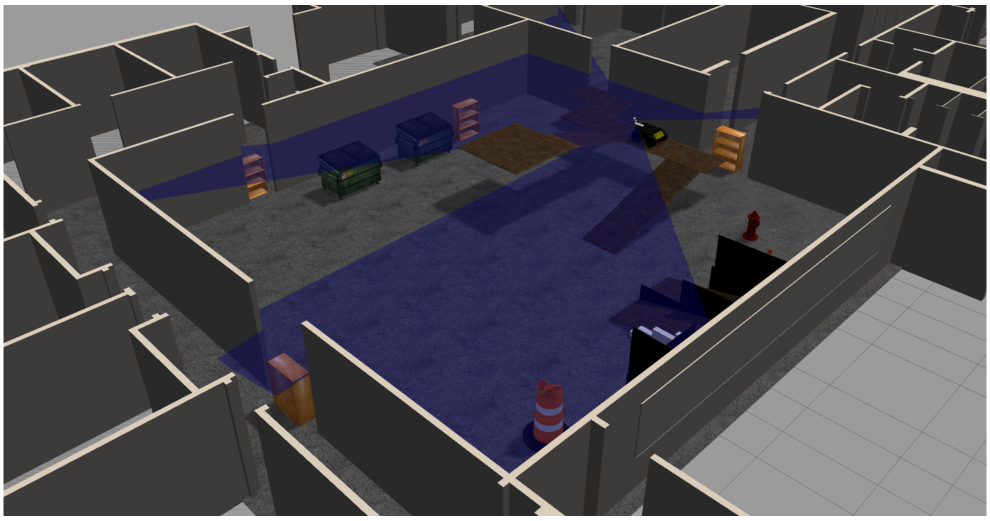

Figure 5.8: Mapping process viewed from the Gazebo simulation

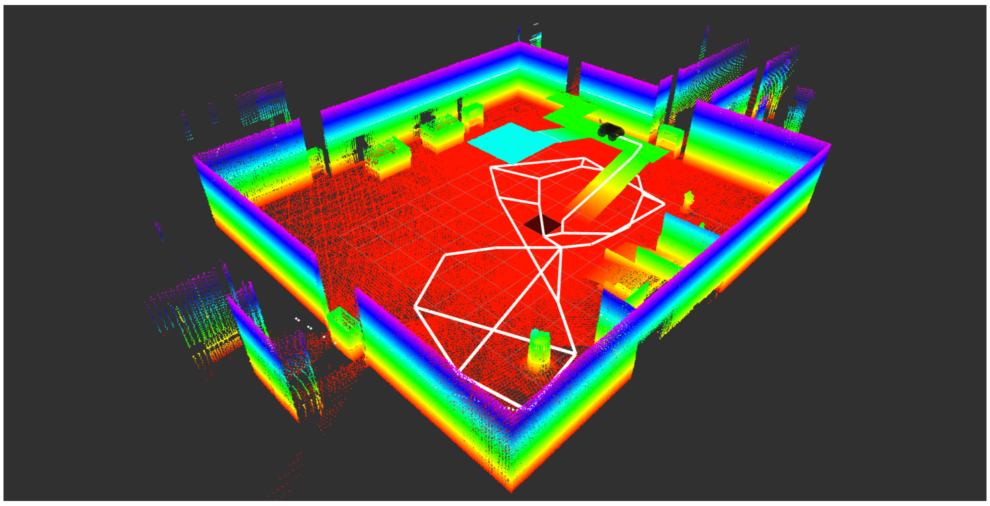

Figure 5.9: Mapping process viewed from rviz. Point cloud is colored accourding to height. The white lines represent the underlying pose-graph which result into this map.

high enough with respect to the octomap resolution. Summarized, a cuboid which can be multiple octomap-layer thick is scanned for traversable voxels and the nearest traversable voxel beneath the robot center is chosen as a starting voxel.

A neighbouring voxel is marked as traversable if the source voxel is also traversable and the height difference between them is within a certain limit so the robot is able to cross this height difference. Additionally this neighbouring voxel has no occupied voxel above within the robot height plus some security margin. This way only surface voxels are considered as traversable when the don't exceed a height difference and the robot doesn't collide with obstacles while driving on this voxel. Because the robot itself is represented as a point, or voxel in the terrain analysis, a kind of inflation layer is added to detected obstacles in the octomap to prevent collisions. This is discussed in detail later in this section.

After the identification of the starting voxel, a flood-fill algorithm is applied to the rest of the octomap to classify voxels as traversable, obstacles or as a potential cliff. Because of limited computing capacity only the four-way flood-fill algorithm is used, which just considers horizontal and vertical voxels as neighbours when seen as a 2D grid but within a certain range in height (z-axis). When a voxel has been classified as traversable, its neighbours get also classified but not when it is either an obstacle or potential cliff. This way the flood-fill algorithm expands within a traversable surface where obstacles, potential cliffs or unknown voxels are acting as a kind of frame to this surface. By also considering a height range in the flood-fill algorithm a traversable multi-level surface can be classified for path planning. Figure 5.14 till 5.16 show this classification process.

A good illustration showing the classification of traversable, obstacle, and cliff voxels is Figure 5.10. The doted arrows indicate the search area from a specific voxel to into a direction. If the height difference between to neighbouring occupied voxels is within a given threshold of traversability then that voxel is classified as traversable. However, if an occupied voxel is found above an otherwise traversable voxel that is within a specific height distance (robot height + security margin),

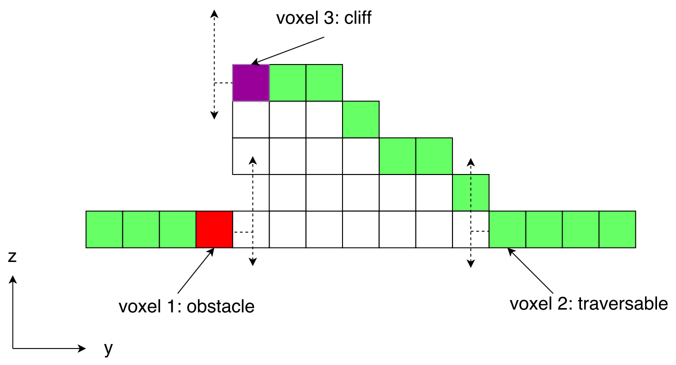

Figure 5.10: Visualization of the classification step to identify traversable, obstacle, and cliff voxels. The white voxels were not classified either because they are no surface voxels or the are not relevant to evaluate because the robot can't reach or drive on them. An example would be a high wall where the lower voxels were classified as obstacles or inflation voxels while the higher voxels stay white because the are not relevant.

it is classified as an obstacle because the robot would hit that voxel otherwise. That is why the doted arrow is longer in the positive z-height and in the negative z-height. Figure 5.11 shows the same setting where voxels inside a specific area around obstacles and cliffs are reclassifed as inflation voxels.

However, there is an additional check for cliff voxels. When inside of the four defined areas around an otherwise traversable voxel, there is not at least one occupied voxel, that voxel is also classified as a cliff. Figure 5.12 shows a simplified 2D representation of this, although the four areas in the implemented algorithm also considers several voxel heights for this check.

If the flood-fill algorithm doesn't find more traversable voxels than a given threshold, then the current starting voxel is discarded and a new one is selected and the flood-fill is performed again. This is repeated until no suitable occupied voxel beneath the robot can be found. This behaviour ensures that a starting point, if available, is chosen that has enough traversable voxels so that a terrain analysis is practicable.

If the flood-fill has been performed an inflation or padding layer is

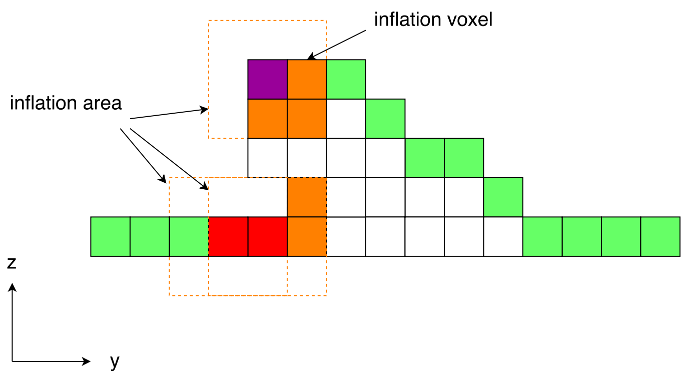

Figure 5.11: Visualization of the classification step after the inflation voxels are added.

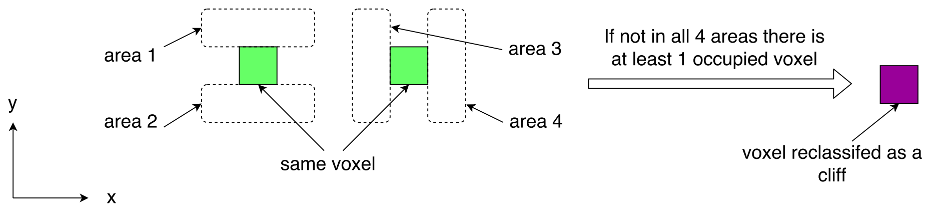
area 1
area 2
area 3
area 4
same voxel
If not in all 4 areas there is at least 1 occupied voxel
voxel reclassifed as a cliff
y
x

Figure 5.12: Visualization of an additional cliff check.

added to the classified traversable octomap surface (see Figure 5.11). This is done by reclassifing all traversable voxels that are inside a defined cuboid of an obstacle- or potential-cliff-voxel, with that voxel as a center of the cuboid. In rare cases the original starting voxel could be inside this padding area and if so, a new starting voxel beneath the robot has to be chosen that is still traversable. This is required because we have to apply a second flood-fill procedure to detect frontier voxels for the exploration process.

Now the second flood-fill procedure is performed. But now only frontier voxels are classified and it's coordinates are stored in a list. If starting from a traversable voxel, no occupied voxel has been found in one of the 4 neighbourhood directions (different heights are considered in one direction) used in the flood-fill algorithmm, the voxel in this direction is a potential-frontier-voxel. Now another step in that direction is made from the potential-frontier-voxel, and the number of occupied voxels in this 4-way neighbourhood is counted. If there are no occupied voxels, then the potential-frontier-voxel is actually a frontier-voxel, otherwise it stays a traversable voxel. This is illustrated in Figure 5.13. Next, all detected frontier voxels are sorted according to a cost function which calculates the cost by Euclidian distance between the frontier and the robot plus an addition with a weighted angle: $cost = distance + angle * weight$. The angle is calculated between the current yaw-angle the robot is facing and the yaw-angle of the frontier in respect to the robot orientation. This cost function is used to prefer frontier voxels that aren't too far away and are oriented roughly in front of the robot. That way the robot doesn't move back and forth if the nearest frontier is alternating between the front and the back of the robot.

Because of time constraints not the whole available octomap gets classified in the flood-fill algorithm just an area within a given range of the robot. Only when no frontiers are found in this area, the floodfill algorithm is called a second time, classifies the whole octomap and is searching for frontier voxels. If a frontier has been found, the next flood-fill procedure is executed in the area around the robot, because it is likely if the robot moved to one frontier, that there are also other frontiers in its proximity. Assuming a corridor the robot explores, it is

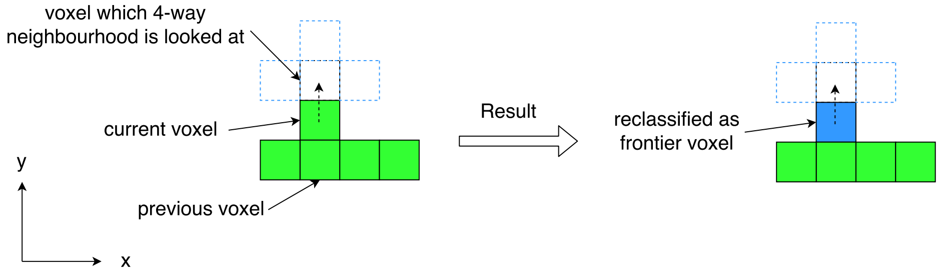
voxel which 4-way neighbourhood is looked at
current voxel
previous voxel
y
x
Result
reclassified as frontier voxel

Figure 5.13: Visualization of an frontier voxel check.

very likely that frontiers will be found further down the corridor and therefore not the whole map needs to be searched for frontiers but only the current area. When the robot reaches the end of the corridor and doesn't find any frontiers, the complete map is considered for the frontier-search.

After selecting the frontier with the lowest costs, a path between the robot center and the selected frontier voxel is planned so that the robot can move close to its position. This is described in the Section 5.5.1. The different terrain classification steps are illustrated in Figure 5.14 till Figure 5.19 where the full octomap is classified. Additionally the different terrain analysis steps are also described in algorithm 3. Figure 5.15 and 5.16 show the expansion of the first classification while in Figure 5.16 the inflation (or padding) area is added around the obstacles and cliffs to prevent the robot of driving too near to them. The second classification step is illustrated in Figure 5.18 till 5.19 where in the last image it can be seen, that there are traversable voxels along the big ramp so the robot would be able to navigate to the highest plateau. Frontier voxels on the ground as well as on a part of the ramp have been found. However, the frontiers on the ground are much nearer than the ones on the ramp which led to a planned path (in yellow) to a frontier on the ground. Traversing the entire octomap for terrain classification is very time consuming and in most cases not really necessary because in many cases an unexplored area can be found near the robot. For this reason we try to traverse just a limited area around the robot and if we don't find any frontier voxels, the entire octomap is scanned for it. Such a limited search can be seen in

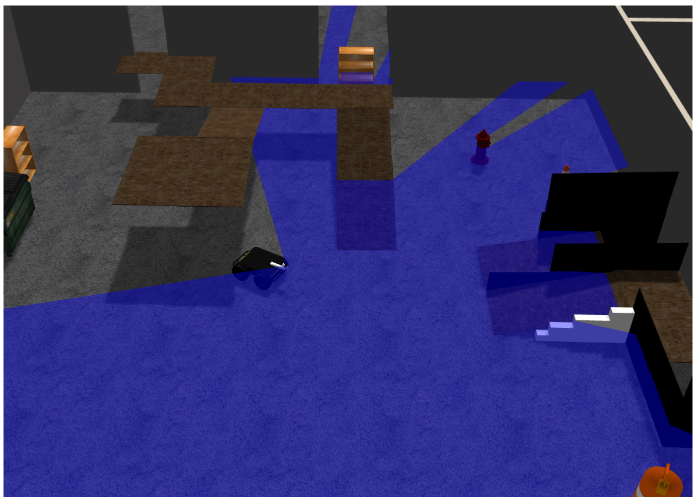

Figure 5.14: Husky robot in the ramp world viewed in Gazebo during the terrain classification step

Figure 5.20. An overview of the terrain classification process can be seen in algorithm 3.

### 5.5.1 Global Planning

During the terrain classification of the first flood-fill procedure, a graph for path planning is constructed. If the neighbour of a traversable voxel is also traversable, a vertex for this neighbour is added to the graph and this two vertices are connected by an edge where the edge cost depends on the height difference of these two vertices. After the flood-fill is finished, the robot padding is added. For all voxels that were traversable and and get reclassified as a padding voxel, the corresponding vertex in the graph gets updated by changing the stored type of the vertex from traversable to padding. For path planning on this graph the A*-algorithm is used with the Euclidean distance as a heuristic. And to avoid vertices with type padding a cost of infinity is returned. That way only paths along vertices with type traversable result in a valid path.

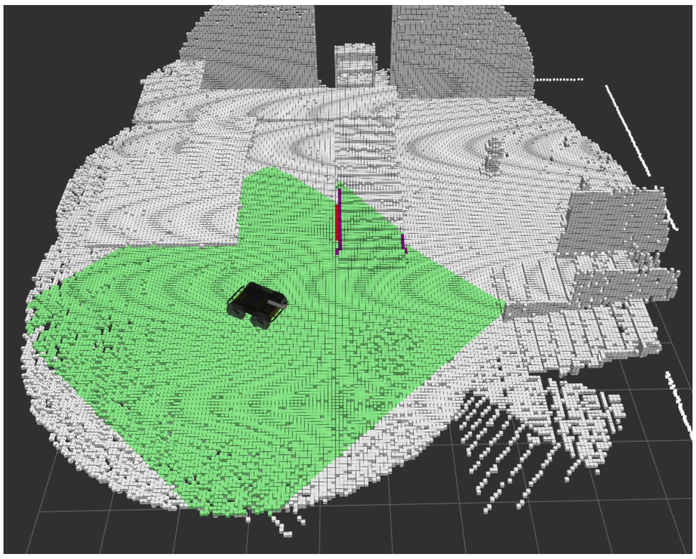

Figure 5.15: Beginning of the flood-fill algorithm for the terrain classification process

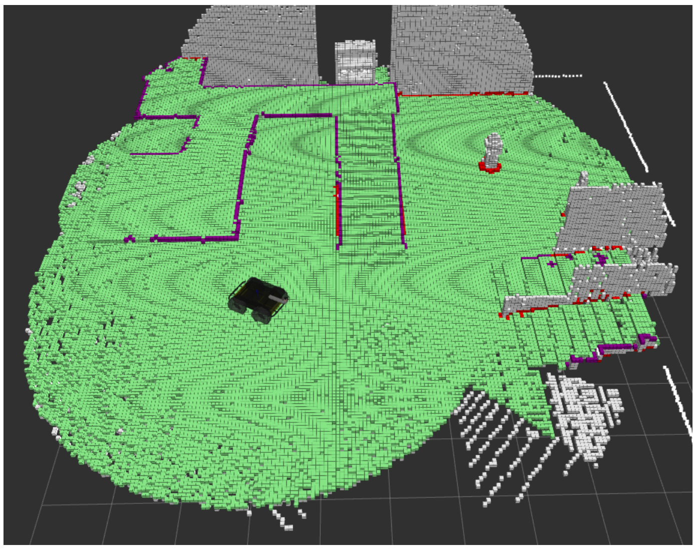

Figure 5.16: The flood-fill algorithm classified voxels as traversable (green), obstacle (red) or potential cliff (purple)

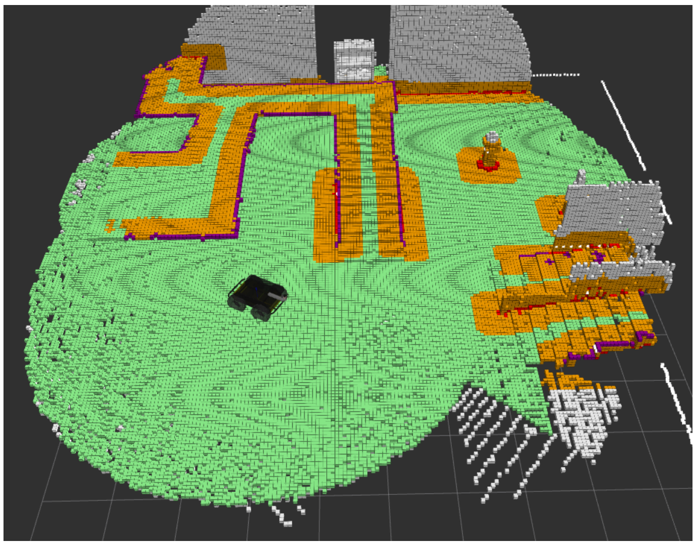

Figure 5.17: Next the robot padding/inflation-layer (orange) is added around obstacles and potential cliffs

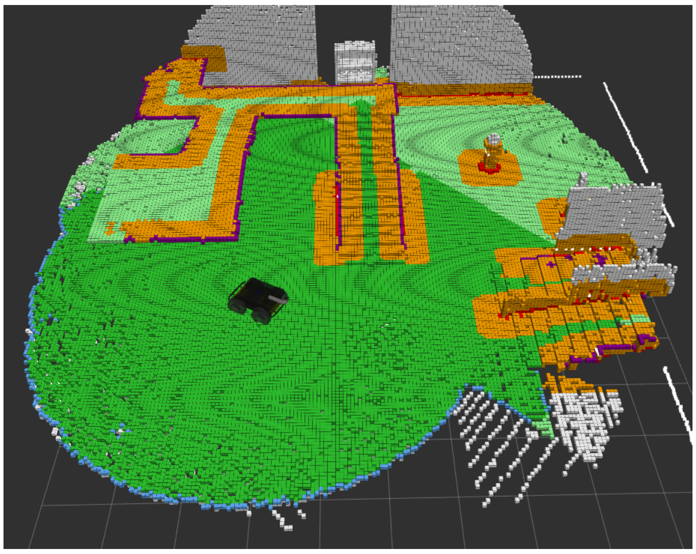

Figure 5.18: The second flood-fill identifies the frontier voxels (blue)

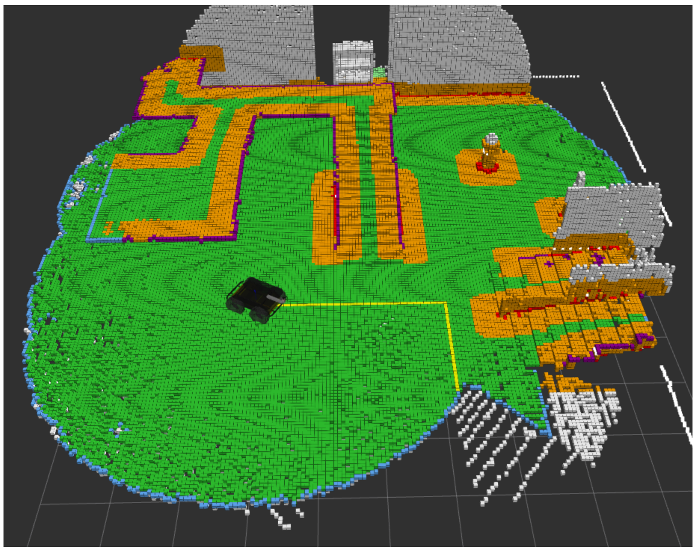

Figure 5.19: After successful classification, the most suitable frontier voxel is selected and a path to it is planned (yellow)

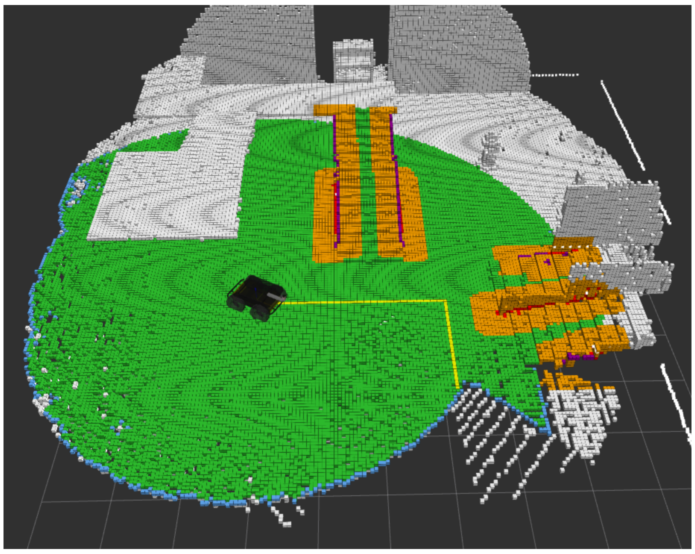

Figure 5.20: Terrain classification limited to the area near the robot

After the path between the robot center and the frontier voxel has been planned, the path is projected to a 2D plane and sent to the move_base software module which handles the rest of the navigation task. This down-projection is possible because the path is located along a closed surface. The move_base gets the calculated global path as well as a local costmap of the area around the robot as input and moves the robot near the frontier location. The global path is shortened a bit because the robot shouldn't move too close to the frontier voxel but should remain a distance between it and the robot. This way it is very likely that this area gets mapped when the robot reaches this posyition and a new laser sweep is started.

### 5.5.2 Local Planning

For local planning only a very small area of the octomap arround the robot is considered. The octomap gets classified like described in section 5.5 and is projected into a 2D costmap. Therefore the costmap of the local planner is equal to the costmap of a 2.5D heightmap that is projected to it. We assume that given the small local area of the local map it is sufficient that the local planner plans in 2.5D while the global planner is capable to plan in 3D because the robot is bound to the ground surface and has certain limits in terms of slopes that it can transcend. To avoid dynamic obstacles the horizontal laser scans during the movement of the robot are also included into the local costmap. However, the range of the horizontal scans are very limited in size so that only very near obstacles are detected and not for example a traversable ramp in the distance.

**Algorithm 3:** Terrain classification

```
Data: $robot\_position$ ... current robot position in the map frame
Data: $map$ ... octomap of the mapped environment
Data: $local\_dist\_limit$ ... distance limit arround robot for floodfill
Data: $COMPLETE\_MAP$ ... no distance limit for floodfill algorithm
Result: $path$ ... path between the robot and a chosen frontier if it exists
begin
    $addStartingPointGroundPlane(map, robot\_position)$
    $starting\_candidates \longleftarrow getStartingVoxel(map, robot\_position)$
    $changeSearchArea \longleftarrow false$
    $dist\_limit \longleftarrow local\_dist\_limit$
    do
        $success\_floodfill \longleftarrow floodfill(map, starting\_candidates)$
        do
            $\{success\_flood, barriers, graph\} \longleftarrow$
            $floodfill(map, starting\_voxel)$
            if $starting\_candidates = 0$ then
                $cancel \longleftarrow true$
            end
            if $(success\_flood = false) AND (starting\_candidates \neq 0)$ then
                $starting\_candidates.pop\_front()$
            end
            if $cancel = true$ then
                $return$
            end
            $graph \longleftarrow addRobotPadding(map, barriers, graph)$
            $changeStartingPointIfInsidePadding(map, starting\_candidates)$
            $frontiers \longleftarrow searchFrontiers(map, starting\_candidates)$
            if $(frontiers.size = 0) AND (dist\_limit \neq COMPLETE\_MAP)$
            then
                $dist\_limit \longleftarrow COMPLETE\_MAP$
                $changeSearchArea \longleftarrow true$
            end
            if $(frontiers.size = 0) AND (dist\_limit = COMPLETE\_MAP)$
            then
                $return$
            end
            if $(frontiers.size \neq 0) AND (dist\_limit = COMPLETE\_MAP)$
            then
                $dist\_limit \longleftarrow local\_dist\_limit$
                $changeSearchArea \longleftarrow false$
            end
        while $(success\_floodfill = false) AND (cancel == false)$
    while $changeSearchArea = true$
    $planPath(map, starting\_candidates, frontiers)$
end
```

# 6 Implementation Details

This chapter discusses the implementation details of the mapping and navigation concepts discussed in chapter 5. The created software uses Ubuntu 14.04 and the ROS Indigo framework. First we discuss the details of the graph-based mapping and second we examine the terrain classification and its role in the navigation system. An illustration of the different ROS nodes and their interaction with each other over topics, service calls or action commands can be seen in Figure 6.1.

## 6.1 3D Map Generation

The aim of this paragraph is to give a more detailed desciption how the 3D mapping process was implemented which was described in section 5.4. Each laser sweep coresponds to a node in the pose-graph. The wheel odometry is used as a first guess for the ICP matcher which returns a transformation to correct the odometry. For a consistent pose-graph construction we used the $g^2o$ library. The mapping functionality is located in the ROS package called *pose_graph_mapping*. The $g^2o$ library already provides some object types which can be used for simultanious localization and mapping. The graph itself consists of two $g^2o$ provided classes, VertexSE3, and EdgeSE3, which allow a representation in the three-dimensional space (x, y, z, roll, pitch, yaw). Both are a specialization of the Hypergraph class (see Figure 6.2). To store the pointcloud in the graph, we created a custom class type called *Robot3dLaser* which is derived from the $g^2o$ provided class *RobotData*. The class *Robot3dLaser* contains the point cloud of that vertex and the estimate of it as well as the covariance matrix of the ICP match between that one and the point cloud of the previous vertex.

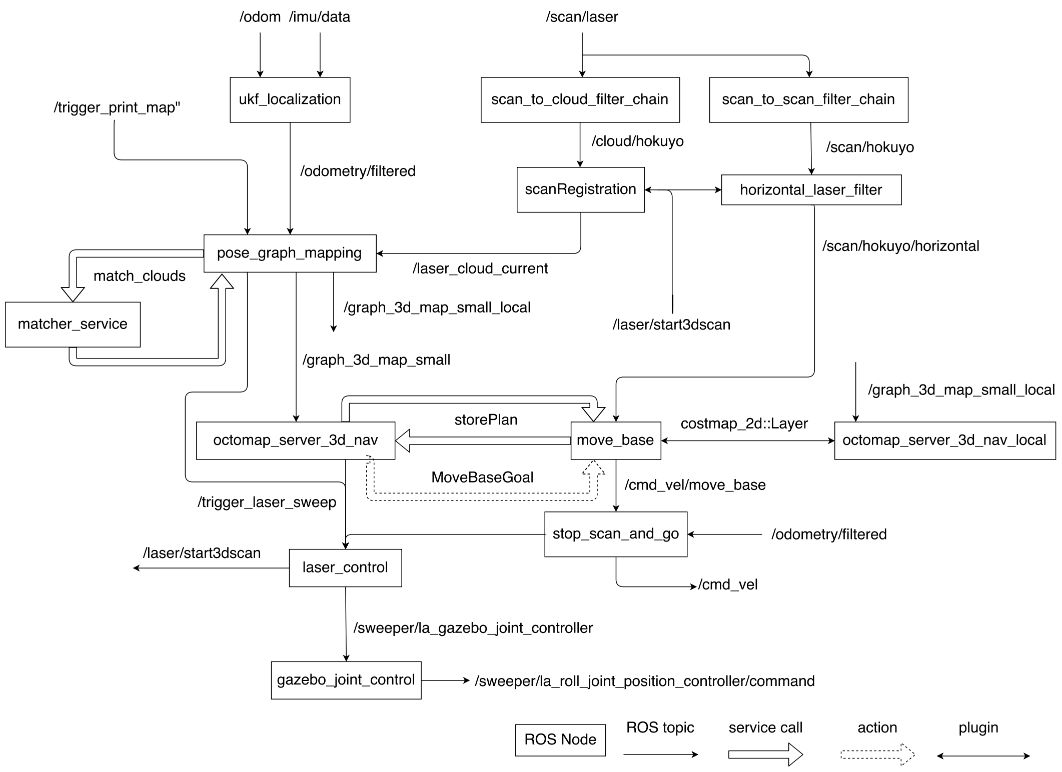

Figure 6.1: Interaction of the different ROS nodes over topics, services or actions.

The node called $pose_{g}raph_{m}apping_{n}ode$ subscribes to point clouds of the type *sensor_msgs*::*PointCloud*2 and to the fused odometry messages from the IMU and wheel encoders of type *nav_msgs*::*Odometry*. The odometry messages are published with 30 Hz. When a pointcloud from a 3D sweep is received, it is transformed into the map frame using the latest odometry at that time and the ICP matcher compares the current point cloud with the point cloud of the last vertex of the graph (both in the map frame). The ICP matcher is located in the package *ethzasl_icp_mapper* and can be called using the ROS service *match_clouds*. This service was modified to return more parameters like the covariance matrix of the match, the match ratio or overlap ratio which is needed by the $pose_{g}raph_{m}apping_{n}ode$. Afterwards the odometry transformation gets corrected by the transformation provided by the ICP and a vertex for the new measurement is added to the graph. The constructed pose-graph can be visualized by markers of type *visualization_msgsMarker* on topic *pose_graph*.

## 6.2 Terrain Classification

The terrain classification consists of 2 different nodes, one is called *octomap_server_3dnav_node* and the other *octomap_server_3dnav_local_node*. The first node handles the overall map, it can classify voxels of a part of the map (local area) as well as the whole map if no frontiers have been found in the local area. This node also computes the path from the robot position to a chosen frontier and sends it together with the goal positon to the *move_base*, which handles the rest of the navigation task.

### 6.2.1 Local Planner and Navigation

The node *octomap_server_3dnav_local_node* is needed to calculate a part of the local costmap used by the *move_base*. It doesn't has a mapping purpose, it only receives a local part of the map and each time a laser sweep was done but doesn't store it. The next time it receives a new

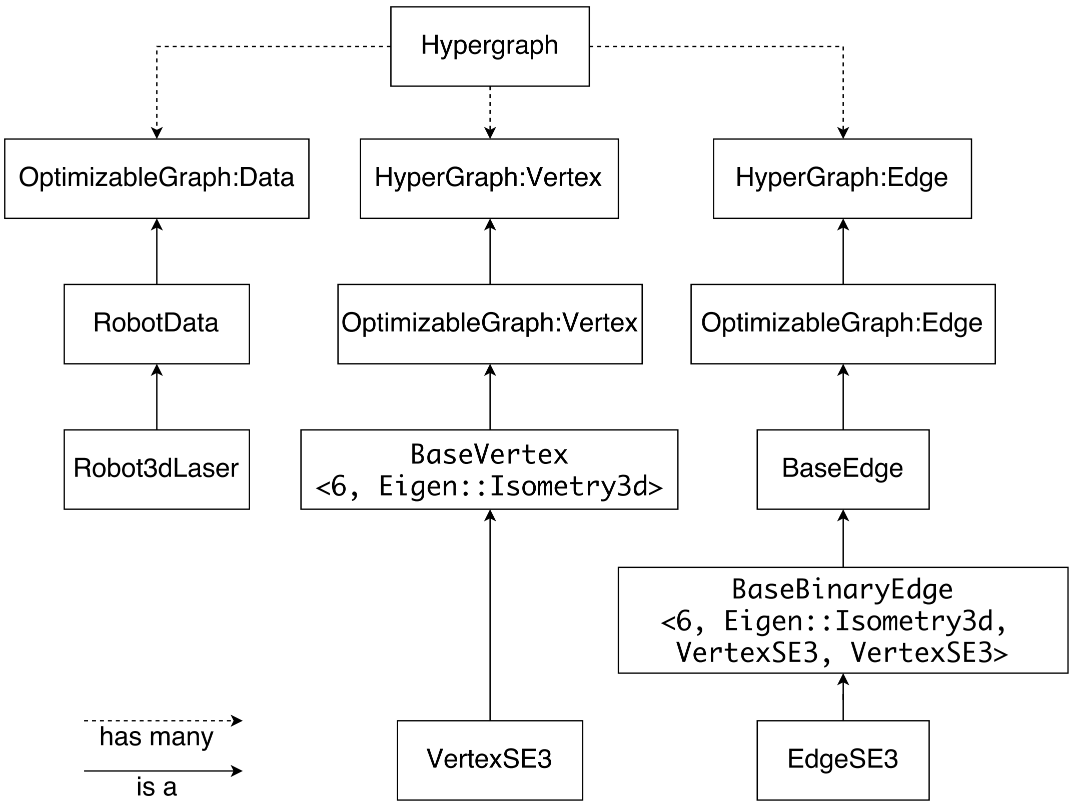

Figure 6.2: Class diagram of the used g$^2$o data types. Adapted from [15].

part of the map, the old one gets deleted. The current Octomap is only needed to construct a local costmap layer of a specific size which can then be integrated into the *move_base*. The terrain classification is done similar as in the node *octomap_server_3dnav_node* but its classified voxels are projected down to a 2D costmap layer.

The costmap in the *move_base* consists of 3 different layers. The first layer is of type *terrain_costmap_2d_layers*::*TerrainLayer* from the package *octomap_server_3dnav_local* to consider classified obstacle/cliff-voxels from the Octomap. The costmap of the robot is in a rolling window mode but the obstacles in this layer are only updated at every laser sweep. Therefore the obstacles in this layer are shifted between the cells inside this costmap layer when the robot moves to keep the costmap consistent.

The *move_base* sends the command velocities for the robot on the topic */cmd_vel/move_base* but the robot listens for this commands on the topic */cmd_vel*. The *stop_scan_and_go_node* forwards the command velocities sent from the *move_base* to the robot except when a specific threshold has been reached in terms of traveled distance and yaw-angle since the last laser sweep. The distance of the robot since the last laser sweep as well as the covered angular movement (yaw) of the robot is summed up and if a specific threshold is reached, this node stops the forwarding of the velocities and instead sends a command to stop all wheel movement to make a laser sweep. After the laser sweep, the velocities of the *move_base* are forwarded again. This is necessary in case the robot drives to a distant frontier and to ensure a correct localization, a laser sweep has to done and inserted into the pose-graph which corrects the position of the robot according to the ICP match. Otherwise, when driving a long distance without correcting the localization via ICP, the odometry error could get too big for the ICP matcher to correct when the robot finally arrives at the frontier position. This would lead to a wrong localization and map representation.

The second layer is of the type *costmap_2d*::*ObstacleLayer* which represents obstacles from the horizontal laser scans. The scans are limited to a short range by a laser-scan-filter-node so that dynamic obstacles

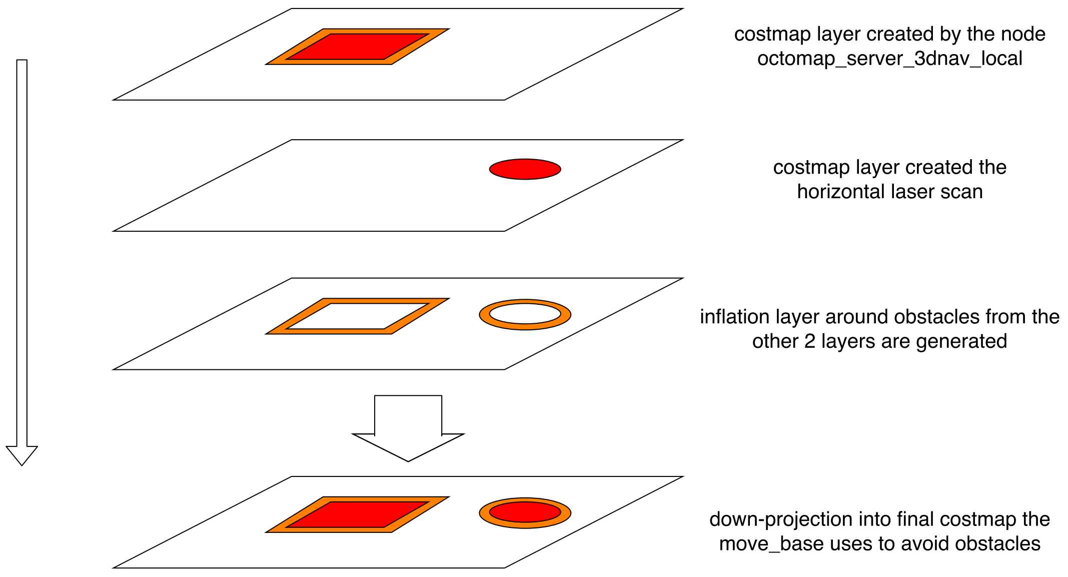
costmap layer created by the node octomap_server_3dnav_local
costmap layer created the horizontal laser scan
inflation layer around obstacles from the other 2 layers are generated
down-projection into final costmap the move_base uses to avoid obstacles

Figure 6.3: Illustration of the used costmap layers.

in a close area can be detected and avoided. The range limitation is needed because otherwise when a path up a hill (or ramp) is planned the hill could be classified as an obstacle and the robot wouldn't drive there although the terrain classification modules classified this hill as traversable. Because the laserscanner is mounted relatively high, and the maximal climbing rate of the husky base is 45° this approach seems to work fine.

In the third layer, which is of type $costmap\_2d::InflationLayer$, adds an inflation area around the added obstacles in the costmap. In illustration of the used costmap layers and the down-projection into the final one used by the *move_base* can be seen in Figure 6.3.

### 6.2.2 Global Planner

In the package *octomap_server_3dnav* the class *OctomapServer* is responsible for creating an Octomap from the incoming point clouds after a laser sweep except during laser sweeps issued from the *stop_scan_and_go_node* because in this case the robot is already in the process of moving to a frontier and no new frontiers need to be chosen. After the Octomap has been updated be the new measurements, the *TerrainClassification* class handles the classification of the voxels as described in section 5.5. During the classification process another graph is constructed inside the *NavigationGraph* class which is used for A* path planning. Each traversable voxels in the first classification step represents a vertex in this navigation-graph and edges connect neigbouring traversable vertices. Inside such a navigation-vertex it also stores that it is a traversable voxel. Later in the terrain classification process, when some traversable voxels near obstacles are reclassified as padding voxels, the corresponding vertices in the navigation-graph change the state of this variable to *padding*. When the A* algorithm visits such a padding-vertex, it returns infinite costs so they are avoided and only traversable vertices return a resonable cost. An illustration for such a navigation-graph is Figure 6.4. When a frontier has been chosen, the A* algorithm plans a path between the robot-position-vertex and the frontier-vertex of the navigation-graph. When a plan for some reason couldn't be found, another frontier is chosen for path planning. When no frontiers have been detected, the exploration is finished. When a plan has been found, it is sent to the *move_base* by a ROS service called *storePlan*. To ensure the robot maps the frontier, the goal for the navigation has to have a certain offset, otherwise the robot would drive directly to the frontier. Therefore, not the whole path is sent but it is trimmed at the end to provide enough space to map the frontier when the robot reached the frontier. The end-position of this trimmed down path is the goal-position which is then sent as a goal to the *move_base* with the goal-orientation facing to the frontier position. A sequence diagram of the interaction between the pose-graph-node as well as the classification-nodes and the move-base at the beginning of the exploration or when a frontier has been reached and a point cloud of a laser sweep is created, is shown in Figure 6.5.

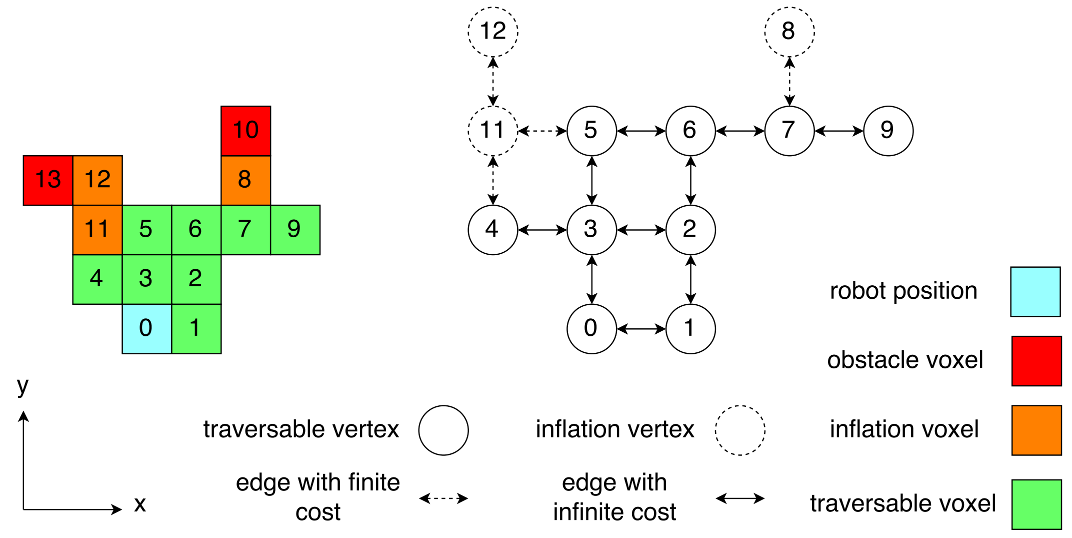

Figure 6.4: Illustration of a navigation graph created from classified voxels.

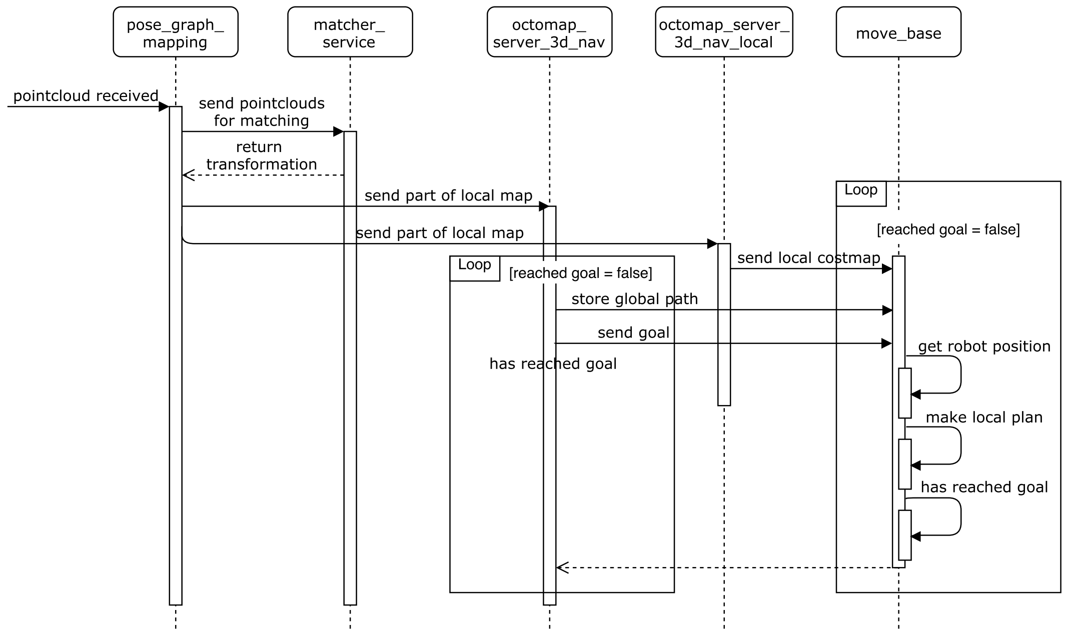

Figure 6.5: Illustration of the procedure after receiving a point cloud from a 3d laser sweep when a new frontier needs to be chosen.

# 7 Evaluation

In this chapter we present an evaluation of the mapping and terrain classification algorithm. We let the robot autonomously explore 2 sets of different maps, one urban scenario and the other in a rough terrain in the Gazebo simulation. Afterwards we compare the sets of created and classified maps in terms of accuracy and performance.Finally we will discuss the qualitative and quantitative results.

## 7.1 Terrain Analysis

To evaluate the mapping and exploration system we let it explore 2 different maps, one indoor (see Figure 7.1) and one outdoor (see Figure 7.7) for 10 times and compare the results in terms of map quality and runtime with each other.

## 7.2 Indoor Scenario

The table 7.1 summarizes 10 simulation runs in the indoor map. Apart from run number 5, 7, and 8 the exploration and mapping algorithm worked well. A detailed description why these 3 runs failed is explained below. With *runtime*, the time between the start and end of the autonomous exploration is meant. The column *Success* indicates how good a run was. If there is a "$\checkmark$", that means that the run was successful and the map looked as expected while a "$\sim$" means that although the run was successful, the resulting map or classifcation could have been better. A "$x$" means that the simulation failed and stopped either from its own or we saw that there were some mismatches in the map and we stopped the exploration manually. The next columns specify how many local classifcations as well as global classifications of the map were necessary during the exploration and finally the time used for both, local and global classification are displayed.

In table 7.1 it can be seen that the first exploration run took 11 minutes and 27 seconds and explored the environment nicely as illustrated in figure 7.2. For every run the number of local classification runs are given where only a local area with a given radius is classified (in this case 6 meters) as well as the number of classifications of the whole map and the overall time needed for both. The path over the ramp to the top plateau is classified as perfectly traversable as well as below the plateau. Illustration 7.3 shows for example more details of good classifed map, this time from a another run. The area directly under the first ramp is marked as an obstacle because the robot doesn't fit through it regarding the robot height plus some security margin. The 2 smaller ramps are not traversable because the ramps are not wide enough so the robot can turn on the spot on these ramps. For each obstacle or cliff there is a square padding or inflation area around it. However, there is also a plateau over one of these smaller ramps which couldn't be mapped or classified because the robot couldn't see it or reach this area. The a part of the edge of the ramp with the unmapped plateau is even considered as a (potential) cliff and only if a part of that ramp would be traversable and would contain frontiers would result in a mapping of this area and a reclassification to a traversable voxel. The implemented algorithm only explores areas that are adjacent to traversable voxels which could lead to unexplored parts of the map because it may be dangerous for the robot to navigate to it.

Figure 7.4 shows the same view of that run but instead of the classified Octomap, it displays the mapped area as a pointcloud. The pointcloud itself shows that the mapping was quite nice and shows a bit more detail than the Octomap (with a voxel size of 8cm) so we are able to conclude that the mapping process worked quite well in this run. Some unmapped areas can also be seen in this representation like the back of the trash containers on the left in the first image or a missing quarter of the hydrant in the middle of the second image.

| **Run** | **runtime [mm:ss]** | **Success** | **local class.** | **local class. time [mm:ss]** | **global class.** | **global class. time [mm:ss]** | **class. time [mm:ss]** |
|---|---|---|---|---|---|---|---|
| 1 | 11:27 | ✓ | 14 | 01:17 | 3 | 01:29 | 02:46 |
| 2 | 11:03 | ✓ | 16 | 01:17 | 3 | 01:21 | 02:38 |
| 3 | 11:14 | ✓ | 19 | 01:36 | 3 | 01:10 | 02:46 |
| 4 | 12:24 | ✓ | 17 | 01:40 | 4 | 01:45 | 03:25 |
| 5 | 01:15 | x | - | –:– | - | –:– | –:– |
| 6 | 10:02 | ✓ | 18 | 01:14 | 2 | 00:54 | 02:08 |
| 7 | 08:08 | x | 13 | 01:07 | 1 | 00:11 | 01:18 |
| 8 | 15:57 | x | 18 | 01:52 | 5 | 02:23 | 04:15 |
| 9 | 18:21 | $\sim$ | 18 | 02:45 | 6 | 04:30 | 07:15 |
| 10 | 11:41 | ✓ | 16 | 01:37 | 4 | 02:07 | 03:44 |

Table 7.1: Summary of the indoor world simulation runs. *Success* means if the resulting map and terrain classification resulted an acceptable solution (✓ = good, $\sim$ = acceptable, and $x$ = not acceptable) while *local class.* and *global class.* mean the number of local and global terrain classifications that were needed to explore the environment. The columns *local class. time* and *global class. time* represent the used time for the respective classification and *class. time* is the overall time used for both classifications.

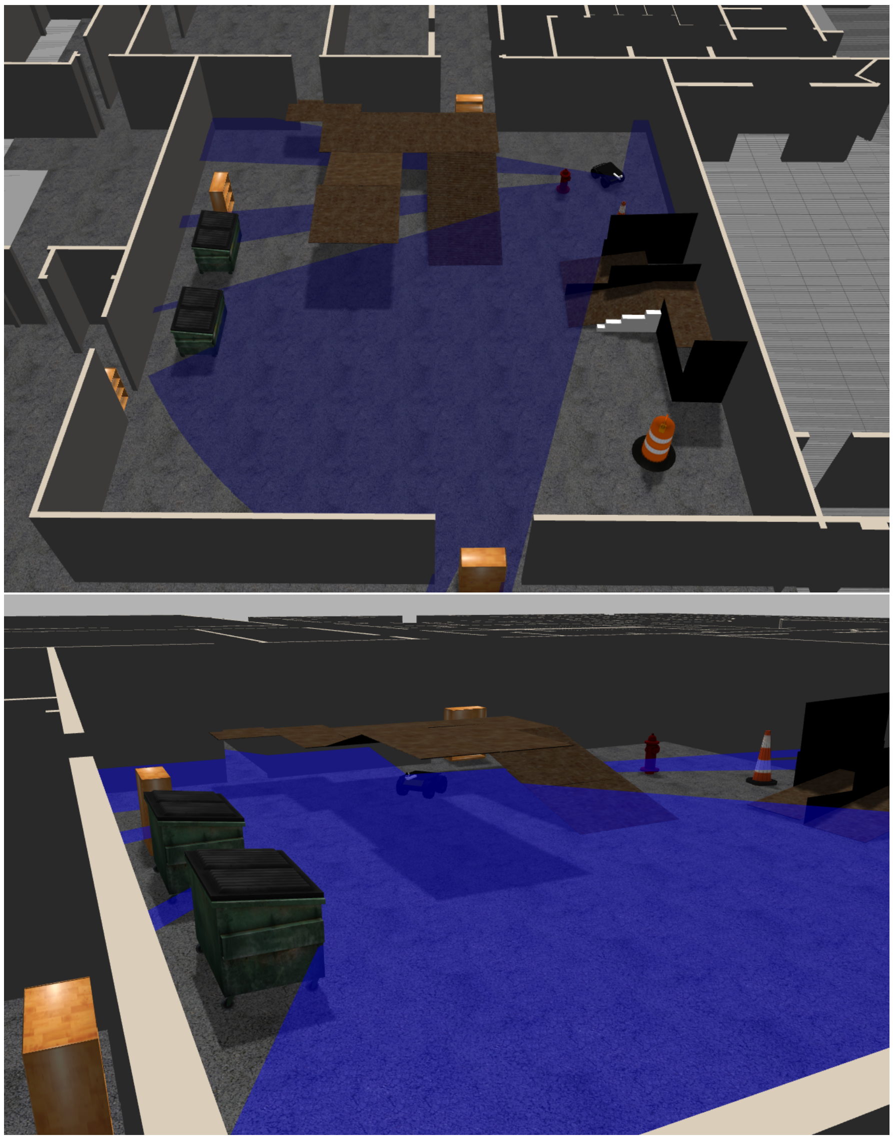
Figure 7.1: Illustration of the indoor world in the Gazebo simulation.

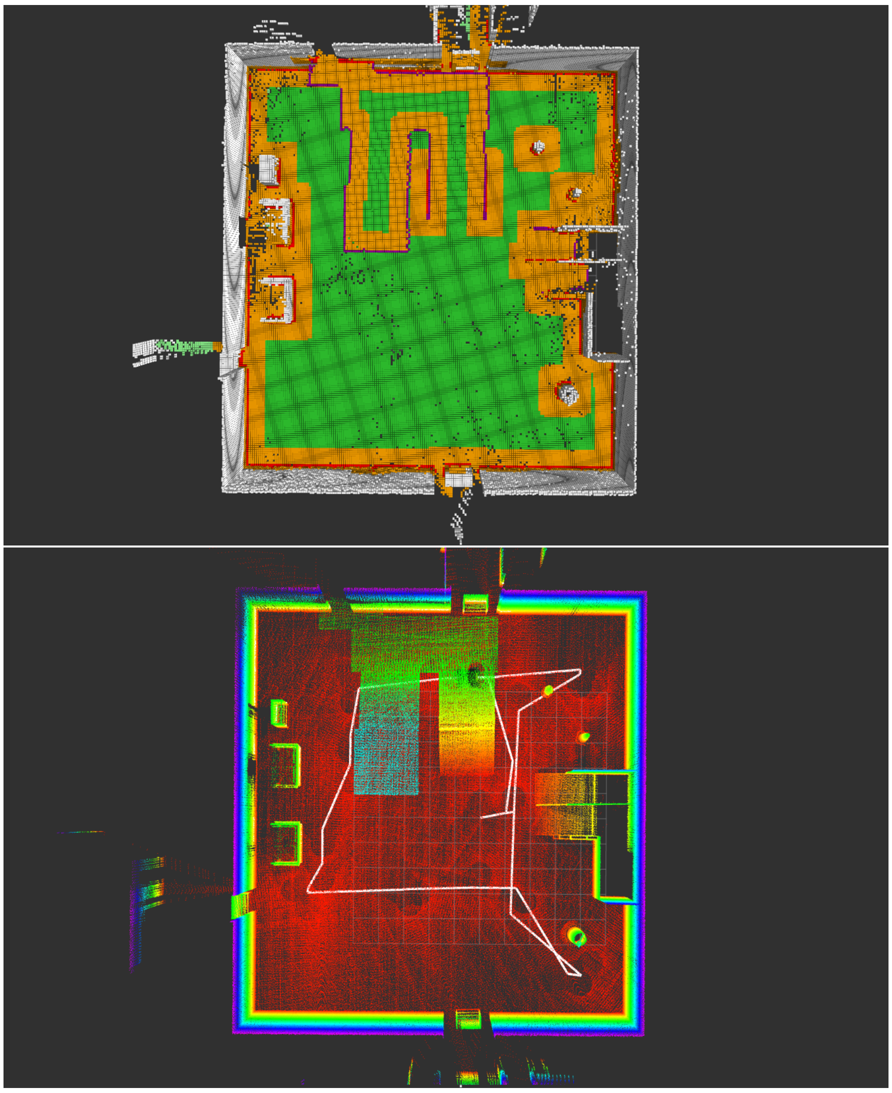

Figure 7.2: Illustration of the 1st evaluation run in the indoor world (view of the classified Octomap and the raw pointcloud).

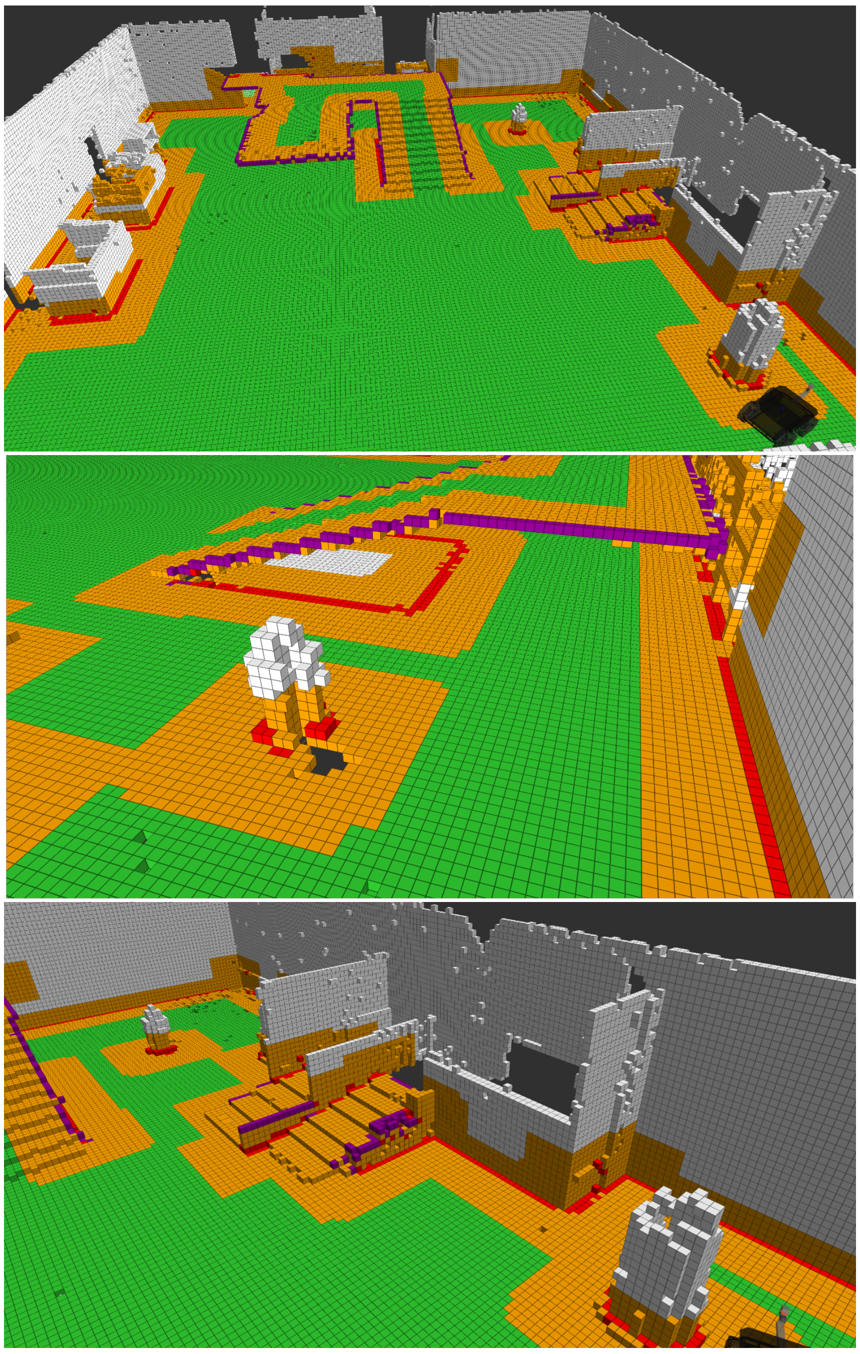

Figure 7.3: More detailed illustrations of a classified Octomap. The top image gives an overview of the map while in the middle image the terrain classification beneath the ramps can be seen. The smaller ramps that weren't traversable are displayed in the lower image.

The rest of the runs marked with a checkmark look similar although not at every run was the ramp classified as perfectly traversable

In run number 5 and 8 an ICP mismatch occured because the odometry error got too big for the ICP matcher to handle when the robot first explored a part of the ramp and then went back down to explore a frontier on the ground floor. This mismatch can be seen in Figure 7.5. The arrows in the figures point to the structures of the map that were mismatched. For run 5 the table 7.1 doesn't show the local and global classifications as well as the needed time for it. This is because a short time after the start of the exploration, it planned a short path onto the ramp and back down and doing so the odometry error got that bad that a ICP mismatched of the ramp happened. We cancelled the run afterwards but forgot to note the missing parameters.

In run 7 the robot failed to plan to the frontier and stopped the exploration while in run 9 (see Figure 7.6) he explored successful the whole map apart from a small piece on the highest plateau of the ramps. The size of the local costmap however was set too big so that there were also obstacle in the local costmap that were behind the ramp and therefore he couldn't find a path to the frontier. Because the mapping and classification was ok, just the last frontier was not classifed to be a cliff I would that the run was somehow ok. If the costmap would have been set smaller it should have been able to follow the path to the plateau.

### 7.2.1 Summary

The exploration of the indoor map was successful in 6 of 10 tries and there are still some improvements to be made to make the implementation more stable. The runtime of all successful runs is ranging between 10 minutes 2 seconds and 12 minutes 24 seconds with an average of 11:18. The produced classification is seems to be good enough for the exploration and navigation as long as no invalid ICP matches occur which can mess up the localization of the robot. To reduce the needed classification time a bigger octomap resolution can be chosen or the

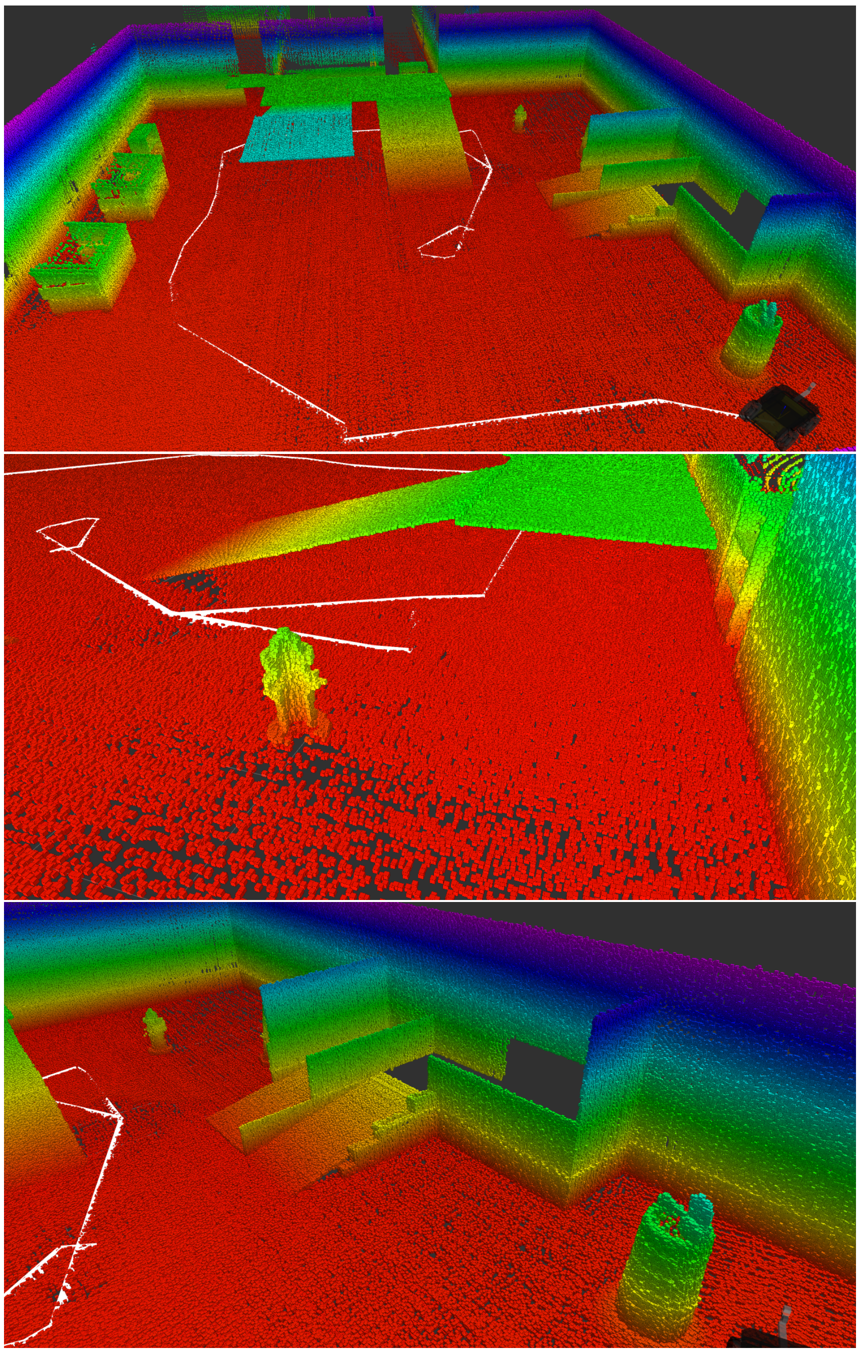

Figure 7.4: More detailed illustrations of the mapped area. These three images show the same setting as in Figure 7.3 but this time instead of the octomap, the point cloud of the created map is visualized.

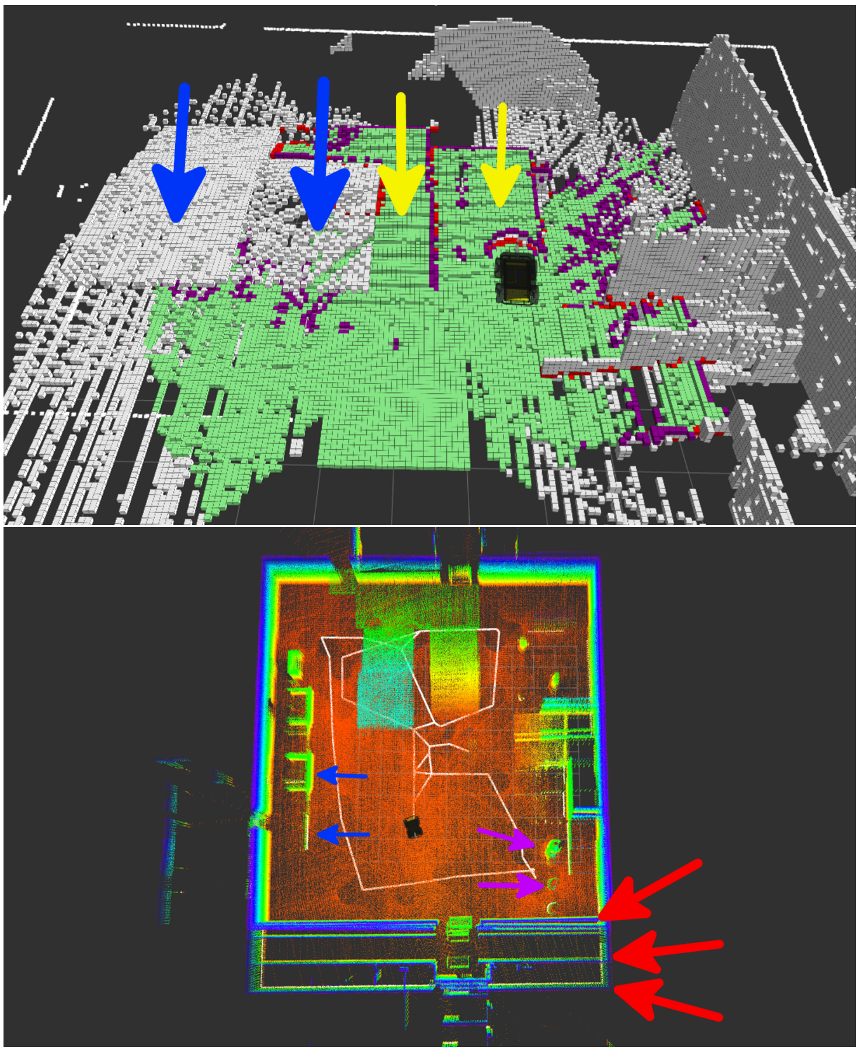

Figure 7.5: Illustration of the ICP mismatch of the 5th (upper image) and 8th (lower image) run. Arrows with the same color indicate areas that were duplicated in the map because of the mismatch

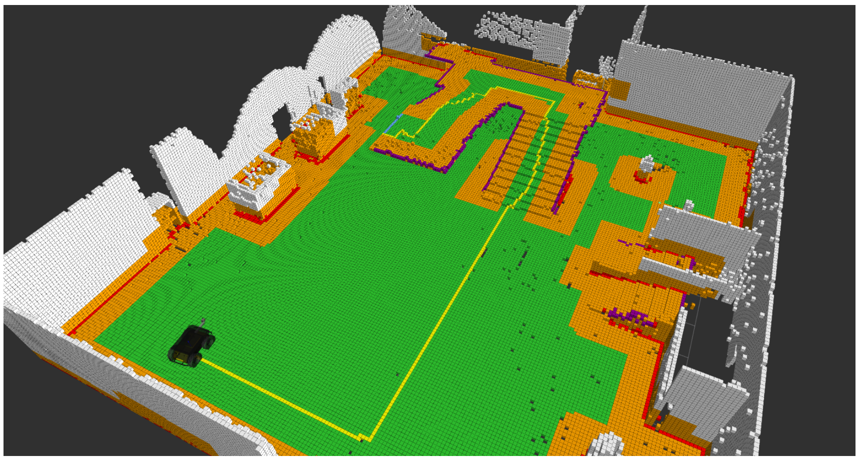

Figure 7.6: Illustration of the 9th run where the robot planned a path over the ramps to the highest plateau.

perhaps the algorithm can be further improved so that perhaps it is also possible to classify frontiers during the first classification run so no separate frontier classification run is needed in the algorithm.

## 7.3 Outdoor Scenario

In this section we evalute the mapping and exploration of a map with rough terrain, a part of the Apollo landing site to be exact. To limit the area which should be explored and be able to compare the different runs, four walls limit the exploration space.

As it can be seen in table 7.2 there occured only one failed run on this map because and that is because it only explored a small area of the map. The reason for this is because the cliff detection to ensure that the robot stays away from areas that may be a cliff. Because of the small hills, the laser scanner doesn't see voxels behind the hills but some other voxels from the other bigger hill behind it, the terrain classification algorithm classifies these voxels adjacent to the hidden voxels as a potential cliff and prevents the robot from driving there. While exploring other parts of the map, some of these voxels are

| **Run** | **runtime [mm:ss]** | **Success** | **local class.** | **local class. time [mm:ss]** | **global class.** | **global class. time [mm:ss]** | **class. time [mm:ss]** |
|---|---|---|---|---|---|---|---|
| 1 | 08:43 | ✓ | 13 | 00:34 | 2 | 00:13 | 00:47 |
| 2 | 07:22 | ✓ | 11 | 00:24 | 3 | 00:23 | 00:48 |
| 3 | 07:39 | ∼ | 8 | 00:12 | 2 | 00:11 | 00:23 |
| 4 | 06:23 | ✓ | 11 | 00:20 | 2 | 00:14 | 00:34 |
| 5 | 05:31 | ✓ | 10 | 00:20 | 1 | 00:08 | 00:28 |
| 6 | 06:47 | ∼ | 15 | 00:28 | 2 | 00:10 | 00:38 |
| 7 | 05:34 | ∼ | 14 | 00:32 | 2 | 00:19 | 00:51 |
| 8 | 07:20 | ∼ | 9 | 00:23 | 2 | 00:24 | 00:47 |
| 9 | 05:31 | x | 7 | 00:10 | 1 | 00:03 | 00:13 |
| 10 | 05:25 | ✓ | 12 | 00:26 | 3 | 00:29 | 00:55 |

Table 7.2: Summary of the indoor world simulation runs. *Success* means if the resulting map and terrain classification resulted an acceptable solution (✓ = good, ∼ = acceptable and *x* = not acceptable) while *local class.* and *global class.* mean the number of local and global terrain classifications that were needed to explore the environment. The columns *local class. time* and *global class. time* represent the used time for the respective classification and *class. time* is the overall time used for both classifications.

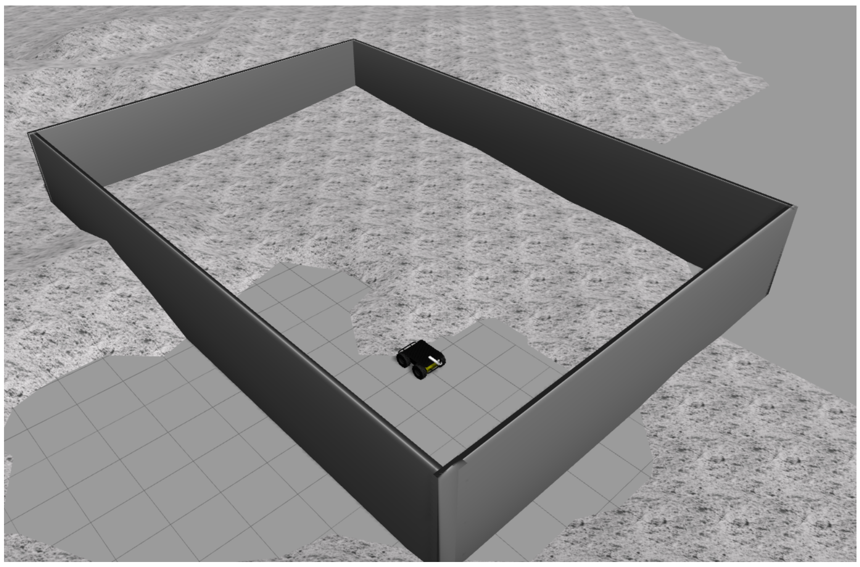

Figure 7.7: Illustration of the Gazebo simulation of the outdoor world.

reclassified as a traversable voxels so other parts can be explored leading to bigger explored map. However, in run 9 the area was quite small while the runs 3, 6, 7, and 8 were ok but they could be better explored and classified.

In Figure 7.7 the Gazebo simulation is illustrated while in Figure 7.8 an example for a good classification run is shown. A not so good run is depicted in Figure 7.9 where some parts terrain parts weren't classified as traversable although the robot would have been able to drive there..

### 7.3.1 Summary

The exploration of the outdoor map was successful in 5 of 10 tries and 4 of 10 were successful although they showed bigger unexplored areas that should have been traversable. Only one run showed such a small explored area that we classified this run as failed. To improve the size of the explored area the part of the algorithm that detects cliffs

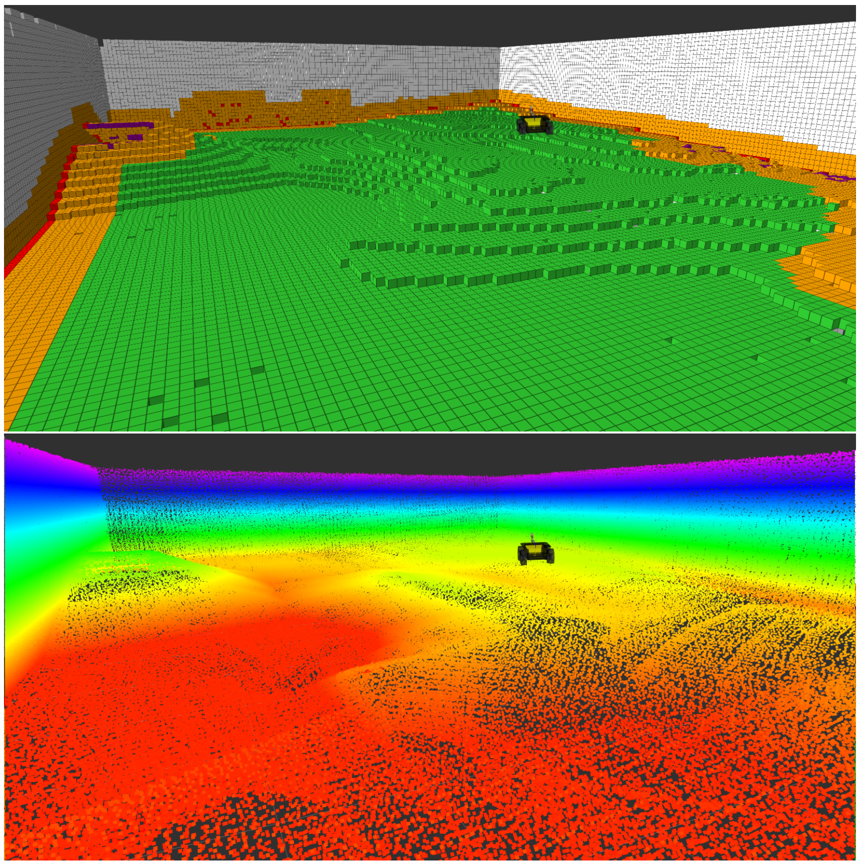

Figure 7.8: Illustration of the 2nd run in the outdoor world. Upper image shows the classified octomap while the lower image displays the corresponding point cloud.

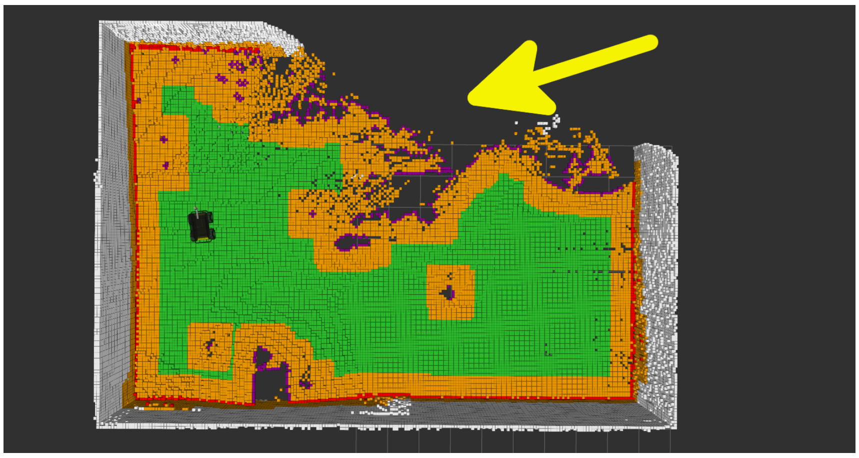

Figure 7.9: Illustration of the 3rd run of the outdoor world where some parts weren't correctly classified. The yellow arrow indicates the area that is also traversable but wasn't classified as such.

needs to be modified or a risk parameter can be used which can be set according to the terrain that should be mapped.

## 7.4 Real environment

We also made a short test on a real husky robot (see Figure 7.12) and mapped a parking lot at first manually by remotly controlling the robot and later we let it autonomously explore the area. In Figure 7.10 the point cloud of the manually created map can be seen and in Figure 7.11 the octomap of a part of parking lot that was autonomously explored is illustrated.

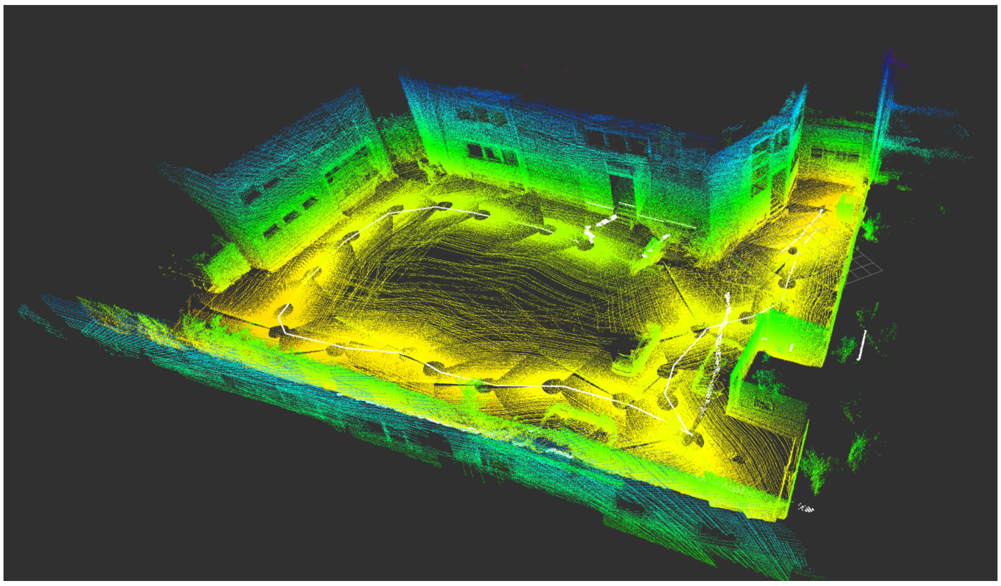

Figure 7.10: Illustration of the pointcloud of a measurement taken in a parking lot.

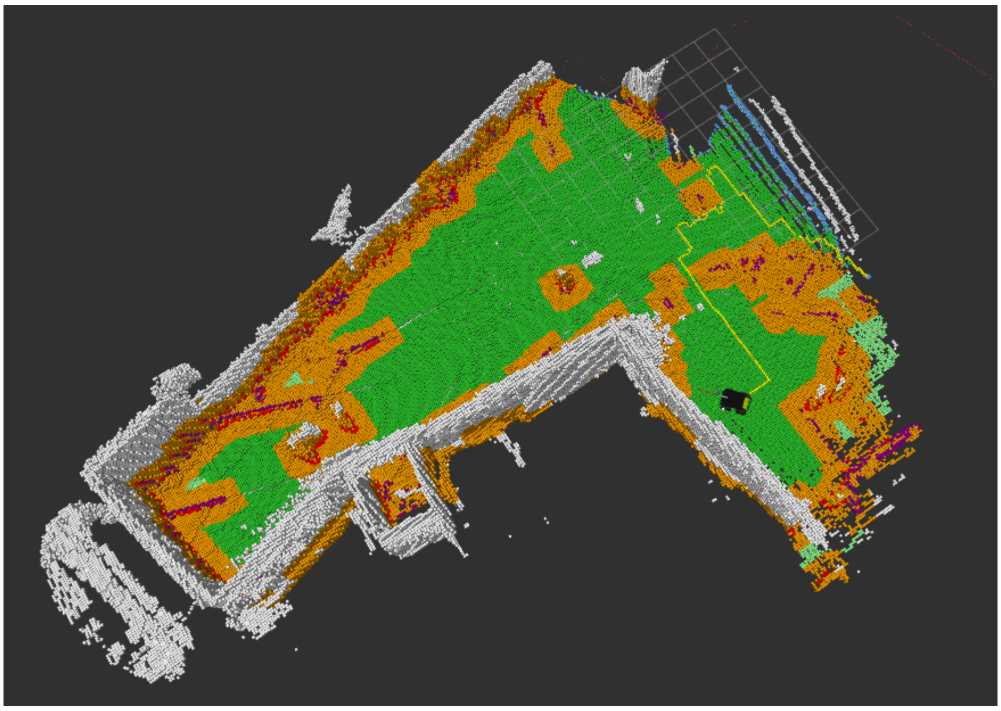

Figure 7.11: Illustration of the octomap created in an autonomous exploration in a parking lot.

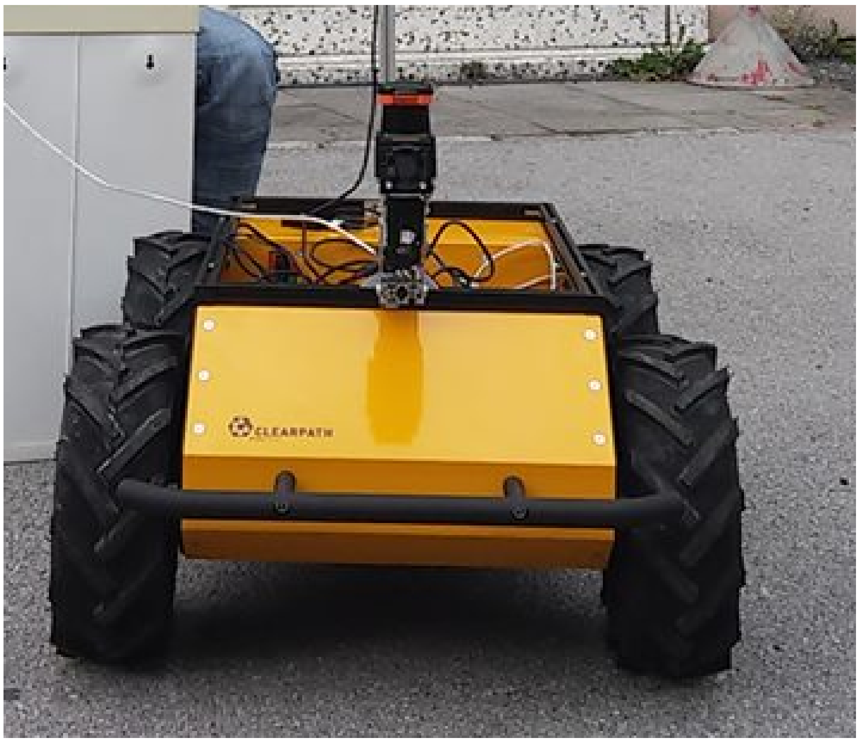

Figure 7.12: Illustration of the real robot prototype used for testing.

# 8 Conclusion

This chapter summarizes the system presented in this thesis and also provides some suggestions for future improvements.

## 8.1 Discussion

In this thesis we presented a robotic system that is capable of creating a 3D representation of its environment and exploring unknown regions autonomously. The robot is equipped with a sweeping 2D laser scanner to create a three-dimensional point cloud of its surroundings. The wheel odometry fused with the IMU gives a rough localization estimate for the ICP-matcher which is used to insert the created point cloud into the right position of the pose-graph and correct the current localization of the robot.

A part of the created map is then converted into a 3D grid map and its surface voxels are classified in terms of traversability for the robot platform. During the classification process a navigation graph is constructed which connects all traversable voxels with each other and an A*-algorithm can be used to plan a path along this graph. Due to the three-dimension terrain classification the robot is able to generate paths over or under bridges as well as across multiple floors of a building.

A drawback of the implemented 3D classification is that it takes relatively long for big maps in comparison to a 2D or 2,5D obstacle avoidance because a lot of cells inside the 3D grid map have to be evaluated to build the navigation graph.

The robot tries to avoid potential cliffs. But especially in rough terrain voxels behind small hills are often not mapped at the beginning of the exploration phase and therefore avoided because they could be a cliff. Therefore, in certain situations the robot seems to be a bit too cautious in some rough terrain environments.

In the evaluation we showed the performance of the implemented algorithm and came to the conclusion that the terrain classification works well on smaller maps but for classifing big maps some performance improvements are needed. We also tested our algorithms on a physical robot and not only in the simulation to verify that it also works in the real world.

## 8.2 Future Work

To speed up the terrain classification a hierarchical approach could be used so that the big 3D grid get split into many small ones and classified in a multi-threaded manner. If during the terrain classification enough near frontiers have been found, a further classification is stopped and a path to the most suitable one is planned. This would prevent the classification of the whole map and save a lot of time.

Additionally a better way to detect cliffs should be found as well as a different strategy to explore the potential cliffs in more detail. For example by setting frontiers near some of the potential cliffs to further map this area and detect if it is really a cliff or just a small hill and therefore traversable.

# Bibliography

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623 (cit. on p. 9).

[2] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. "G2o: A general framework for graph optimization." In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 3607–3613 (cit. on p. 9).

[3] Frank Dellaert. "Factor Graphs and GTSAM: A Hands-on Introduction." In: 2012 (cit. on p. 9).

[4] Ji Zhang and Sanjiv Singh. "LOAM: Lidar Odometry and Mapping in Real-time." In: *Robotics: Science and Systems Conference*. Pittsburgh, PA, July 2014 (cit. on p. 10).

[5] M. Leingartner, J. Maurer, G. Steinbauer, and A. Ferrein. "Evaluation of sensors and mapping approaches for disasters in tunnels." In: *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. Oct. 2013, pp. 1–7 (cit. on p. 10).

[6] Jochen Sprickerhof, Andreas Nüchter, Kai Lingemann, and Joachim Hertzberg. "An Explicit Loop Closing Technique for 6D SLAM." In: *ECMR*. Ed. by Ivan Petrovic and Achim J. Lilienthal. KoREMA, 2009, pp. 229–234. ISBN: 978-953-6037-54-4 (cit. on p. 10).

[7] Andreas Nüchter, Kai Lingemann, Joachim Hertzberg, and Hartmut Surmann. "6D SLAM&Mdash;3D Mapping Outdoor Environments: Research Articles." In: *J. Field Robot.* 24.8-9 (Aug. 2007), pp. 699–722. ISSN: 1556-4959 (cit. on p. 10).

[8] Renaud Dubé, Daniel Dugas, Elena Stumm, Juan I. Nieto, Roland Siegwart, and Cesar Cadena. "SegMatch: Segment based loop-closure for 3D point clouds." In: *CoRR* abs/1609.07720 (2016) (cit. on p. 10).

[9] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. "Visual SLAM algorithms: a survey from 2010 to 2016." In: *IPSJ Transactions on Computer Vision and Applications* 9.1 (June 2, 2017), p. 16. ISSN: 1882-6695 (cit. on p. 10).

[10] Thomas Whelan, Stefan Leutenegger, Renato F. Salas-Moreno, Ben Glocker, and Andrew J. Davison. "ElasticFusion: Dense SLAM Without A Pose Graph." In: *Robotics: Science and Systems*. 2015 (cit. on p. 10).

[11] Jakob Engel, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM." In: *Computer Vision – ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part II*. Ed. by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars. Cham: Springer International Publishing, 2014, pp. 834–849. ISBN: 978-3-319-10605-2 (cit. on p. 10).

[12] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. *The Dynamic Window Approach to Collision Avoidance*. Tech. rep. 1995 (cit. on p. 11).

[13] A. Pratkanis, A. E. Leeper, and K. Salisbury. "Replacing the office intern: An autonomous coffee run with a mobile manipulator." In: *2013 IEEE International Conference on Robotics and Automation*. May 2013, pp. 1248–1253 (cit. on p. 11).

[14] G. Grisetti, C. Stachniss, and W. Burgard. "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters." In: *IEEE Transactions on Robotics* 23.1 (Feb. 2007), pp. 34–46. ISSN: 1552-3098 (cit. on p. 11).

[15] Konstantin Lassnig. "An autonomous robot for campus-wide transport tasks." MA thesis. Graz: Technical University Graz, 2016 (cit. on pp. 12, 28, 48).

[16] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. "The Office Marathon: Robust Navigation in an Indoor Office Environment." In: *International Conference on Robotics and Automation*. 2010 (cit. on p. 12).

[17] D. Maier, A. Hornung, and M. Bennewitz. "Real-time navigation in 3D environments based on depth camera data." In: *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. Nov. 2012, pp. 692–697 (cit. on p. 12).

[18] P. Fankhauser, M. Bloesch, C. Gehring, M. Hutter, and R.Y. Siegwart. *Robot-centric Elevation Mapping with Uncertainty Estimates*. ETH-Zürich, 2014 (cit. on p. 12).

[19] Sebastian Pütz, Thomas Wiemann, Jochen Sprickerhof, and Joachim Hertzberg. "3D Navigation Mesh Generation for Path Planning in Uneven Terrain." In: 49 (Dec. 2016), pp. 212–217 (cit. on p. 13).

[20] F. Colas, S. Mahesh, F. Pomerleau, M. Liu, and R. Siegwart. "3D path planning and execution for search and rescue ground robots." In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Nov. 2013, pp. 722–727 (cit. on p. 13).

[21] A. Hertle and C. Dornhege. "Efficient extensible path planning on 3D terrain using behavior modules." In: *2013 European Conference on Mobile Robots*. Sept. 2013, pp. 94–99 (cit. on p. 13).

[22] B. Yamauchi. "A frontier-based approach for autonomous exploration." In: *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*. July 1997, pp. 146–151 (cit. on p. 14).

[23] J. Maurer and G. Steinbauer. "Autonomous risk-aware exploration." In: *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. Oct. 2013, pp. 1–8 (cit. on p. 14).

[24] C. Dornhege and A. Kleiner. "A frontier-void-based approach for autonomous exploration in 3d." In: *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*. Nov. 2011, pp. 351–356 (cit. on p. 14).

[25] P. G. C. N. Senarathne and D. Wang. "Towards autonomous 3D exploration using surface frontiers." In: *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. Oct. 2016, pp. 34–41 (cit. on p. 14).

[26] Morgan Quigley, Ken Conley, Brian P Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "ROS: an open-source Robot Operating System." In: 3 (Jan. 2009) (cit. on p. 15).

[27] François Pomerleau, Francis Colas, and Roland Siegwart. "A Review of Point Cloud Registration Algorithms for Mobile Robotics." In: 4 (May 2015), pp. 1–104 (cit. on p. 17).

[28] François Pomerleau, Francis Colas, Roland Siegwart, and Stéphane Magnenat. "Comparing ICP Variants on Real-World Data Sets." In: *Autonomous Robots* 34.3 (Feb. 2013), pp. 133–148 (cit. on pp. 17, 26, 28).

[29] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees." In: *Auton. Robots* 34.3 (Apr. 2013), pp. 189–206. ISSN: 0929-5593 (cit. on pp. 17, 30).

[30] P. Beno, V. Pavelka, F. Duchon, and M. Dekan. "Using Octree Maps and RGBD Cameras to Perform Mapping and A* Navigation." In: *2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)*. Sept. 2016, pp. 66–72 (cit. on p. 18).

[31] R.P. Goebel. *ROS by Example: Packages and Programs For Advanced Robot Behaviors*. Pi Robot Production Bd. 2. P. Goebel, 2014. ISBN: 9781312392663 (cit. on p. 21).

[32] Lentin Joseph. *Mastering ROS for Robotics Programming*. Ed. by Packt Publishing. Packt Publishing, 2015. ISBN: 9781783551798 (cit. on p. 21).

[33] Thomas Moore and Daniel Stouch. "A Generalized Extended Kalman Filter Implementation for the Robot Operating System." In: *Intelligent Autonomous Systems 13 - Proceedings of the 13th*

*International Conference IAS-13, Padova, Italy, July 15-18, 2014*. 2014, pp. 335–348 (cit. on p. 25).