# Recognition and Tracking of Dynamic Objects Using Static Sensors

handed in
MASTER'S THESIS

Raimund Zille

born on the 24.01.1994
living in:
Arberweg 28
85748 Garching
Tel.: 0049 - 176 565 84 398

Human-centered Assistive Robotics
Technical University of Munich

Univ.-Prof. Dr.-Ing. Dongheui Lee

| | |
|---|---|
| Supervisor: | Objective: Alexander Dratva, Hcr: Shile Li |
| Start: | 01.05.2018 |
| Intermediate Report: | 25.2.2019 |
| Delivery: | 25.3.2016 |

TECHNISCHE UNIVERSITÄT MÜNCHEN
**Human-centered Assistive Robotics**
UNIV.-PROF. DR.-ING. DONGHEUI LEE

March 23, 2019

M A S T E R ' S  T H E S I S
for
Raimund Zille
Student ID 03639723, Degree EI

# Recognition and Tracking of Dynamic Objects Using Static Sensors

Problem description:

Searching for a parking spot, even just on a parking lot can be time consuming, exhausting and uncomfortable for the driver. This can have different reasons, e.g., it's stressful to drive in a narrow and sometimes confusing layout. Therefore, the next step is autonomous valet parking. For every autonomous driving task, it is critical to have a precise localization. This can be challenging in complex structures, e.g., a parking garage. Also, current car models aren't equipped with the needed sensors to do this task on their own. One way to solve this problem, is to equip the parking garage with appropriate sensors, which provide a secure and affordable localization. André Ibisch et. al [1] showed that it's a promising approach to localize and track a vehicle based on environment-embedded LIDAR sensors. To achieve real-time, they process only a small subset of active points, to avoid reprocessing static objects, e.g., walls.
The main aim of this thesis is to develop and optimize a method to achieve a fast and precise localization of a vehicle and to identify and track other dynamic objects in the area. The input will be a fusioned pointcloud from several static multi-layered LIDARs. With these data, the vehicle will be controlled autonomously through a parking area. This method shall be optimized in respect to controllability of the vehicle, which is affected by the speed and the accuracy of the dynamic objects recognition. There is nearly always a tradeoff between speed and accuracy. Finally, the method will be evaluated with real data against human ground truth estimation.

Tasks:

- Literature research on LIDAR based dynamic objects recognition, tracking and optimization methods
- Development and integration of an optimization-based approach for recognition and tracking of dynamic objects
- Theoretical and experimental evaluation and tuning of the algorithm in regard to controllability of the vehilce

Bibliography:

[1] A. Ibisch, S. Stümper, H. Altinger, M. Neuhausen, M. Tschentscher, M. Schlipsing, J. Salinen, and A. Knoll. Towards autonomous driving in a parking garage: Vehicle localization and tracking using environment-embedded lidar sensors. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 829–834, June 2013.

| | |
|---|---|
| Supervisor: | M. Sc. Shile Li |
| Start: | 01.05.2018 |
| Intermediate Report: | 25.02.2019 |
| Delivery: | 25.03.2019 |

(D. Lee)
Univ.-Professor

**Abstract**

In this thesis, we propose a new approach for localization and tracking of vehicles and objects on a parking area, based on environment-embedded LIDAR sensors. The sensors are set up such that they scan for the auto body. After identifying the foreground data in each sensor using minimal return value, we integrate multiple sensor readings into a common world coordinate system. In order to perform detection and tracking, we developed two loosely connected modules. One utilizes grid based clustering and generating minimal rectangle boundingboxes for every cluster to estimate the position and dimension of each object. This module always processes the complete pointcloud and aims to detect every object in the area. To boost the precision of positions for vehicles and to overcome the drawbacks of a clustering based approach, a second module is introduced. It only maintains existing tracks and updates them by optimizing a fixed size rectangle to the current tracks position. For tracking purpose, both modules employ kalman filtering. The system was tested on an area of $\approx 40m \times 80m$ with 5 multi layered LIDAR sensors. In our experiments, we showed that the proposed system allows to control multiple cars with the generated localizations, while sending the data via LTE over a remote server who controls the vehicles. Our system's results were compared to human-labelled ground-truth data and to the odometry from a controlled car.

# Contents

# Chapter 1

# Introduction

**Motivation** Fully autonomous driving vehicles has been subject to research for over 30 years. The first publications towards this field were released around 1990 by E.D. Dickmans at University Bundeswehr Munich [DMC90] and T- Kanade at Carniegie Mellon University [THKS87]. Since then, this field of research is increasing in academia as well as in the automobile industry. A main drive is the continuous development of computation and sensing technologies. At this point, after decades of research, vehicles can be equipped with various driver assistant system, such as adaptive cruise control, lane keeping assist and automatic emergency brake. On the other side, fully autonomous vehicles in series production is not expected to arrive before the next decade according to several analysts [HKMS17].

In contrast, automated driving within controlled environment has a high potential to take the next step from research and development to series production. This is because the driving tasks can be considered less complex, as the operational area is known in advance and can be restricted. Furthermore, the area can also be restricted for e.g. pedestrians, to minimize the risk. One concept falling into this category is automated valet parking (AVP). In AVP, the vehicle is passed to the system at a drop off point, then automatically parks itself on the area and comes back on demand. This can be used as a quality of life improvement for the drivers, because searching for a parking spot, even just on a parking lot can be time consuming, exhausting and uncomfortable for the driver. This can have different reasons, for example it is stressful to drive in a narrow and sometimes confusing layout or finding an empty parking spot can take forever. Additionally it can also be used to extend the automatic manufacturing process for cars or help with testing cars where they have to drive certain manoeuvres. Furthermore another benefit is the potential to reduce parking space. Parking space can be reduced by 15% to 50% since no space is needed to exit the vehicles and the density of the parked vehicles can be increased [BW15, BNKZ17]. Last but not least important data and knowledge can be derived, which further pushes the development of fully automated vehicles.

For every advanced autonomous driving task, it is critical to have a precise self localization and knowledge of the surroundings, known as environment model. This

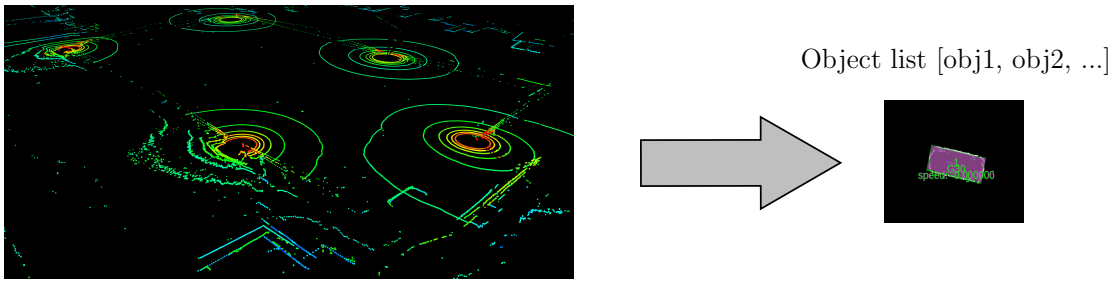



Figure 1.1: Illustration of the goal of this thesis. A module should be developed which takes as input the sensor data and processes them to generate an object list with all objects in the scene. Each object shall consists of a pose, dimension, class and id.

task is normally solved by using on-board sensors of the driving vehicle. Most of the current produced cars aren't equipped with needed sensors to do this task on their own. One way to solve this problem, is to equip the infrastructure with appropriate sensors. Therefore, the infrastructure provides the necessary information and the vehicle gets controlled remotely. The vehicle follows the commands from the infrastructure blindly, therefore doesn't need any sensors nor high computational power. The only constraint is some sort of connection from the infrastructure to the car, e.g. internet. Nowadays, most of the cars are connected to the internet anyway.

**Problem Statement** The objective of this work is to develop a module for dynamic object recognition and tracking with static LIDAR sensors in a known environment. This includes the placement of the LIDARs, processing the raw sensor data and outputting a object list with every object in the area. Each object shall have a position, dimension, classification and an identification number. The constraints for the module are, that the sensors are set up such that they scan the area horizontally as well as having them calibrated. Calibration for LIDAR sensors means, knowing the placement of each other in respect to a known shared coordinate system.
This raises different challenges in comparison to having the sensors equipped to a vehicle. First of, the importance of every object, no matter how far or near it is to a sensor, stays the same. Normally, the closer the objects to the vehicle, the more important they are. The closer they are to the vehicle, the closer they are to the sensor, resulting in more dense and less noisy measurement data for the object. Furthermore, by using sensors distributed over a large area, the pointcloud changes a lot in density, which might be a problem. Timing is also a important factor, which is undoubtedly always a challenge. In this case, the importance even increases, because we introduce delay by controlling the vehicle via Internet, caused by the latency of the data transmission, which we can not change. Last but not least, occlusion also plays a big role. This task however is mostly solved by smart sensor

placement and is not a focus of this work.
This work is structured as follows: Chapter 2 presents literature review about object recognition and tracking. The complete system is shown in Chap. 3. Chapter 4 elaborates the proposed system and its implementation. In Chap. 5 the experimental setup and the results achieved by the developed approach are shown. Finally, we have the conclusion in Chap. 7 and the discussion in Chap. 6.

# Chapter 2

# Literature Review

This chapter covers related literature for object recognition and tracking with the focus on LIDAR data. In Sec. 2.1, some general approaches and researches in this field are presented. Section 2.2, two approaches tackling the same problem, object detection and tracking with environment embedded sensors, are shown.

## 2.1 Object Recognition and Tracking

**Leg Tracker** Object Detection and Tracking is necessary for nearly every robotic task. A. Leigh *et. al* [LPOZ15] propose a system which tracks person with the use of LIDAR measurement data. They use one LIDAR mounted to a movable roboter at a height of roughly $30cm$ above the ground plane. The scan points are clustered based on a fixed distance threshold, such that any points within the threshold are grouped together as a cluster. Every cluster is classified as human or non-human based on a set of geometric features of the cluster with a random forest classifier. For each discovered cluster a kalman filter [TBF] with a linear transition model $(x, y, \dot{x}, \dot{y})$ is initialized. For associating the observations to the tracks, global nearest neighbour data association is used. For this task, they employ the Munkres algorithm [Kuh55]. Persons are also tracked with kalman filters. The clusters are interpreted as the legs of persons, therefore up to two clusters can be assigned to a single person track. To further increase the robustness, they construct a occupancy grid map which is updated with the use of the odometry of the robot. The mid-point of the map corresponds to the position of the LIDAR sensor. They update this map with all scan clusters not assigned to a human track on each iteration to build up a occupancy map. The occupancy map is used to assist data association by not allowing assignments between clusters in occupied space and human tracks.

**2.5D Motion Grid** A. Asvadi *et. al*[APN15] propose an approach for detection and tracking of moving objects using 2.5D motion grids. The sensor is mounted on top of the car and the system knows about the odometry of the car. A 2.5D

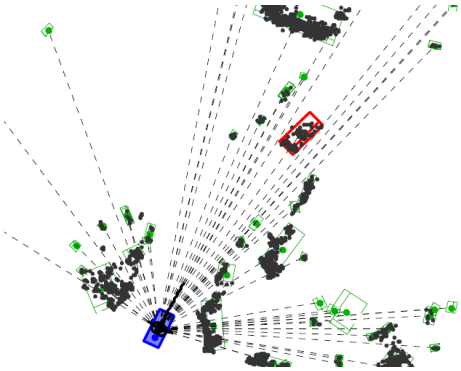


Figure 2.1: Optimizing rectangles on LIDAR data to find vehicle poses (red rectangle). The LIDAR is mounted to a vehicle (blue). Objects not identified as vehicles are marked green. Image taken from [MHH17].

local grid map is constructed at each time frame. The 2.5D grid stores the average height and variance of points (.5D) in a 2D grid (x,y). To remove the ground, the variance of the height of points falling into each cell is used. If it is under a certain threshold, it is selected as a candidate for a ground cell. To avoid labeling every planar surface as ground, e.g. also car roofs, a cell is confirmed as ground cell if the average height value of the cell is below a given threshold. They buffer the latest local grid maps and update each of them with the use of odometry to the current car frame. To calculate the current 2.5D map, they average the height in the cells of the last $m$ buffered local grids. Because they are interested in only moving objects, they calculate the motion map by comparing the current local grid map from the 2.5D map (consisting of the mean of the last $m$ local grids, therefore representing the static objects). To reduce false detections because of errors in the odometry, they also query neighbours of the current 2.5D map to compare the values, yielding to a more stable result. Connected component clustering is employed on the motion map and the centroids of the obtain clusters are tracked using kalman filters with constant velocity model. For data association between observation and tracks, nearest neighbour search is employed.

**Optimizing a Rectangle** D. Morris *et. al* [MHH17] propose a optimization based approach for obtaining fast and precise measurements of vehicle poses in the surroundings. They use a LIDAR sensor mounted to the top of a vehicle. After clustering the measurements, they optimize a matching function with the Levenberg Marquardt local optimizer, to derive the pose and size of the vehicle. The matching function is a filter function consiting of 4 summed rectangular regions. They optimize for the pose and the dimension of the rectangle. They employ a kalman filter for each tracked object. The result can be obtained in Fig. 2.1.

**Survey on background subtraction** For real-time capable detection and recognition, it is crucial to remove the background to reduce the amount of data which has to be processed and to be able to segment different objects.LIDAR measurements can either be presented in depth images or with pointclouds. A pointcloud is a set of points. We want to distinguish between organized and unorganized pointclouds. An organized pointcloud resemble an organized image or matrix like structure, where the data is split into rows and columns. Stereo cameras or Time of Flight cameras for example produce such data. The advantages of an organized dataset is that knowing the relationship between points (e.g. pixels), nearest neighbour operations are much more efficient [RC11]. Unorganized pointclouds resemble a list of points, where the sequence of points does not matter. Merging multiple organized pointclouds together results for example in an unorganized pointcloud.
K. Greff *et al.* [GBK+12] examined some well known background subtraction methods. Although they use depth images, organized point clouds can be seen the same way because of their image like structure. Therefore, following methods only work for organized point clouds.

- **First Frame Subtraction:** In this method the first frame of the sequence is subtracted from every other frame.
- **Single Gaussian:** In this method, the scene is modelled as a texture and each pixel (row, col) of this model is associated to a Gaussian distribution. During a learning phase, the mean and variant values for each pixel are calculated. Later on every pixel value which differ more than a constant times the standard deviation of its mean are considered foreground.
- **Multiple Gaussian:** This method [SG99] is similar to the above, Single Gaussian. The difference is that each pixel is not associated to a single but to multiple gaussian distributions. Therefore it is capable of learning multiple background values for a pixel. *This method is often used in image processing, because it can adapt to slow lightning changes by adapting the values and is capable of removing different colors corresponding to the background (e.g. sky is gray or blue)*
- **Codebook Model:** This model aggregates the sample values for each pixel into codeworks. The Codebook model considers background values over a long period of time. This method allows to bootstrap the system in the presence of foreground objects and can account for dynamic backgrounds.
- **Minimum Background:** This is the first model specifically developed for depth images or organized point clouds. During training stage, the minimum returned value for each pixel is stored. Later on, every pixel closer to the sensor (depth value is smaller than the saved value) is considered foreground. This works well for range based data, because the foreground usually is *in front of* the background.

K. Greff *et al.* [GBK+12] conclude, that the statistical approach used by the *Single Gaussian* method is affected by the high variances of alternating pixels and low variances of stable pixels. A pixel is classified as background if the value differs less than a constant times the standard deviation compared to the mean. This means a constant has to be chosen to accomplish this. Having a high constant results to false negatives when foreground objects occludes the high variance region. On the other hand, if the constant is small, stable pixels will emit a lot of noise. A big advantage of this approach is the constant learning rate and possibility to adapt to changes. They state that for ranged based measurement methods the *Minimum Background* and the *Codebook Model* method produce the best results, favouring the *Minimum Background* approach because of its simplicity.
For unorganized point clouds, [RC11] propose a background subtraction method similar to the *First Frame Subtraction*. In unorganized point clouds, the relationship of the points are unknown. Therefore they use an octree [Mea82] based change detection algorithm. They build up an octree with the incoming point cloud and compare the structure with a reference octree created with the first retrieved point cloud. Every point stored in a voxel part of the current octree which was not present in the previous octree is seen as foreground.

## 2.2 Autonomous Valet Parking

Mercedes-Benz and Audi both invested into the field of autonomous valet parking (AVP) using intelligent infrastructure. Intelligent infrastructure means, that the autonomous vehicle does not need any sensors, because the infrastructure is equipped with sensors and computational power to achieve this task. Other than that, there is not a lot of research directly related to this problem.

**Audi** A. Ibisch *et al.* [ISA+13] from Audi AG published a paper about AVP in a parking garage. They use LIDAR sensors placed on ground level, aiming to detect the wheels of the vehicle. By using environment embedded sensors, they achieve a cost efficient approach, which does not require vehicles with specific equipment, making it transferable to real-world scenarios. They estimate the position and orientation of the vehicle using a feed forward approach. First, they label each point as *active point* or *inactive point* and only process *active points*. After identifying the *active points* for each LIDAR sensor, they stitch them together in a shared coordinate system. They find out the position of each sensor in regard to each other by matching them into a CAD-map and using direct linear transformation (DLT) [AA71]. By putting the sensors on ground level, the sensors only detect the wheels of the vehicles. Using three wheels, they match a rectangle, representing the vehicle model, onto these by using a RANSAC algorithm. The output is the position and orientation of the vehicle. For tracking, they are using a extended kalman filter [TBF] with a physical motion model and reasonable observation noise. They


(a) Position of the LIDAR sensor on the parking garage.

(b) Typical parking garage scenario.

(c) LIDAR data of the recorded setting.

Figure 2.2: RANSAC modelfitting by detecting the wheels of a car. The 2D 1 layered LIDAR sensor is placed at ground level. A RANSAC is employed to fit a rectangle on three detected wheels.

use nearest neighbour data association between observations and trackers, resulting in a consistent and reliable localization.
They achieve a real-time system using a desktop PC with an Intel Core i7 CPU at 2.8 GHz and 4 GB RAM memory. The RANSAC algorithm has a constant runtime of approx. 5ms compared to an Hough transformation algorithm which varies between 25ms and 50ms. This variety comes by the amount of active points in each time frame.
For evaluation, they compared the results with human labelled ground truth data

**Mercedes-Benz and Bosch** Since the second quarter of 2018, the first public autonomous valet parking system, developed and published by Mercedes-Benz and Bosch, is available. It is located in the parking garage of the Mercedes-Benz museum [unk17]. Regarding the commercial video, they used a high density setup of sensor on a ground level, most probably following the approach from [ISA+13].

# Chapter 3

# Autonomous Valet Parking System Overview

This Chapter describes the complete autonomous valet parking system. It consists of the infrastructure based localization system, developed in this work. Furthermore it consists of a remote server merging the data from both sides, the localization system and the vehicles, and providing trajectories for the controlled vehicles. Every controlled vehicle is also part of the system, but the goal is that they should be kept as simple as possible, making the system available for the majority of the cars.

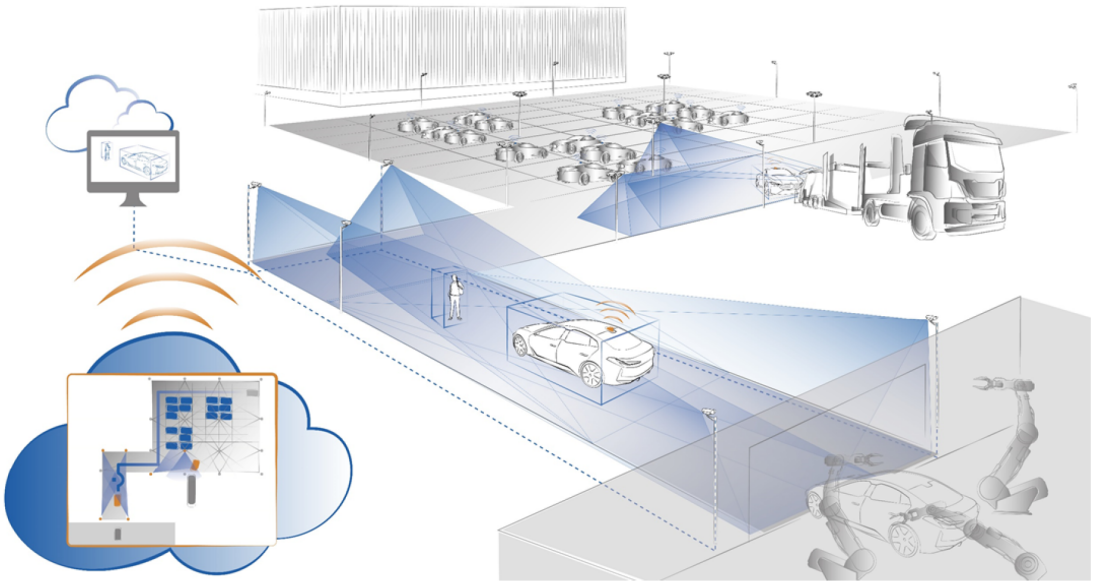


Figure 3.1: AVP illustration in an industrial environment [Sch19]. Sensors are equipped to the infrastructure. A localization system detects every objects in the area and sends them via internet to the remote server. The remote server takes care of the path and trajectory planning and sends the vehicle a save trajectory to follow.

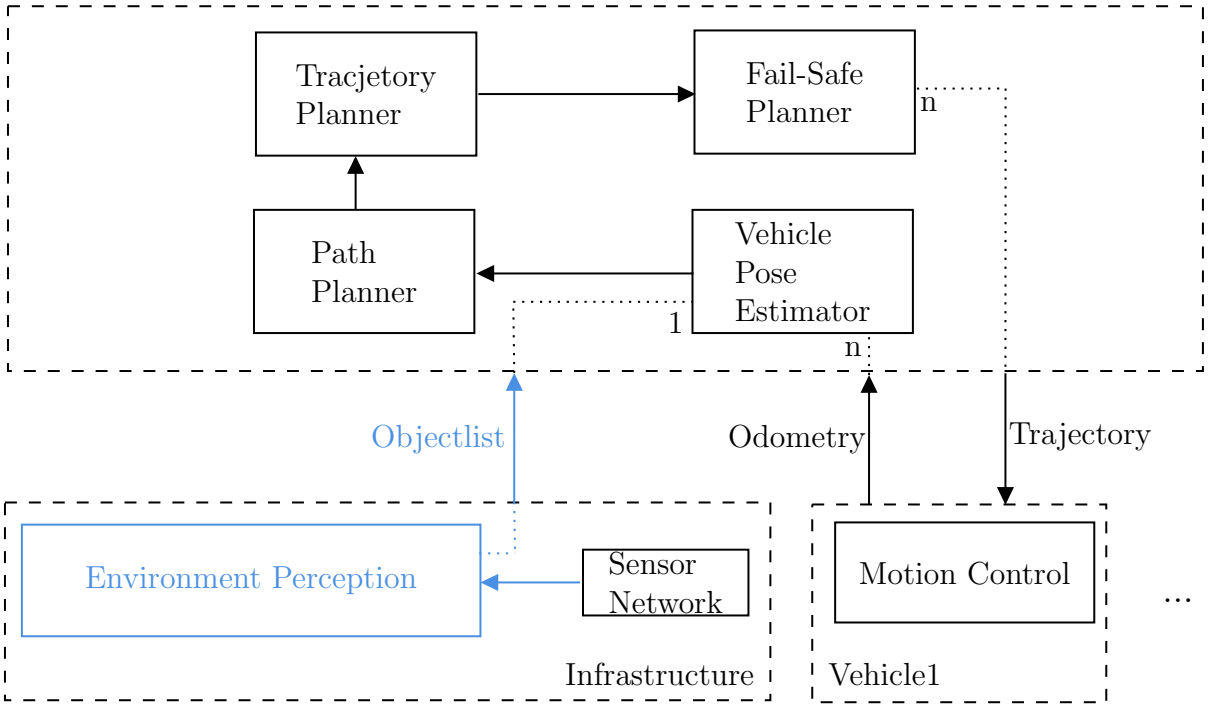



Figure 3.2: Overview of the complete AVP system. It consists of an infrastructure based environment perception module, developed in this work. It communicates via LTE to a remote server, which fuses and filters the localizations of vehicles with the corresponding odometry received from the cars if available. Furthermore, the remote server generates a path for each car and sends a safe trajectory via LTE back to the vehicle, which follows this with a simple motion controller. This leads to having every expensive algorithms running either on the remote server or on the infrastructure, making the system available to most cars with a internet connection.

## Different Components of the System

The complete system consists of following three different components:

**Environment Perception** This module is embedded into the infrastructure and provides an object list of every object inside the area. It is described in chap. 4.

**Remote Server** This module, which is responsible for conducting the vehicles safely to their destiny. It is connected via internet to the *Environment Perception* as well as to every *Vehicle* which is controlled by the system. It merges the informations of the *Vehicles* with these of the *Environment Perception* by using an extended kalman filter for each controlled vehicle. ThekKalman filter fuses the odometry of the vehicles with the absolute localization poses of the from the *Environment Perception* to achieve a noise free and stable position for every controlled vehicle.

It is also responsible to generate a path for each vehicle as well as then convert it to a trajectory. Ideally, the trajectory should avoid every obstacle in the scene, sensed by the *Environment Perception*. For safety reasons, the server checks if every generated trajectory is safe. This means, that each trajectory has to be safe for the next $x$ seconds. If this is not the case, the speed of the trajectory is getting lowered till it is safe again or the speed is at zero. It is important to verify for the next $x$ seconds because of the latency between the server,the vehicle and the environment perception. The fail safe concept is presented here [Sch19].

**Vehicles** The vehicles controlled by the system. Every task needing high computational power should be located in either the *Environment Perception* or the *Remote Server*. The vehicle therefore just has to upload its own data, the odometry and some informations about the car, as well as to receive and execute the appropriate section of the trajectory. The upload data is available in every series production vehicle already. For following the trajectory, it uses a simple motion controller.

## Communication between System Components

The components communicate via LTE. Because the delay in mobile communication can vary, a time synchronization between the systems is needed. This is done by synchronizing the *Environment Perception* and the *Vehicles* with the *Remote Server*. Because of the additional delay, around $50ms$, through the LTE-communication, it is crucial for the *Environment Perception* to be as fast as possible in providing its informations to keep the overall delay in a reasonable range.

# Chapter 4

# Localization Module

This Chapter presents the localization module aiming to detect and track objects. An object consists of a dimensions (width, length), a pose $(x, y, \theta)$, a identification number ID and a class. The proposed system is based on LIDAR data only and aims to estimate position, orientation, dimension and class of objects on a predefined area. To achieve applicability, the system should be failsafe, accurate and real-time capable. Objects which are vehicles have a higher priority to be accurately localized, because they are getting controlled (closed loop) with these informations.
An overview of the complete module is shown in Fig.4.1. Following the different steps and how the chapter is structured. The sensor setup is described in Sec. 4.1. Every sensor measures data points from its environment by emitting laser rays. These data points are getting separated into foreground and background points using the Minimum Background model, described in Sec. 4.2. Foreground points of each sensor are then getting transformed into a common world coordinate system, described in Sec. 4.3. This allows to use detection algorithms on combined sensor data for generating localization hypotheses. The merged foreground data is processed by two different localization algorithms in parallel:

- The *Fast Localization* aiming to detect every obstacle on the area and providing its information as fast as possible, elaborated in Sec. 4.5. The data is clustered and for each cluster a minimal area rectangle covering all cluster points is generated. The rectangle defines the pose, orientation and dimension of the object.

- The *Precise Localization* generating hypotheses for already known objects, classified as vehicles, to boost the accuracy of tracked vehicles, elaborated in Sec. 4.6. It processes one tracked object at a time, using a non linear optimization function to fit a rectangle based model on the points at the current track position.

Finally, hypotheses of both modules are tracked to ensure a identification number for each object and to keep it present if it might not be detected at a time frame.

Finally, the fusion of vehicle hypothesis with the corresponding odometry by the remote server is shortly presented in Sec. 4.7.

## 4.1 Sensor Network

A Light Detection and Ranging (LIDAR) sensor measures distances between the sensor and its environment by using a set of laser rays. Similar to a Radar sensor, it emits a laser pulse and detects the light reflected by an object. The distance is calculated with the time of flight of the pulse. To minimize false detections, e.g. a reflection from a raindrop, some LIDARs emit multiple pulses into the same direction to check for consistency. The result of a single measurement step is a set of 3D points (x,y,z) represented in an coordinate system with the sensor in the origin. Following, these type of data sets are called point clouds. Point clouds separate as organized or unorganized data set, defined in Sec. 2.1. The used LIDAR sensors produce organized point clouds. Pointclouds are denoted as $\mathcal{C}$, organized pointclouds are marked with $\mathcal{C}^{m \times n}$ and unorganized as $\mathcal{C}^{1 \times n}$.
The Point Cloud Library (PCL) [RC11] is used to handle the data structures.
Figure 4.2 shows that we can not reliable put the sensors near and parallel to the ground to scan for the wheels as proposed by [ISA+13]. To come up with a more generic approach, loosening from the constraint of having a complete flat ground like in a parking garage, we adjust all LIDAR sensors in the parking area so they scan horizontal over the complete area, without hitting the ground with the horizontal layer. This setup aims to sense the auto body and perform detection based on this information (see Sec. 4.5 and Sec. 4.6).

## 4.2 Background Subtraction

A large set of LIDAR measurements originate from static objects, e.g. walls. To increase robustness and computational efficiency of the proposed approach, we follow the line of [ISA+13] by processing only a small subset of all points, called foreground points. It is also mandatory to remove the ground plane to be able to segment the data. K. Greff *et al.* [GBK+12] state the Minimum Background model to work best for ranged data, see Sec. 2.1, and is used here as well. For this method, the data has to be present in a organized manner.
The used sensors return an array of range measurements which then get converted to point clouds by their software using the azimuth angle and the measured range.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cos(\theta) \\ sin(\theta) \\ cos(\epsilon) \end{bmatrix} * r \tag{4.1}$$

With $x, y, z$ representing the point in a 3D Cartesian coordinate system with the origin corresponding to the LIDAR location. $\theta$ is the azimuth angle of the detecting

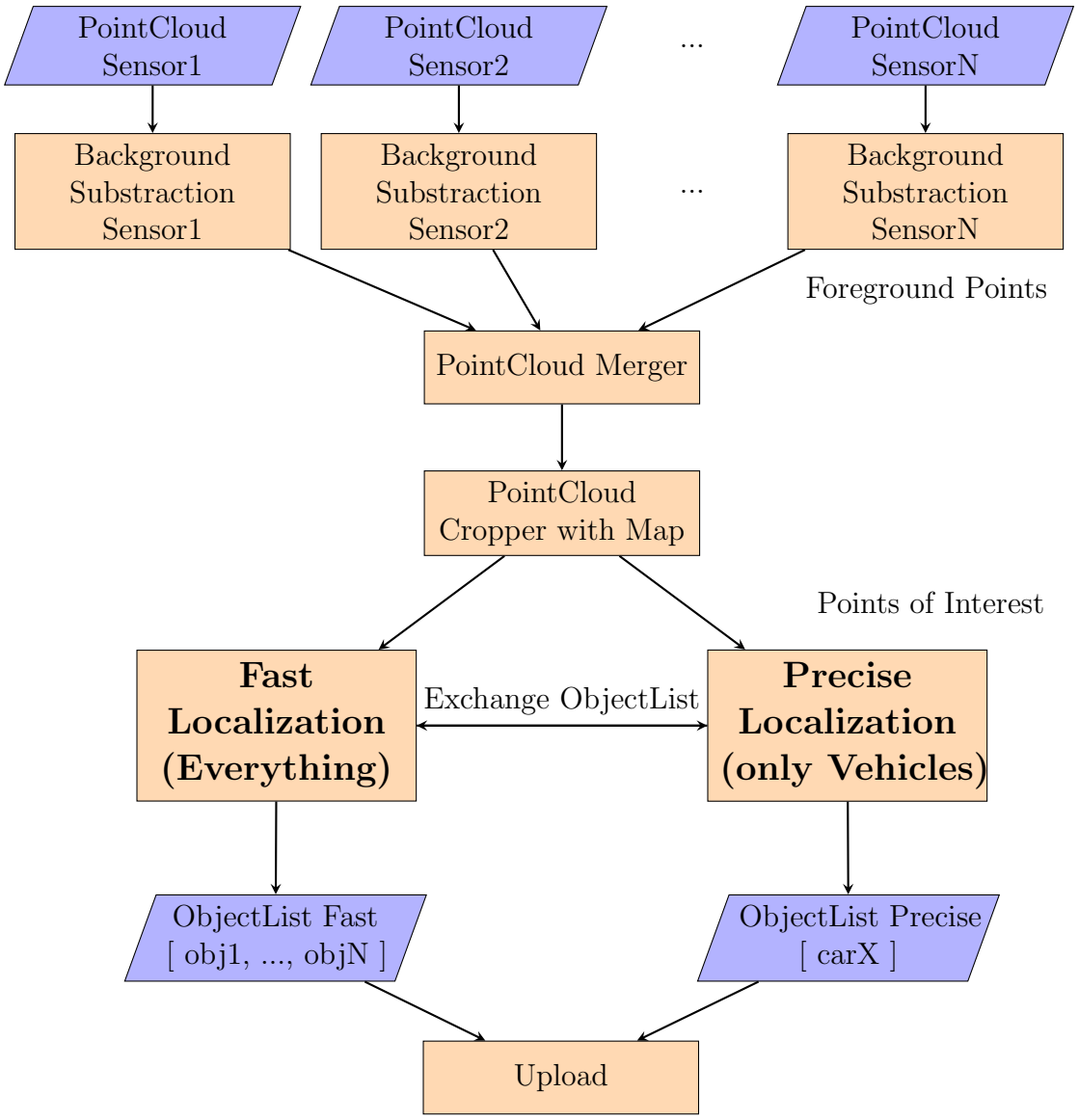



Figure 4.1: Overview of the complete localization system. First, for each Sensor independently, the background is subtracted from the retrieved data (Sec. 4.2). The resulting foreground Points are merged and only Points inside a predefined area of interest are kept (Sec. 4.3. This data is then provided to the two localization modules, the *Fast* (Sec. 4.5) and the *Precise* (Sec. 4.6) Localization module. Finally, the object lists are uploaded and merged (Sec. 4.7).


Figure 4.2: Test to sense the wheels of a car on outside area. Even that the ground is relatively flat, it is not reliable to scan for the wheels of a car, especially on higher distances. In position (a), the wheels can be detected for the car, on position (b), the wheels of the same car merge into the measurements from the autobody.

sensor (which is spinning) when it measured the range $r$. $\epsilon$ represents the vertical angle of the sensor layer.
The Minimum Background model compares the distance from each "pixel" with a learned distance during the initial phase. During initial phase, the closest distance for each cell of the matrix structured pointcloud
$mathcalC_t^{m_t \times n_t}$ is saved. While processing, if the current distance is lower than the stored one, the point is seen as foreground, because it is in front of the background. This means we have to calculate $r$ for each point in the measurement. The step size of the azimuth angle $\theta$ gets calculated at the beginning, as it should not change and is defined by the number of columns $n$ of the incoming point cloud. For azimuth angle step size, the average step size for the last $x$ pointclouds gets calculated using:

$$\epsilon_{\text{stepsize}} = \frac{1}{x} \sum_{t=0}^{x} \frac{2\pi}{n_t}$$

With $n_t$ representing the width of the retrieved pointcloud $\mathcal{C}_t^{m_t \times n_t}$ at time $t$. $n_t$ varies a bit over time ( $\pm 2$), therefore the mean of a given time period is calculated. The Minimum Background model needs the data to be of a fixed size to compare it with the saved values. Because of the small variation in the width of the pointcloud, we project each point into a fixed sized matrix with $2\pi\epsilon_{\text{stepsize}}$ columns. The number of rows is equal to the number of layers of the sensor. The algorithm splits in two parts, the learning phase to learn the ranges or processing phase, filtering the background. Each sensor has its own minimum background model.
During the learning phase, the minimal range $r_{ijt}$ of each matrix element from the measurement pointclouds $\mathcal{C}_t^{m_t \times n_t}$ collected over a time period $d$ is saved. . $r_{ijt}$ is the range from the point $p_{ijt} \in \mathcal{C}_t$ to the origin $\overrightarrow{0}$ in 2D space (only x and y coordinates are used, to minimize computational steps). This leads to the result of the learning phase of:

$$r_{ij} = min(r_{ij0}, r_{ij1}, ..., r_{ijd}) \tag{4.2}$$

The foreground points $\mathcal{C}_f$ for a pointcloud $\mathcal{C}$ are retrieved as follows:

$$\mathcal{C}_f = \{p_{ij} | p_{ij} \in \mathcal{C}, \sqrt{p_{ij}.x^2 + p_{ij}.y^2} < r_{ijs}\} \tag{4.3}$$

Eventually, it would make more sense doing the filtering directly in the sensor software, when the ranges and angles are still available. Because of testing and simplicity, we decided to seperate the module from the sensor software. The algorithm takes a organized pointcloud as input and returns the foreground pointcloud, which is unorganized. Pseudocode can be obtained in Alg 1.
The learned values can be saved and loaded to skip the learning phase. This model needs the area to be without objects during learning phase. Ideally, the background values should never change. It is still advisable to relearn the values from time to time. The background subtraction is employed for each sensor parallel.

**Algorithm 1** Background Learning

```
1: function BACKGROUND LEARNING(organized_pointcloud)
2:     col_idx ← 0
3:     θ ← 0
4:     initialize empty filtered_pointcloud
5:     for all row in pointcloud do
6:         for all point in row do
7:             r ← √(point.x² + point.y²)
8:             row_idx ← floor(tan⁻¹(point.x / point.y) / ε_stepsize)
9:             r_saved ← background_ranges[col_idx][row_idx]
10:            if r < r_saved then
11:                if learn ranges then
12:                    background_ranges[col_idx][row_idx] ← r
13:                else
14:                    filtered_pointcloud append point
15:                end if
16:            end if
17:        end for
18:        col_idx++
19:    end for
20:    return filtered_pointcloud
21: end function
```

## 4.3 Sensor Merging

After identifying the *foreground points* for each sensor individually, to avoid multiple detections of the same object and to handle occlusion or intersections of moving objects across different sensors, we project merge the data into a shared coordinate

system. We are then able to perform object detection based on the complete data, avoiding multiple detections of the same object.
To successfully merge measurement data from different sensors into a shared coordinate system, knowledge about the positions of the sensors in regard to each other is needed. LIDAR sensors typically output their data in a local reference frame, where the origin corresponds to the sensors location. To transform this points to a common shared coordinate system, a transformation between these two has to be found. [ISA+13] use a direct linear transformation (DLT) [AA71] to find the projection matrix for each sensor. The DLT needs known correspondences between both reference frames (point to point or line to line). The finding of the correspondences can either be done manually or some feature extraction and correspondences finding algorithm have to be employed to do this automatically. LIDAR manufacture Quanergy [Qua] provide a calibration software capable of finding the transformation completely automatically, which was planned to be used.
The calibration software was working when the LIDARs were standing near each other, but on higher distances, the software could not find enough correspondences to successfully estimate the transformation and did not improve the calibration over the initial guess, done by hand.
The workaround for the experiment is doing the calibration by hand on only three parameters $(x, y, \theta)$, instead on all six $(x, y, z, \theta, \phi, \rho)$. This means we are assuming that each sensor is perfectly horizontal aligned ($\phi = 0, \rho = 0$), which was tried to achieve with a level. The z-value is not important for the calibration, as we will reduce the pointclouds to two dimensions in the next step by projecting every point on the z-plane.
To merge the measurements, we need to wait for every sensor to output its data. Ideally the sensors are synchronized, meaning they start and finish their measurement phase at the same time. This leads to all sensors outputting their measurements at the same time. During our experiment, the sensors were not synchronized. To still merge the clouds together, we wait up to $\frac{1}{f}$ seconds for every sensor to return a pointcloud. The timestamp from the merged pointcloud gets setted to the average of all used to construct it. $f$ is the frequency of the sensors.

## 4.4 Reducing amount of Data by Dimension Reduction and defining an Area of Interest

To further reduce the amount of data, a user predefined area of interest can be declared. After merging the results, points lying outside of the defined area are discarded, further reducing the amount of data. For computational efficiency and because we are only interested in the x and y coordinate of the objects (2D-positions), the point clouds dimension is also reduced to 2D by projecting every *foreground point* to the z-plane of the world coordinate system. Because of the layer based scanning mechanism of the sensors, most of the time (except object is near a sensor)

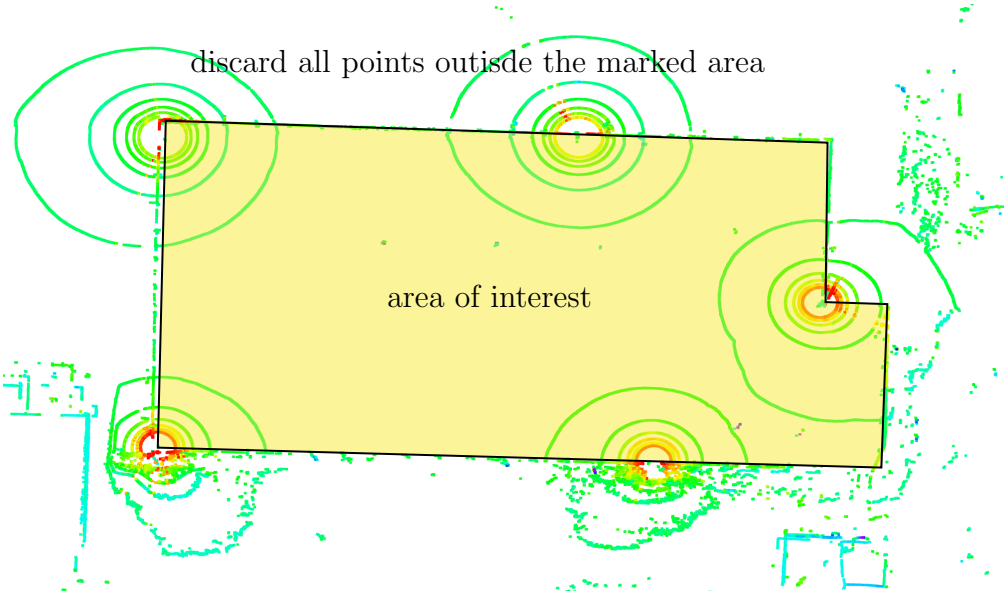



Figure 4.3: Area of interest (yellow polygon). All points lying outside are discarded.

only one layer hits the object anyway. The resulting pointcloud of the foreground point of each sensor in the area of interest is denoted as $\mathcal{C}_{\text{foreground}}$ and referred to as foreground pointcloud. It is an unorganized pointcloud.

## 4.5 Fast Localization

After preprocessing the input data to only contain points of interest, resulting in the foreground pointcloud $\mathcal{C}_{\text{foreground}}$, two different approaches are employed to achieve a robust and precise object detection. The main focus is on localizing cars. This Section describes the first approach, referred to as the *Fast Localization* module, aiming to detect and track every object in the area at every time step. Furthermore, the module should provide its data as fast as possible. The second approach, described in Sec. 4.6 and referred to as the *Precise Localization* module, responsible to further boost the positional accuracy and robustness of detected cars. The module shall provide the position, orientation, dimensions, classification and tracking ID number for each detected object.

An overview of the workflow of the *Fast Localization* module is shown in Fig. 4.4. It follows a similar approach as described by [APN15]. First, the incoming merged foreground point cloud is separated into clusters, described in Sec. 4.5.1. For each cluster, a rectangle bounding box with minimal area including all points of the cluster is generated , described Sec. 4.5.2. Finally, each cluster is classified and tracked, described in Sec. 4.5.3 and Sec. 4.5.4 respectively.

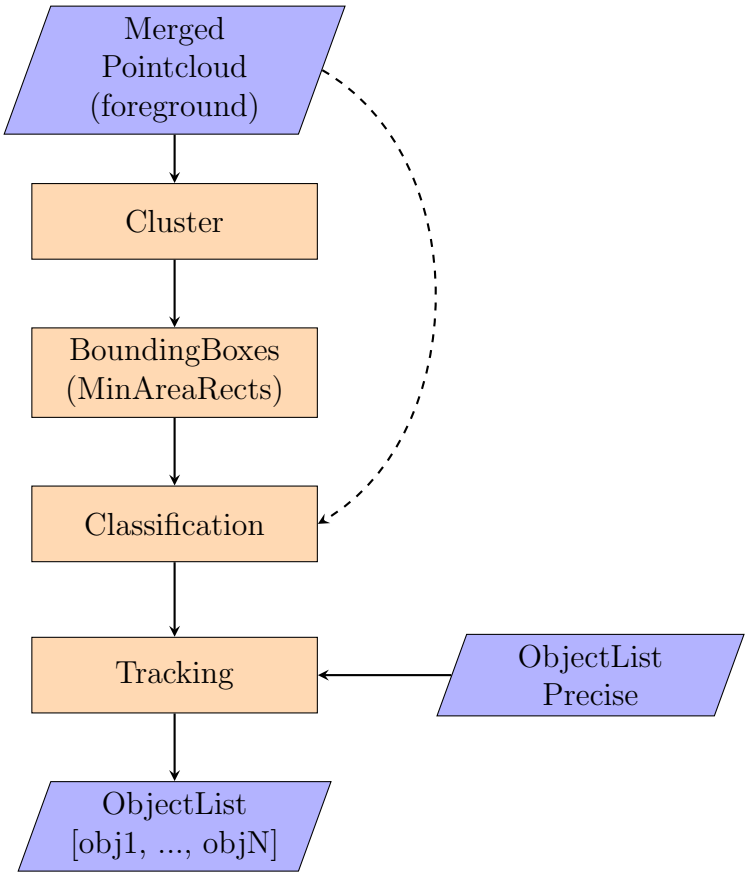



Figure 4.4: Workflow diagram of the Fast Localization module. Input is the merged foreground point cloud. The data gets separated by a clustering algorithm. A bounding box is assigned to each cluster and gets classified. Finally, all tracks are getting updated and an object list is uploaded.

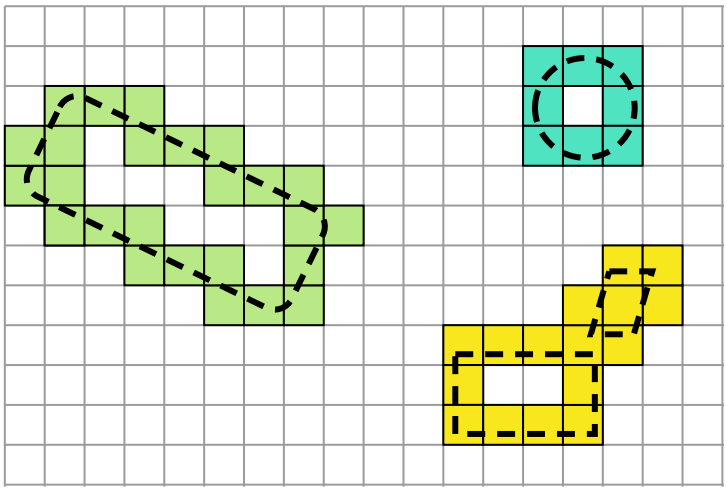


Figure 4.5: Illustration of a grid map based 8-connectivity connected component labelling. There are four objects in the scene. Depending on the grid size of the map, a cluster might have multiple objects inside of it, e.g. the yellow one.

### 4.5.1 Clustering Algorithm

To subdivide the measurement data into several smaller groups of measurement data and/or to find structures inside of it, clustering is a common technique employed for this task. We aim to cluster the data such that each cluster corresponds to one object. It is important, that the clustering method is capable of detecting arbitrary numbers of clusters without knowledge about the dimensions of the objects. There are several algorithms, e.g. DBSCAN [EKSX96], 4C [BKKZ04], RBNN [KWB08], which employ neighbourhood operations to find the density of points in an area and depending on it, adding them to a cluster or labelling them as noise. For region query either some generic and advanced methods like kd-trees [Ooi87] are employed or grid based methods are used. Grid based methods are faster, because the construction time is $O(n)$ and neighbour query takes $O(1)$. However, grid based methods aren't efficient in terms of memory allocation for sparse data bases, in regard to e.g. KD-trees, because a lot of grid cells stay empty. KD-trees have a construction time of $O(nlog(n))$ and a expected complexity of a nearest neighbour lookup of $O(log(n))$ [KWB08].
We follow the approach from [APN15] by building up a local grid map and use connected component labelling, aiming for the lowest processing time possible. An illustration of the method can be obtained in Fig. 4.5.

**Grid-Map** The grid map is defined with a fixed size and resolution. The size is defined by a range of x-values $[x_{min}, x_{max}]$ and y-values $[x_{min}, x_{max}]$. The resolution $r$ is defined by the user and by using connected component clustering, it also defines the maximum distance between a point from a cluster to be assigned to it. E.g. in

Fig. 4.5, the yellow cluster (bottom right) consists of two objects. By lowering the resolution, at some point the cluster will break up into two. The grid map is a two dimensional array, which can be seen as a matrix $\mathcal{M}^{a\times b}$. With $a = \lceil (x_{max} - x_{min})/r \rceil$ and $b = \lceil (y_{max} - y_{min})/r \rceil$.
The grid map $\mathcal{M}^{a\times b}$ is built up with all points from the foreground pointcloud $\mathcal{C}_{\text{foreground}}$ by projecting their position to a cell in $\mathcal{M}$. If the point lies outside of the area defined by the gridmap, it is ignored. The corresponding indices $i, j$ of $\mathcal{M}$ for a point $p$ with coordinates $(x, y)$ can be found as follows:

$$i = Round((x - x_{min})/a) \tag{4.4}$$

$$j = Round((y - y_{min})/b) \tag{4.5}$$

Each entry $m_{i,j}$ of the grid map $\mathcal{M}$ consists of a list of points assigned to the cell and a label id.

$$m_{i,j} = \{\mathcal{P}, l\}$$

With $\mathcal{P}$ being the Set of points assigned to the cell and $l$ the label id. Algorithm 2 shows the pseudo-code for filling the grid map with a pointcloud. The values for $a, b, x_{min}, y_{min}$ are initialized at the start depending on the range and resolution of the map. By assigning a point to a cell, we save a reference to the point and do not copy the data.

**Algorithm 2** Fill detection grid map for clustering

```
1: function fillGridMap(foreground_cloud)
2:     for all point in foreground_cloud do
3:         idx_x = round((point.x − x_min)/a)                ▷ Eq. 4.4
4:         idx_y = round((point.y − y_min)/b)                ▷ Eq. 4.5
5:         if NOT out_of_bounds(idx_x, idx_y) then
6:             grid_map(idx_x, idx_y) assign point
7:         end if
8:     end for
9: end function
```

**Connected Component Labelling** For clustering connect component labelling is employed. Each cell of the grid map $\mathcal{M}$ is either empty or occupied if at least one point falls into this cell. Based on this, occupied connected cells are assigned to the same cluster. We use 8-connectivity to connect the cells, also known as Moore neighbourhood. 8-connected cells are neighbours to every cell that touches one of their edges or corners. They are connected horizontally, vertically and diagonally. Cell $m_{i,j}$ is connected to each cell with the indices $(i\pm1, j)$, $(i, j\pm1)$ and $(i\pm1, j\pm1)$. The algorithm consists of two parts:

- The top function, presented in Alg. 3, clearing the grid map from the points of the previous data, filling in the new points with Alg. 2 and finally iterating over every cell of the map.
- The neighbourhood queering function shown in Alg. 4, a recursive method, to find all connected cells and assign the same label to them.

The runtime of the algorithm highly depends on the resolution $r$ and the range of the grid map, because they defines the amount of cells which has to be checked. During our experiment we used a resolution $r$ of $0.5m$.

**Algorithm 3** Top connected component clustering function

```
1: function connectedComponentClustering(foreground_cloud)
2:     clearGridMap()                ▷ remove prev. labels and point assignments
3:     fillGridMap(foreground_cloud)
4:     clusters ← empty
5:     label ← 1
6:     for all i = 0 to a do
7:         for all j = 0 to b do
8:             if grid_map(i, j).points is NOT empty then
9:                 if grid_map(i, j).label unassigned then
10:                    cluster ← empty
11:                    checkNeighbours(i, j, label, cluster)
12:                    label ← label + 1
13:                end if
14:            end if
15:        end for
16:    end for
17: end function
```

### 4.5.2 Boundingbox Generation

To further abstract the clusters, it is assumed that each cluster corresponds to an object. To get the localization hypothesis (dimensions, orientation, pose) of each object, a boundingbox is generated for each cluster. Our main focus is on detecting cars, therefore we choose to represent every object with a minimal enclosing rectangle boundingbox. The bounding box defines the dimensions, orientation and pose of the object. We choose rectangle boundingboxes because boundingboxes from cars naturally highly correspond to a rectangular box. There are also fast and well known methods to calculate rectangular bounding boxes enclosing a set of point. Under the assumption that the sides of the bounding box are aligned to the car sides, the position (mid-point of the box) corresponds to the car mid-point and the dimension to the width and length of the vehicle. For other objects, it is mandatory that the

**Algorithm 4** neighbours queering function

```
1: function checkNeighbours(i, j, label, cluster)
2:     if out_of_bounds(i, j) then
3:         return                                  ▷ indices not inside grid map
4:     end if
5:     if grid_map(i, j).points is empty then
6:         return                                  ▷ no points in current grid map cell
7:     end if
8:     if grid_map(i, j).label assigned then
9:         return                                  ▷ points of current grid map cell already labelled
10:    end if
11:    // Mark current cell to label and assign points to cluster
12:    grid_map(i, j).label ← label
13:    cluster assign grid_map(i, j).points
14:    // Check every neighbour cell (8-connectivity)
15:    for all (i_neighbour, j_neighbour) of (i, j) do
16:        checkNeighbours(i_neighbour, j_neighbour, label, cluster)
17:    end for
18: end function
```

bounding boxes covers the complete object, which is still fulfilled. At some point, it might be necessary to have better approximations for other objects, like circles etc., to drive through narrow passages. For our case, rectangle bounding boxes were sufficient enough in the first iteration to ensure no collisions and calculating other types of bounding boxes would increase the computational cost, which should be kept as low as possible.
We are using the OpenCV [Bra00] implementation *MinAreaRectangle*. All points assigned to a cluster are used to construct the minimal area rectangle. It uses the Sklansky's algorithm [Skl82] to find the convex hull of a 2D point set which has a complexity of $O(nlog(n))$ in the current implementation. A convex hull of a point set $\mathcal{S}$ is defined by the smallest convex polygon that contains all the points of $\mathcal{S}$ [Goo99]. It can be seen as putting a rubber band around the points. A polygon $P$ is convex if:

- $P$ is non-intersecting
- for any two points $p$ and $q$ on the boundary of $P$, segment $pq$ lies entirely inside $P$

To find the final minimal enclosing rectangle to the convex hull, a rotating caliper algorithm [Tou14, Ebe15] is employed. This method rotates a *caliper* around each edge of the convex hull, calculating the resulting rectangle enclosing all points. It finally returns the rectangle with the minimal area.

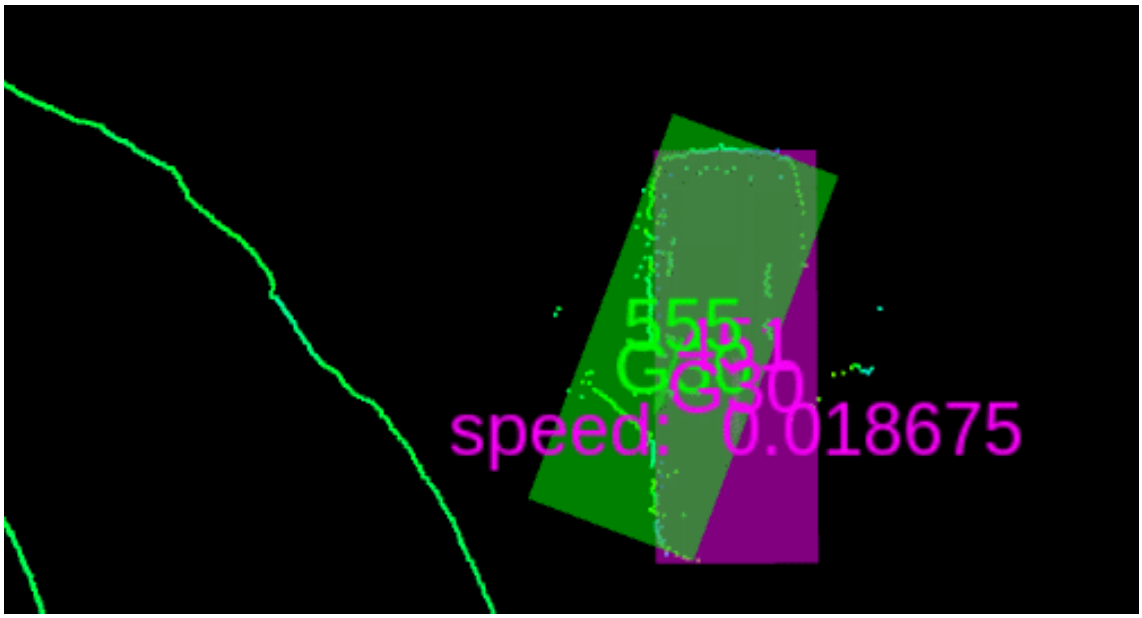



Figure 4.6: Drawbacks of the boundingbox and cluster based *Fast Localization* (green box). We have a car with an open door, sensed only from the back and from one side. The resulting minimal area rectangle is falsely aligned. The purple box is the correct vehicle position, calculated by the *Precise Localization*.

Another commonly used method is the principal component analysis (PCA) [DKKR09]. The idea is to identify the most significant directions (principal components) in a set of points. This approach is only a approximation and might not find the correct rectangle enclosing all points, especially if the density of points changes a lot.

**Drawbacks** This approach has some cases, where the resulting bounding box doesn't corresponds to the vehicle. Figure 4.6 shows that for e.g. "L-shaped" clusters, which result through occlusion or a poor sensor setup, the orientation is rotated up to 40° leading to a false localization hypothesis. Another scenario where the *Fast Localization* fails is when the vehicle is split up into several clusters, e.g. through occlusion, or cluster of multiple objects are merged together. Furthermore, this approach is highly affected by measurements noise, because we always force to have every point from the cluster inside of the boundingbox. In most cases, it still produces reliable, fast and accurate results, but can fail in some scenarios, making it not reliable enough to precisely control a vehicle. For this reason, a second localization approach, described in Sec. 4.6, to achieve a higher robustness and accuracy for vehicles, which is mandatory for controlling a vehicle.

### 4.5.3 Object Classification

Every object gets classified into two major classes, *vehicle* or *no vehicle*. The vehicle class divides further into classes for each known vehicle model. To assign a class to a localization hypothesis, the dimension (width and length) of the bounding box is used. The width and length of every vehicle model is known to the system in advance. A classification candidate for a object is chosen to be the vehicle model with the minimal euclidean distance $d = \sqrt{\Delta w^2 + \Delta l^2}$ between the width and length of the model and object. With $\Delta w = w_{\text{vehicleclass}} - w_{\text{object}}$ and $\Delta l = l_{\text{vehicleclass}} - l_{\text{object}}$. If the distance to the classification candidate exceeds a predefined threshold, it gets discarded and the object is labelled as *no vehicle*.
It is important to assign different vehicle models to the vehicle class for the next module, the *Precise Localization*, because it uses the vehicle model class to set the dimension of the car, which should not changed, and is highly dependent on the vehicle model.

### 4.5.4 Tracking and Data Association

The center of each localization bounding box is assumed to be the pose of an object. To estimate and predict a new target location for an object, for each object a kalman filter with a linear motion model is kept up to date. A kalman filter tracking an object is referred to as track, representing the object over time. The main purpose is to keep an ID for each object, we do not want to filter the localization hypothesis to create smooth output. The reasoning behind this is that the remote server will filter the results again and every time we filter, we might loose valuable information. Therefore every future module can decide on its own how much it wants to filter and smooth out the data.

**Data Association** Nearest neighbour search is employed to associate the localization hypothesis with the tracks. If there are multiple localization hypothesis assigned to the same track, the closest is used. The distance between a track and a localization hypothesis must not exceed a defined threshold for considering the localization as a candidate for the track. If a localization hypothesis is not assigned to any track, a new track is initialized. To calculate the distance, the pose as well as the dimensions from the bounding box is used. The defined distance threshold differs for vehicle objects and non vehicle objects, being more strict for objects classified as vehicles.

**Kalman Tracking** The kalman filter [TBF] is a powerful method to track and filter observations to achieve a more robust and stable result by recursively estimating and updating its state. It uses a linear prediction step to estimate the current state from the previous time step and a linear update step to correct the prediction. The kalman filter uses gaussian noise to model the errors in the system, observation and

transition. If the system is not linear, a extended kalman filter could be employed, which linearises the system. There are also more advanced kalman filter algorithms, capable of solving the non linearity.
A kalman filter is defined by the following vectors and matrices for each time step $k$:

- $\mathbf{x}_k$: the system state
- $\mathbf{z}_k$: the observed measurement
- $\mathbf{A}_k$: the state-transition model
- $\mathbf{H}_k$: the observation model
- $\mathbf{Q}_k$: the covariance of the process noise
- $\mathbf{R}_k$: the covariance of the observation noise
- $\mathbf{u}_k$: the control input
- $\mathbf{B}_k$: the control-input model

The prediction is calculated as follows:

$$\mathbf{x}_k = \mathbf{A}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k \tag{4.6}$$

A measurement $\mathbf{z}_k$ of the true state $\mathbf{x}_k$ is calculated as follows:

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \tag{4.7}$$

$\mathbf{w}_k$ is the process noise which is assumed to be drawn from a zero mean multivariate normal distribution, $\mathcal{N}$, with covariance $\mathbf{Q}_k$: $\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k)$.
$\mathbf{v}_k$ is the measurement noise which is assumed to be zero mean Gaussian white noise with covariance $\mathbf{R}_k$: $\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R}_k)$.
The current state of the filter is represented by two variables:

- $\hat{\mathbf{x}}_k$, the current estimated state at time $k$ given observations up to and including at time $k$;
- $\mathbf{P}_k$, the error covariance matrix of the current estimated state (a measure for the estimated accuracy) at time $k$

Figure 4.7 illustrates the recursive steps for a kalman filter with the input $u_k$ and $z_k$.
Following the states and models used to track the objects:

$$\mathbf{x} = \begin{bmatrix} x & y & \Theta & \dot{x} & \dot{y} & \dot{\Theta} \end{bmatrix}^\top$$

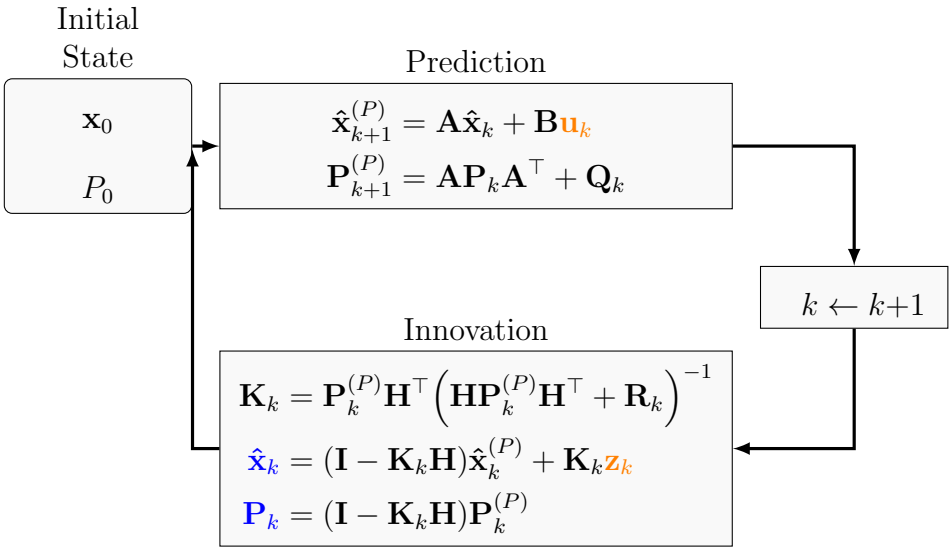



Figure 4.7: Illustration of the different steps done by a kalman tracker. It consists of the prediction step, predicting the current state $\hat{x}_k$ to $\hat{x}_{k+1}^{(P)}$ and update step, correcting the predicted state $\hat{x}_k^{(P)}$ to it's finale state $\hat{x}_k$ with covariance matrix $P_k$. Illustration adapted from [Tho].

$$\mathbf{z} = \begin{bmatrix} x & y & \Theta \end{bmatrix}^\top$$

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{Q} = \sigma_{process} * \mathbf{I}$$

$$\mathbf{R} = \begin{bmatrix} \sigma_{xy} & 0 & 0 \\ 0 & \sigma_{xy} & 0 \\ 0 & 0 & \sigma_\Theta \end{bmatrix}$$

Because we do not have to control input of the objects, $\mathbf{u}$ and $\mathbf{B}$ is not present in our model. $\sigma_{xy}$ and $\sigma_\Theta$ is set to a very low value(less than $0.01m$ and 0.01 respectively) to not filter the output. $\Delta t$ is set to $\frac{1}{f}$ with $f$ being the frequency of the module, which is defined by the sensor spinning rate.

## 4.6 Precise Localization

Because of the limitations of the *Fast Localization*, see Sec. 4.5.2, and the focus on preciously locating vehicles, because they are controlled by the system, a second approach using the same data is introduced. For each tracked vehicle it maintains a track. The *Precise Localization* module is independent on clustering algorithms by optimizing a rectangle into a restricted area. To restrict the area, prior information about the updating track of the system is used.
The *Precise Localization* initializes its tracks by listening to the object list from the *Fast Localization*. A new track is initialized by finding a new object classified as vehicle not corresponding to any already existing track. The tracks are getting update iteratively and takes all foreground points around the position of the track into account.

### 4.6.1 Initialization from Fast Localization

The *Precise Localization* can be seen as a module only maintaining existing tracks and is not responsible for finding new added objects. Because the *Fast Localization* always scans the complete area, new vehicles will also be included into its object list, which can be used to initialize new tracks for the *Precise Localization*.
To initialize a new track for the *Precise Localization*, the object list from the *Fast Localization* is monitored and gets checked for newly added objects. Only objects classified as a vehicle are getting considered. For each of these objects, it gets checked if it is already tracked. This can be done by comparing the object ID with the monitored track IDs. If the ID is found, the object is already tracked. If no track with the same ID is found, it still can have a corresponding track, because IDs can change in the *Fast Localization*-method. This is because we want to avoid assigning the same ID to a different object, making the assignment between localization hypothesis and tracks more strict, leading to sometimes assigning a different ID to the same object. To avoid also initializing a new track in the *Precise Localization* on such a scenario if we found an unknown ID, we check, if the position of the object (detected by the *Fast Localization*) with the unknown ID corresponds to a already monitored track. Correspondences are found by using the euclidean distance over $x, y$ and $\psi$ between the track and object. The distance has to be below a certain predefined threshold to be considered as the same object and if multiple corresponding tracks are found, the one with the lowest distance is used. If we match an object to a existing track, we add the new ID from the object to the track, making it possible to e.g. notify the server that the ID of this object from the *Fast Localization* has changed. Future objects from the *Fast Localization* with this ID can then again be matched directly to this track by comparing the IDs. This does not mean that the ID of the track changes, but multiple IDs from the *Fast Localization* can be assigned to the same track. Each track has a main ID which is set on the creation of the track and kept fix. A pseudo code for this can be found in

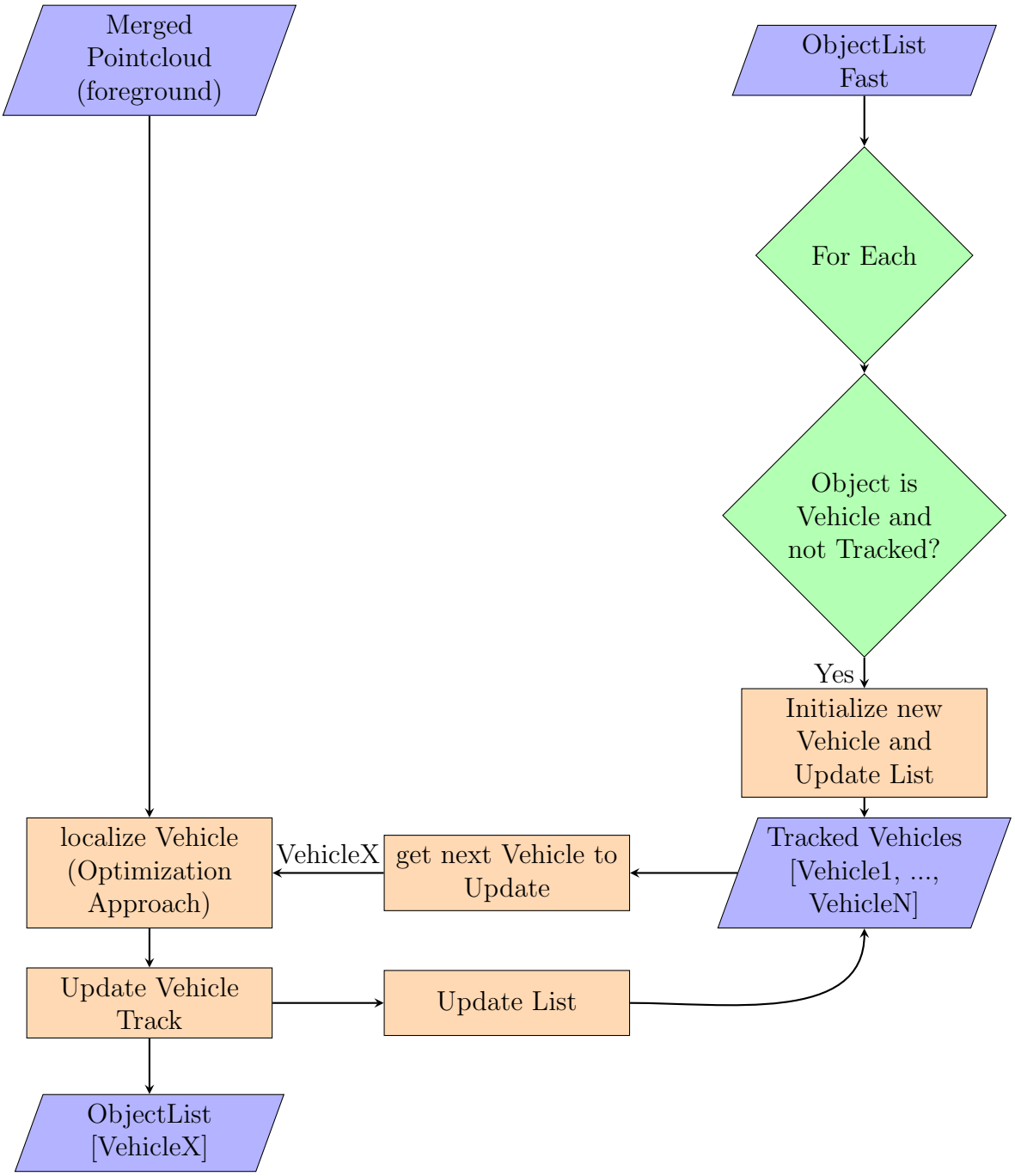



Figure 4.8: Precise Localization Algorithm

Alg. 5. If no corresponding track is found a new track with the position set to the position of the object is initialized.

**Algorithm 5** CheckFastObjectList

```
1: function CheckFastObjectList(object_list)
2:     for all object in object_list do
3:         if object.class equals vehicle then
4:             if not idTracked(object.id) then
5:                 tracked_object ← trackedObjectAtPose(object.pose)
6:                 if tracked_object is empty) then
7:                     tracks.append(newTrack(object))
8:                 else
9:                     tracked_object.addId(object.id)
10:                end if
11:            end if
12:        end if
13:    end for
14: end function
```

### 4.6.2 Retrieving Points to Update a Track

Tracks are updated iteratively, cycling through the list over and over again. Because each track is updated independently, multiple tracks can be updated at once utilizing multi-threading. To update a track, only a subset of all foreground points from the foreground pointcloud $\mathcal{C}_{\text{foreground}}$ are processed, denoted with $\mathcal{C}_{\text{optimization}}$. A rectangle is defined by a pose and size parameters $R(x, y, \psi, w, l)$ with $x, y, \psi$ representing the pose of the midpoint and $w, l$ the width and length of the rectangle. The points in $\mathcal{C}_{\text{optimization}}$ are chosen by setting a rectangle $R$ around the current predicted track position $(x_{\text{track}}, y_{\text{track}}, \psi_{\text{track}})$ and retrieving all points which fall into $R$. A point $p$ is inside $R$ if it lies on the area of $R$. The size $(w, l)$ of $R$ varies, depending on the time passed since the last successful update for this track, the velocity as well as on the dimension $(w_{\text{vehicle}}, l_{\text{vehicle}})$ of the vehicle model class. An illustration for the rectangle areas to retrieve the points for two tracks can be seen in Fig. 4.9. Each track gets its own subset of points.

We tested updating multiple tracks at once if the search spaces of different tracks overlapped. The idea was to receive all points from the tracks with the overlapping search spaces, and iteratively locate one car and remove the corresponding points from the search space. This failed on many scenarios by optimizing the first position over both cars, resulting in wrong localizations.

Trusting on the track and using its position to retrieve the points yields good results, especially on the cases where two cars are close to each other, because the optimization function does not see a lot if any measurements points of the other car. This

takes as assumption that we drive relatively slow on such scenarios. Otherwise, we couldn't trust on the tracks position that much.
The size of the rectangle $R$ to retrieve the points from the foreground pointcloud $\mathcal{C}_{\text{foreground}}$ for a track is calculated as follows:

$$\begin{aligned} m_{\text{time}} &= 1\frac{1}{c_{\text{time}}^2}(\Delta t)^2 \\ m_{\text{speed}} &= 1 + \frac{1}{c_{\text{speed}}}|v| \\ m &= cm_{\text{time}}m_{\text{speed}} \\ w &= w_{\text{vehicle}} * m \\ l &= l_{\text{vehicle}} * m \end{aligned} \tag{4.8}$$

$c_{\text{time}}$, $c_{\text{speed}}$ and $c$ are constants controlling how much each factor shall influence the size of the box. The factor $c$ simulates the localization error of the tracks. The speed $v$ is obtained by $v = \sqrt{\dot{x}^2 + \dot{y}^2}$. $\dot{x}$ and $\dot{y}$ are parameters of the track. $\Delta t$ is the time passed since the last update of the track.
This leads to the subset of points for the optimization function:

$$\mathcal{C}_{\text{optimization}} = \{p | p \in \mathcal{C}_{\text{foreground}}, p \text{ inside } R(x_{\text{track}}, y_{\text{track}}, \psi_{\text{track}}, w, l)\} \subseteq \mathcal{C}_{\text{foreground}} \tag{4.9}$$

A point $p$ is inside $R$ if it lies on the area of $R$. Alg. 6 illustrates the steps. To retrieve the points, the PCL library was used.

**Algorithm 6** RetrieveOptimizationCloud

```
1: function RetrieveOptimizationCloud(track, foreground_cloud)
2:     mult ← 1 * const_factor * time_mult * speed_mult as Eq. 4.8
3:     rec ← rectangle(track.pose, track.dim.length * mult, track.dim.width *
   mult)
4:     opt_cloud ← filter(rec, foreground_cloud)
5:     return opt_cloud
6: end function
```

We use the parameters $c_{\text{time}} = 4$, $c_{\text{speed}} = 5$ and $c = 1.2$.

### 4.6.3 Dimension Reduction by Line Fitting

A common approach to detect vehicles is to rely on finding straight-edge features or corners in the data and infer vehicle positions from these [MDG+05, NSMH06]. Following this, we want to detect and fit a line into the measurement data retrieved from the previous step and assume, that this line represents one side of the car. Because we have possibly up to four or even more strong lines in the data, a simple RANSAC approach, trying to fit one line on the complete data set, is not efficient

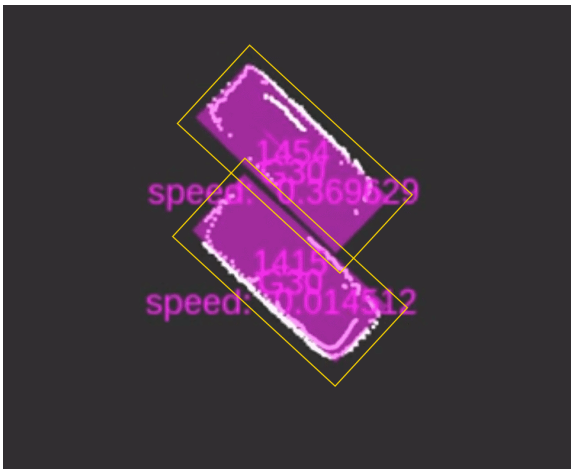


Figure 4.9: Used points to update a track. We have two tracks (purple box). All points lying in the orange rectangle are used to update the track covered by the rectangle. The pose of the rectangle is set to the track position and the size depends on the speed and time since the last update of the track.

because most of the measured points do not even correspond to the line we are trying to find. E.g. if we have a complete Outline of the vehicle, we have four different strong lines we can find in this data set, however only points lying on one side of the vehicle will form a line. The used method to find lines is the Hough Line Transform, which is capable of finding multiple lines and is a voting approach. Following a describtion of the Hough Line Transform and how we choose the strongest line.

**Hough Line Transform** is a voting algorithm which transforms data into the Hough space. Lines can be expressed in different ways:

$$f_1(x) = mx + t \tag{4.10}$$

$$f_2(x) = -\frac{cos(\theta)}{sin(\theta)} * x + \frac{r}{sin(\theta)} \tag{4.11}$$

For the Hough Transforms, we will express lines with the function $f_2(x)$, see Eq. 4.11 and the parameters $\theta$ and $r$, because it can represent vertical lines without having one parameter going to infinity. A visualization can be seen in Fig. 4.10. These two variables define the hough space with the parameters $x$ and $y$ from the Cartesian system.
A line in the Cartesian space corresponds to a point in the Hough space and vice versa a point in the Cartesian space corresponds to a line in the Hough space. The line in the Hough space represents all possible parameters for lines which go through the given point in the Cartesian system. By arranging the terms we get:

$$r = x * cos(\theta) + y * sin(\theta) \tag{4.12}$$

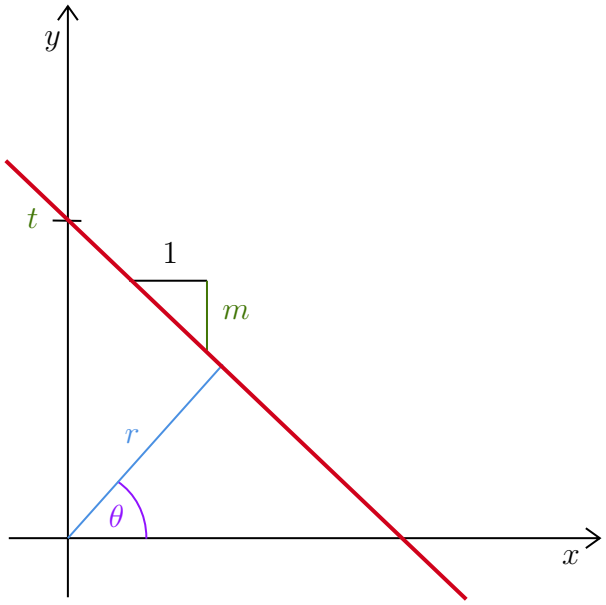



Figure 4.10: Different line representations. Common used is $f(x) = mx + t$. Hough Line Transform uses $f(x) = (-\frac{cos(\theta)}{sin(\theta)})x + (\frac{r}{sin(\theta)})$, because $m$ would goe to $\infty$ for vertical lines

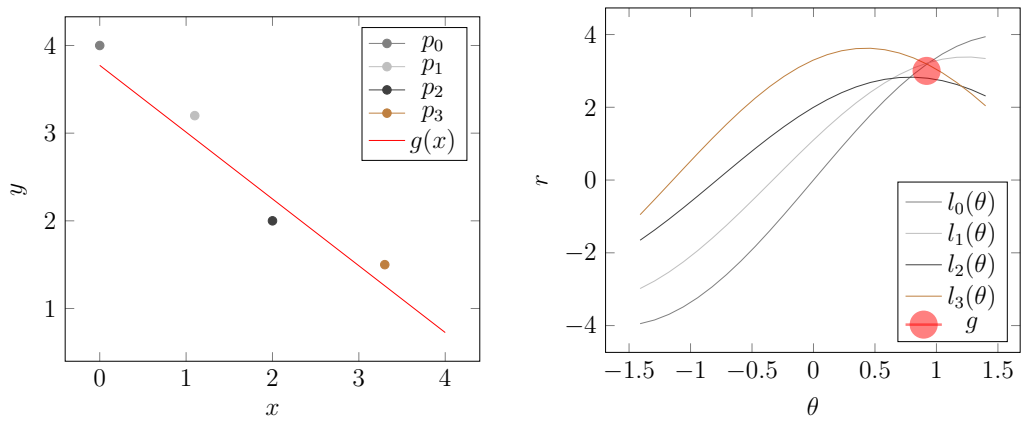
| Cartesian: | Hough: |
|---|---|
| $p_0(0, 4)$ | $l_0(\theta) = 4 * sin(\theta)$ |
| $p_1(1.1, 3.2)$ | $l_1(\theta) = 1.1 * cos(\theta) + 3.2 * sin(\theta)$ |
| $p_2(2, 2)$ | $l_2(\theta) = 2 * cos(\theta) + 2 * sin(\theta)$ |
| $p_3(3.3, 1.5)$ | $l_3(\theta) = 3.3 * cos(\theta) + 1.5 * sin(\theta)$ |
| $g(x) = -0.762 * x + 3.774$ | $g(0.92, 3)$ |

Table 4.1: Points and Equations used in Figure 4.11 for illustrating conversion between Hough and Cartesian systems.

For each point $\mathbf{p}$ $(x_0, y_0)$ we can define all lines that go through that point as:

$$r_\theta = x_0 * cos(\theta) + y_0 * sin(\theta) \tag{4.13}$$

Each pair $(r_\theta, \theta)$ represents a line that passes through $(x_0, y_0)$ by using the function $f_2(x)$, Eq. 4.11. Figure 4.11 shows an example of points getting converted to the Hough line space and vice versa using the Eq. 4.13 and 4.11. The intersecting point in the hough space then represents a line in the Cartesian space. The points and lines are presented in Tab. 4.1.
Because of discrete computation, the Hough coordinate system gets quantized, resulting in a two dimensional array $\mathbf{H}$, one dimension for $r$, the other for $\theta$. For each point, $r$ gets computed using Eq. 4.13. We let $\theta$ go from 0 to $\Pi$ with a predefined stepsize. For each computed $r$ and $\theta$, the value of the corresponding element in $\mathbf{H}$ gets incremented by one (representing the vote). Finally, all pairs of $r$ and $\theta$ which




(a) Cartesian coordinate system

(b) Hough coordinate system

Figure 4.11: Conversion between Cartesian and Hough coordinate systems. Points in Cartesian ($\mathbf{p_i}$) are represented as lines in the Hough space ($l_i(\theta)$). The intersecting point in the Hough space ($\mathbf{g}$) represents a line in the Cartesian space ($g(x)$). If all points would lie perfectly on the line, the intersection in the hough space would be explicitly through one point. A list of the Points and Equations is shown in Tab. 4.1.

received more votes (accumulated in $\mathbf{H}$) than a predefined threshold are returned as lines.
[GMK99] presents a probabilistic approach, minimizing the amount of computational steps needed, by only using a subset of all input points. This method was used with the openCV implementation [Bra00] to identify the strongest lines.

**Choose the strongest line** To evaluate all the returned lines, the number of points ($n_{\text{inliers}}$) representing the line $L$ as well as the length of the line is taken into account. For each line, the number of inliers ($n_{\text{inliers}}$) is calculated by taking the amount of points $\mathbf{p}$ from the input pointcloud $\mathcal{C}_{\text{optimization}}$ which distance $d$ to the line is less than a defined threshold. The threshold was set to $15cm$. To calculate the distance $d$ from a point $\mathbf{p}$ to a line which is defined by a start point $\mathbf{p_s}$ and an end point $\mathbf{p_e}$ we use the following equation [Wei]:

$$d = \frac{|(\mathbf{p_e} - \mathbf{p_s}) \times (\mathbf{p_s} - \mathbf{p})|}{|\mathbf{p_e} - \mathbf{p_s}|}$$

The score for each line is calculated using:

$$s(L) = n_{\text{inliers}} * (1 + \text{length}(L) * c_l)$$

With the weighting factor $c_l = 10$ and length($L$) a function which returns the length of the line $L$. The length of a line $L$ is defined by the euclidean distance between the start $\mathbf{p_s}$ and end point $\mathbf{p_e}$ of the line:

$$\text{length}(L) = |\mathbf{p_e} - \mathbf{p_s}|$$

It is important to take the length of the line into account, favouring the detection of a line on the side of the vehicle in regard to the front or back, because it is usually straighter. Longer lines also approximate the orientation better, because they are less affected by measurement noise. Because the density of the points in the input cloud changes a lot, more inliers does not necessarily mean we have a longer line.

Finally, with all inliers from the chosen line, the final line is calculated using an algorithm based on the M-estimator technique. It minimizes $\sum \rho(r_i)$ where $r_i$ is the distance between the $i^{th}$ point and the line. $\rho(r)$ is a distance function

$$\rho(r) = C^2 * (\frac{r}{C} - log(1 + \frac{r}{C}))$$

with $C = 1.3998$.

For this task, the openCV [Bra00] implementation HoughLinesP to retrieve all line candidates and fitLine to calculate the final line are used.

### 4.6.4 Model Fitting

With the help of the line, giving us some glues about the starting position and orientation of possible positions, we employ a optimization method to find the best rectangle. The dimension $(w_{\text{vehicle}}, l_{\text{vehicle}})$ of the rectangle is known through the classification of the *Fast Localization*, see Sec. 4.5.3, and kept fixed. We assume, that the line we found corresponds to either the front/back or to the side of the vehicle leading us to four different initial positions denoted as $x_{\text{init}}, y_{\text{init}}, \psi_{init}$. For each initial position, we optimize the rectangle by shifting it in the $x$ and $y$ direction. We keep the orientation from the initial position, to keep the computational cost low.

**Starting Positions** The four initial positions are found by using the mid-point coordinates $x$, $y$ and orientation $\theta$ of the detected line $L$. We assume that the line corresponds to either the front, back or one of either side of the vehicle. Therefore, the line should be part of one side of our rectangle. This leads us to our four initial positions dependent on the size of the track $(w_{\text{vehicle}}, l_{\text{vehicle}})$ to use to optimize the

rectangle:

$$
\begin{aligned}
p_{\text{init}} &= \begin{bmatrix} x_{\text{init}} \\ y_{\text{init0}} \\ \theta_{\text{init0}} \end{bmatrix} = \begin{bmatrix} x - \frac{w_{\text{vehicle}}}{2} * sin(\theta) \\ y + \frac{w_{\text{vehicle}}}{2} * cos(\theta) \\ \theta \end{bmatrix} \text{ left to the line} \\
p_{\text{init1}} &= \begin{bmatrix} x_{\text{init1}} \\ y_{\text{init1}} \\ \theta_{\text{init1}} \end{bmatrix} = \begin{bmatrix} x + \frac{w_{\text{vehicle}}}{2} * sin(\theta) \\ y - \frac{w_{\text{vehicle}}}{2} * cos(\theta) \\ \theta \end{bmatrix} \text{ right to the line} \\
p_{\text{init2}} &= \begin{bmatrix} x_{\text{init2}} \\ y_{\text{init2}} \\ \theta_{\text{init2}} \end{bmatrix} = \begin{bmatrix} x - \frac{l_{\text{vehicle}}}{2} * sin(\theta) \\ y + \frac{l_{\text{vehicle}}}{2} * cos(\theta) \\ \theta + 90° \end{bmatrix} \text{ behind the line} \\
p_{\text{init3}} &= \begin{bmatrix} x_{\text{init3}} \\ y_{\text{init3}} \\ \theta_{\text{init3}} \end{bmatrix} = \begin{bmatrix} x + \frac{l_{\text{vehicle}}}{2} * sin(\theta) \\ y - \frac{l_{\text{vehicle}}}{2} * cos(\theta) \\ \theta + 90° \end{bmatrix} \text{ infront of the line}
\end{aligned}
\tag{4.14}
$$

We optimize a rectangle on top of all four rectangles. The optimization parameters are the position of the rectangle $(x, y)$. The position of the rectangle is limited to be inside a defined range $[r_{\text{sideways}}, r_{\text{straight}}]$ of the starting position $p_{\text{init}}$ by using the vehicle dimensions.

$$r_{\text{sideways}} = c_w \frac{w_{\text{vehicle}}}{2} \tag{4.15}$$

$$r_{\text{straigth}} = c_l \frac{l_{\text{vehicle}}}{2} \tag{4.16}$$

Using $c_w = 0.9$ and $c_l = 1.2$ yielded good results. It is also possible to eliminate three of the four starting positions by choosing the one closest to the tracks position lowering the computational effort. However, we decided to optimize on all four positions regardless of the tracks position and favour the one position which corresponds to the tracks position to make it possible to recover from poor tracking results, making the system more robust. Because we retrieve the set of points from the tracks position as well, the optimization function mostly has a simple solution for the three wrong starting points. This is because e.g. if we choose the left starting point, but the real is the right one, all points in $\mathcal{C}_{\text{optimization}}$ will be on the right side of the starting point, pulling the rectangle to the right side till it hits the limitation by its range.

**Cost Function** The cost function is designed to get minimized.We use the following punishing rules for each point:

1. Punish points based on the distance to the rectangle borderline

2. Punish points lying outside of the rectangle

3. Punish points which corresponds to the previously detected line harder

With $\mathcal{C}_{\text{optimization}}$ being the Set of 2D foreground points retrieved as described in 4.6.2 and $L$ being the line we found in 4.6.3 we want to find the optimal rectangle $R(x, y, \psi, w, l)$ minimizing following function:

$$c(R(x, y, \psi, w, l), \mathcal{C}_{\text{optimization}}) = \sum\nolimits_{p_i \in \mathcal{C}_{\text{optimization}}} c_{\text{point}}(p_i, R(x, y, \psi, w, l)) \tag{4.17}$$

$$c_{\text{point}}(p_i, R(x, y, \psi, w, l)) = d(p_i, R(x, y, \psi, w, l)) * w(p_i, R(x, y, \psi, w, l)) \tag{4.18}$$

$$w(p_i, R(x, y, \psi, w, l)) = \begin{cases} 1 & \text{if } p_i \text{ is inside } R(x, y, \psi, w, l) \text{ and } p_i \text{ not inlier of } L \\ w_{\text{line}} & \text{if } p_i \text{ is inside } R(x, y, \psi, w, l) \text{ and } p_i \text{ inlier of } L \\ w_{\text{outside}} & \text{if } p_i \text{ is outside } R(x, y, \psi, w, l) \text{ and } p_i \text{ not inlier of } L \\ w_{\text{line}} * w_{\text{outside}} & \text{if } p_i \text{ is outside } R(x, y, \psi, w, l) \text{ and } p_i \text{ inlier of } L \end{cases} \tag{4.19}$$

$p_i$ inlier of L : $p_i$ was used to construct the line $L$

$p_i$ inside $R(x, y, \psi, w, l)$: $p_i$ lies inside the area of $R(x, y, \psi, w, l)$

$p_i$ outside $R(x, y, \psi, w, l)$: $p_i$ lies outside the area of $R(x, y, \psi, w, l)$

$d(p_i, R(x, y, \psi, w, l))$ is a distance function from the point $p_i(x, y)$ to the rectangle $R(x, y, \psi, w, l)$. Punishing points which correspond to the detected line makes sense, because we assume that the line is a borderline of the rectangle. This is especially important if something is near the vehicle, e.g. a human, obstacle, because we want to minimize the effect of them pulling the rectangle towards them such that the points lie inside the rectangle as well.

**Computing the Cost** To calculate the distances to a rectangle, the distance to each line segment of the rectangle gets calculated and the smallest distance is the distance to the borderline of the rectangle. To make this computational efficient, we rotate and translate all points in $\mathcal{C}_{\text{foreground}}$ to a new coordinate system where the origin corresponds to the mid-point of our starting rectangle and the sides are axis aligned. This means, that the orientation of the rectangle goes along the x-axis. Each point $p_i \in \mathcal{C}_{\text{foreground}}$ has to be transformed to the new coordinate system *rectangle system*. Points in the *rectangle system* are denoted with $p_i^R$. The transformed $\mathcal{C}_{\text{foreground}}$ with $\mathcal{C}^R_{\text{foreground}}$.

$$p_i^R = \begin{bmatrix} x_i^R \\ y_i^R \end{bmatrix} = \begin{bmatrix} cos(\psi) & sin(\psi) \\ -sin(\psi) & cos(\psi) \end{bmatrix} * (p_i - p_{mr}) \tag{4.20}$$

$p_{mr}$ is defined as the origin of the rectangle $R$ and $\psi$ the orientation of the rectangle $R$.

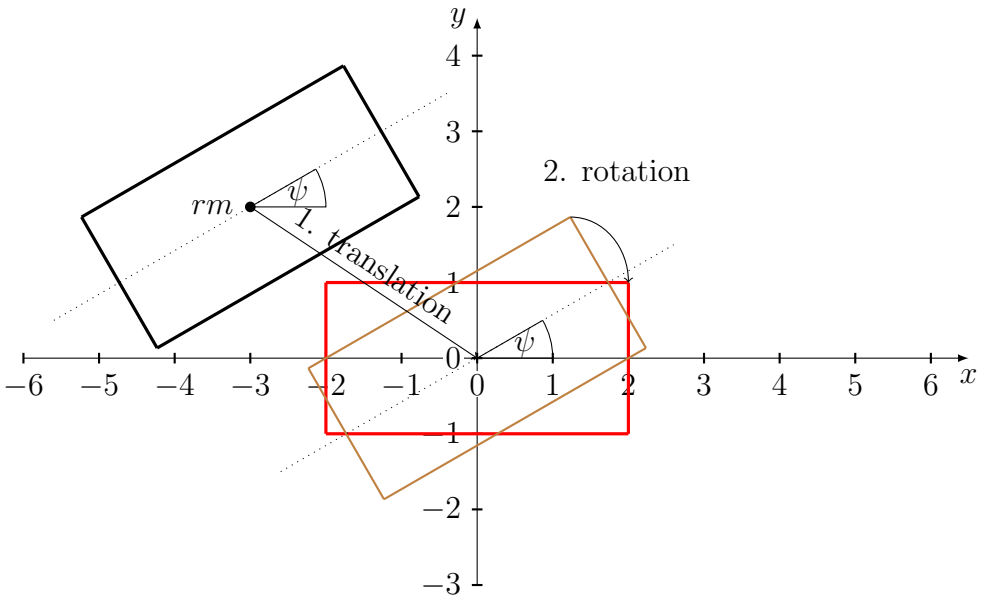



Figure 4.12: Transformation of the points such that the rectangle results in being centred in the origin and axis aligned. First we translate the mid-point $p_{rm}$ to the origin of the rectangle coordinate frame and then rotate all points by the negative angle $\psi$.

Through this transformation, we achieve a simple distance calculation for each point. Let $l_r$ be the length and $w_r$ the width of the origin centred axis aligned rectangle $R$. We want to find the distance function $d(x, y)$ from a point $p^R = \binom{x}{y} \in \mathbb{R}^2$ to the rectangle $R$. As shown in Fig. 4.13, we can have four different cases which we have to take into account:

1. Point $(p_0)$ lies inside of the Rectangle $R$.

2. Point $(p_1)$ lies outside of the Rectangle $R$ and the distance is orthogonal to the length side of the rectangle.

3. Point $(p_2)$ lies outside of the Rectangle $R$ and the distance is orthogonal to the width side of the rectangle.

4. Point $(p_3)$ lies outside of the Rectangle $R$ and the distance is not orthogonal to either side of the rectangle.

First, we want to find out into which case the point $p^R(x^R, y^R)$ falls. Because $R$ is axis aligned, we can compare the $x^R$ and $y^R$ values of $p^R$ with $w_r$ and $l_r$.

$$i_{\text{length}}(x^R) = \begin{cases} 1 \text{ if } -\frac{l_r}{2} \leq x^R \leq \frac{l_r}{2} \Leftrightarrow |x^R| \leq \frac{l_r}{2} \Leftrightarrow 0 \leq (\frac{l_r}{2} - |x^R|) \\ 0 \text{ else} \end{cases} \tag{4.21}$$

$$i_{\text{width}}(y^R) = \begin{cases} 1 \text{ if } -\frac{w_r}{2} \leq y^R \leq \frac{w_r}{2} \Leftrightarrow |y^R| \leq \frac{w_r}{2} \Leftrightarrow 0 \leq (\frac{w_r}{2} - |y^R|) \\ 0 \text{ else} \end{cases} \tag{4.22}$$

With $l(x^R) = (\frac{l_r}{2} - |x^R|)$ and $w(y^R) = (\frac{w_r}{2} - |y^R|)$ this simplifies to:

$$i_{\text{length}}(x^R) = 0 \leq l(x^R) \tag{4.23}$$

$$i_{\text{width}}(y^R) = 0 \leq w(y^R) \tag{4.24}$$

By evaluating every possible case of $i_{\text{length}}(x^R)$ and $i_{\text{width}}(y^R)$ we can calculate the distance $d(x^R, y^R)$ to the rectangle by reusing $l(x^R)$ and $w(y^R)$ shown in Tab. 4.2.

| $i_{\text{length}}(x^R)$ | $i_{\text{width}}(y^R)$ | Case | Eq. $d(x^R, y^R)$ |
|---|---|---|---|
| 1 | 1 | 1. | $d(x^R, y^R) = \min(l(x^R), w(y^R))$ |
| 1 | 0 | 2. | $d(x^R, y^R) = -w(y^R)$ |
| 0 | 1 | 3. | $d(x^R, y^R) = -l(x^R)$ |
| 0 | 0 | 4. | $d(x^R, y^R) = \sqrt{w(y^R)^2 + l(x^R)^2}$ |

Table 4.2: Distance equations depending on every possible case.

This results in the overall distance Equation:

$$d(x, y) = \begin{cases} \min(l(x^R), w(y^R)) & \text{if } i_{\text{length}}(x^R) = 1 \text{ and } i_{\text{width}}(y^R) = 1 \\ -w(y^R) & \text{if } i_{\text{length}}(x^R) = 1 \text{ and } i_{\text{width}}(y^R) = 0 \\ -l(x^R) & \text{if } i_{\text{length}}(x^R) = 0 \text{ and } i_{\text{width}}(y^R) = 1 \\ \sqrt{w(y^R)^2 + l(x^R)^2} & \text{if } i_{\text{length}}(x^R) = 0 \text{ and } i_{\text{width}}(y^R) = 0 \end{cases} \tag{4.25}$$

Because we are interested in shifting the rectangle $R$ around the origin to find the optimal position we have to extend the distance function $d(x, y)$ to also account for a shift of the original rectangle. This can be achieved by simply shifting the points into the opposite direction as the new rectangle:

$$d_{\text{shift}}(x^R, y^R, x_{\text{shift}}, y_{\text{shift}}) = d(x^R - x_{\text{shift}}, y^R - y_{\text{shift}}) \tag{4.26}$$

We want to derive a cost function only dependent on our transformed optimization $\mathcal{C}^R_{\text{optimization}}$ and the values $x_{\text{shift}}, y_{\text{shift}}$. Adapting Eq. 4.17 to use the transformed cloud $\mathcal{C}^R_{\text{optimization}}$ leads us to:

$$c(x_{\text{shift}}, y_{\text{shift}}, \mathcal{C}^R_{\text{optimization}}) = \sum\nolimits_{p_i^R \in \mathcal{C}^R_{\text{optimization}}} c_{\text{point}}(x_{\text{shift}}, y_{\text{shift}}, p_i^R) \tag{4.27}$$

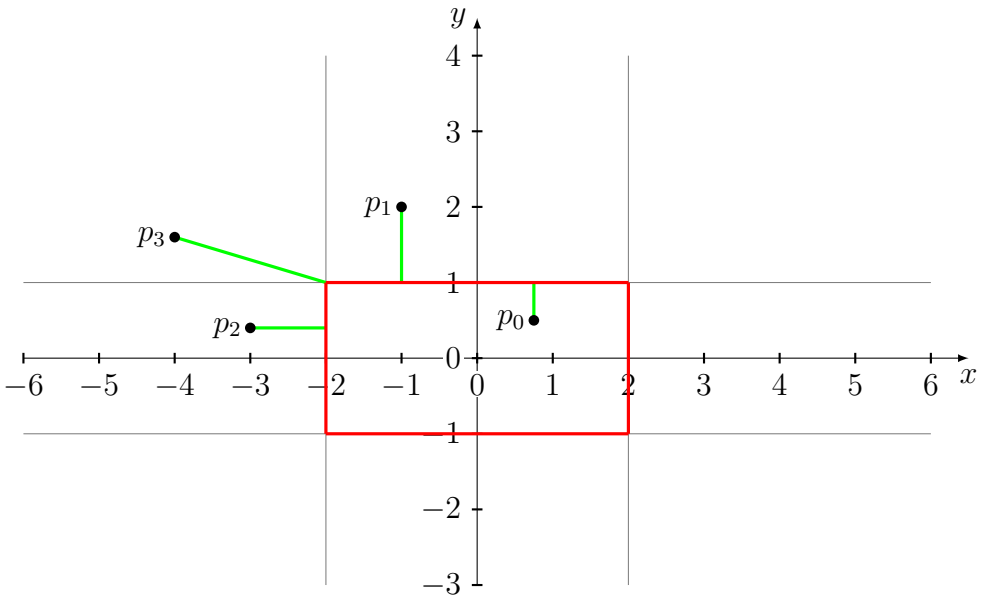



Figure 4.13: Distance calculation for points to an axis aligned and centred rectangle. There are four different cases which have to be taken into account, presented by each point $p_0 \dots p_3$.

Function $c_{\text{point}}(x_{\text{shift}}, y_{\text{shift}}, p_i^R)$ is derived from Eq. 4.18 by using the distance function 4.26 combined with the weight function 4.19:

$$c_{\text{point}}(x_{\text{shift}}, y_{\text{shift}}, p_i^R) = d(x_i^R - x_{\text{shift}}, y_i^R - y_{\text{shift}}) w_{\text{outside}}(p_i^R) w_{\text{line}}(p_i^R) \tag{4.28}$$

$$w_{\text{outside}}(p_i^R) = w_{\text{outside}} + (i_{\text{length}}(x_i^R) i_{\text{width}}(y_i^R)(1 - w_{\text{outside}})) \tag{4.29}$$

$w_{\text{outside}}(p_i^R))$ returns either 1 or $w_{\text{outside}}$ depending if $p_i^R$ lies inside the area of the rectangle or not. This can be merged into the distance calculation Eq. 4.25.
$w_{\text{line}}(p_i^R)$ returns $w_{\text{line}}$ if $p_i$ was used to construct the line $L$, otherwise it returns 1. All points used to construct the line are marked accordingly during the line fitting process making it easy to detect these points during the optimization process.

**Optimization Algorithm** For the optimization algorithm a derivative free and global approach, a modified version of the Controlled Random Search (CRS2) proposed by [KA06] is used. It used less evaluations of the cost function as an evolutionary optimization approach [RY05], which was found to require ($\approx 700$) in regard to the CRS2 ($\approx 500$) evaluations.
Following a brief introduction to the CRS2 algorithm: The controlled random search algorithm is a direct search technique and is purely heuristic. It starts by filling a set $S$ with $N$, $N \gg n$, sample points distributed uniformly over the search space $\Omega$. For us, $\Omega$ is defined by limiting the search space using the Eq. 4.15 and 4.16:

$$\Omega = \{p \in \mathbb{R}^2 | |p.x| \leq r_{\text{straigth}}, |p.y| \leq r_{\text{sideways}}\}$$

**Algorithm 7** optimizePosition

**function** OPTIMIZEPOSITION(*opt_cloud*, *track*)
  $length_{rec} \leftarrow track.dim.length$
  $width_{rec} \leftarrow track.dim.width$
  $best\_line \leftarrow fitLine(opt\_cloud)$
  **if** *best_line* is empty **then**
    **return** ▷ no update for the track this time
  **end if**
  $initial\_positions \leftarrow getInitPosForLine(best\_line)$ ▷ Eq. 4.14
  $range\_x \leftarrow cl * track.dim.length/2$ ▷ Eq. 4.16
  $range\_y \leftarrow cw * track.dim.width/2$ ▷ Eq. 4.15
  $opt\_positions \leftarrow$ empty array

  **for all** *initial_pos* in *initial_positions* **do**
    $opt\_cloud\_rec \leftarrow transform(opt\_cloud, initial\_pos)$ ▷ Eq. 4.20
    $opt\_params \leftarrow (opt\_cloud\_rec, length_{rec}, width_{rec}, range\_x, range\_y, max\_time)$
    $opt\_pos \leftarrow optimizeRectangle(opt\_params)$
    $opt\_positions.append(opt\_pos)$
  **end for**
  $best\_track \leftarrow chooseBestTrack(opt\_positions)$ ▷ Score and Corr. to *track*
  **return** *best_track*
**end function**

The sample set $S$ is gradually contracted by replacing the worst point $x_h$ with a better point $\tilde{x}$, called trial point. The trial point is obtained by forming a simplex with $n+1$ distinct points $x_1, x_2, \ldots, x_{n+1}$ where $x_1$ is the current best point and $x_2, \ldots, x_{n+1}$ are chosen randomly from $S$. The trial point is obtained by reflecting the point $x_{n+1}$ to the centroid of the other chosen points.

$$\tilde{x} = 2G - x_{n+1}$$

$$G = \frac{1}{n} \sum_{j=1}^{n} x_j$$

The trial point replaces the current worst point of the Set $S$ if it achieves a better score. Otherwise the trial point is discarded. This is repeated until a certain stopping condition is met, either a convergence or a certain amount of time has passed.
We use the algorithm CRS with Local Mutation. The difference to the normal CRS is that if the trial point $\tilde{x}$ does not outscore the worst point, it is not discarded but used to project it through the current best point $x_1$ generating a new trial point $\tilde{y}$. This is done by coordinate-wise reflection of $\tilde{x}$ through $x_1$ using following equation:

$$\tilde{y}^i = (1 + w_i)x_1^i - w_i\tilde{x}^i$$

$\tilde{y}^i$, $\tilde{x}^i$ and $x_1^i$ are the $i$th coordinate of $\tilde{y}$, $\tilde{x}$ and $x_1$ respectively. $w_i$ is a random number in $[0, 1]$ for each $i$. This leads to a global effect at the earlier stages when points in S are scattered and a local effect at later stages.
The NLopt library [Joh] implementation was used.

### 4.6.5 Tracking and Data Association

For each vehicle a kalman filter is maintained. Ideally a extended kalman filter with a motion model corresponding to the vehicle would be used. Because of implementation time a normal linear kalman filter as used by the *Fast Localization* 4.5.4 was employed. The main purpose of the tracking is keeping again to keep the IDs as well as having some clues to clever search the area and limit the input set for the *Precise Localization*. It is important to notice that we do not want to filter the localization hypothesis to have a smooth output, because this is done in the remote server with the help of the odometry and the *Fast Localization* results, resulting in a more stable and better filtering result. The initial state is the localization hypothesis received from the *Fast Localization* module which initialized a new kalman filter.
Because each track is updated one by one and the region from the predicted track is used to generate the localization hypothesis, we assume that the found vehicle corresponds to the used track. If no localization hypothesis was found, which means we didn't detect a line in the selected region, no correction step is employed.

## 4.7 Fusion of the Objectlists (remote server)

The remote server fuses the two different object lists and the odometry from the cars to achieve more accurate, stable and robust poses for the controlled car. For this purpose a extended Kalman Filter is used. The remote server is also responsible to detect the real orientation of the vehicles, as the localization module does not distinguish between the front or the back of the car. For each localization hypothesis, the real orientation $\theta_{real}$ can either be the estimated $\theta$ or $\theta + 180°$. This is done by comparing the odometrys, where the speed is either positive or negative, depending on the moving direction, to the localization hypothesis and searching for correspondences. After some initial time, it the fixes the orientation of the track such that it corresponds to its matched odometry and then keeps it. Every localization hypothesis matched to the track then gets corrected to match the track orientation. The implementation was not part of this thesis.

## 4.8 Advantages of two Parallel Localization Approaches

Having two independent localization approaches gives us, next to more robustness, also the possibility to detect objects near a car. Through exchanging data between the two methods, we can detect if an object from the *Fast Localization*, not labelled as a vehicle, is lying on top of a vehicle detected by the *Precise Localization*. This can result e.g. if the car door is opened or a person/obstacle is near the car. The algorithm can be obtained here 8. We buffer the localizations from the *Precise Localization* module and on receiving an object list from the *Fast Localization* module, we check if there is an object not labelled as vehicle is on top of an object from the buffered vehicle positions from the *Precise Localization* module. If so, we remove the object from the *Fast* object list. We can then search at that area for objects again, but remove the points corresponding to the *Precise* vehicle position.
By having two different and independent approaches, we are able to detect the correctness of the two localization modules in the remote server. We have three different measurements, the *Fast* localization, the *Precise* localization and the odometry received from the car. A integrity check as proposed by [SL15] can be made. This might be necessary, as the *Fast Localization*module can produce false localization depending on the data, see Sec. 4.5.2, and the *Precise Localization* module is dependent on finding the correct line. Especially the first leads to constantly false localization, if the car is not moving. By filtering these incorrect measurements before feeding them into the kalman filter, we can avoid creating bias in the final result.

**Algorithm 8** nearCarObstacleDetector

```
1: function nearCarObstacleDetector(foreground_cloud, object_list)
2:     if object_list from Precise Localization then
3:         updatePreciseLocBuffer(object_list)
4:         return
5:     end if
6:     for all object in object_list_fast do
7:         if object is NOT vehicle then
8:             for all precise_vehicle in object_list_precise do
9:                 if precise_vehicle overlaps object then ▷ Bounding boxes overlap
10:                    Remove object from object_list_fast
11:                    // get points used for object (cluster points)
12:                    near_car_cloud ← foreground_cloud at object.bounding_box
13:                    // remove points which correspond to precise_vehicle
14:                    near_car_cloud filter out precise_vehicle.bounding_box
15:                    // redo Fast Localization on sub cloud
16:                    add detected objects in near_car_cloud to object_list_fast
17:                end if
18:            end for
19:        end if
20:    end for
21: end function
```

# Chapter 5

# Experiment and Results

This chapter covers the experimental setup in Sec. 5.1 and the results in Sec. 5.2.

## 5.1 Setup

The system is used on an outside area of around $80m$ length and $40m$ width, shown in Fig. 5.1. We are using five Quanergy M8 LIDAR sensors [Qua] mounted on tripods. The sensors were used at a spinning rate of $20Hz$. The sensors are mounted on a height between $0.5m$ and $1m$ depending on the ground panel of the area at the sensor location. The ground panel elevates to the mid line of the experimental area like a saddle roof. To achieve the best overview of the area, they where mounted horizontal aligned such that they can scan over the highest point of the area with the 0° (2. Layer) laser scanner, illustrated in Fig. 5.2.

**Quanergy M8 Sensor** The Quanergy M8 LIDAR system consists of eight 2D line scanners mounted on a spinning head which can spin at a rate between $5Hz$ and $20Hz$. The eight lasers cover a 20° vertical field of view (FOV) and the entire unit rotates to give a full 360° horizontal FOV. An overview of the important specifications can be obtained in Tab. 5.1 using the datasheet of the manufacture [Qua]. Quanergy provides a ROS-interface for the sensors which outputs the sensor data as organized pointclouds.
M.-A. Mittet *et. al* [MNR+16] evaluated the Quanergy M8 Sensors range accuracy in 2016. At that time, Quanergy stated a range accuracy of $< 5cm(1\sigma$ at $50m)$, which is $2cm$ more than the $3cm$ according to the current datasheet (2018). M.-A.Mittet *et. al* found out that for outdoor measurements, this value is exceeded before $50m$. According to them, this error is reached at a distance of $11m$. They also found out that the sensor provides different results for indoor and outdoor studies in terms of measurement error. However, it is important to note that the used sensor for this study was one of the first versions released.
The position of the sensors in regard to the world coordinate frame with the origin in the bottom left corner can be obtained from Fig. 5.3.

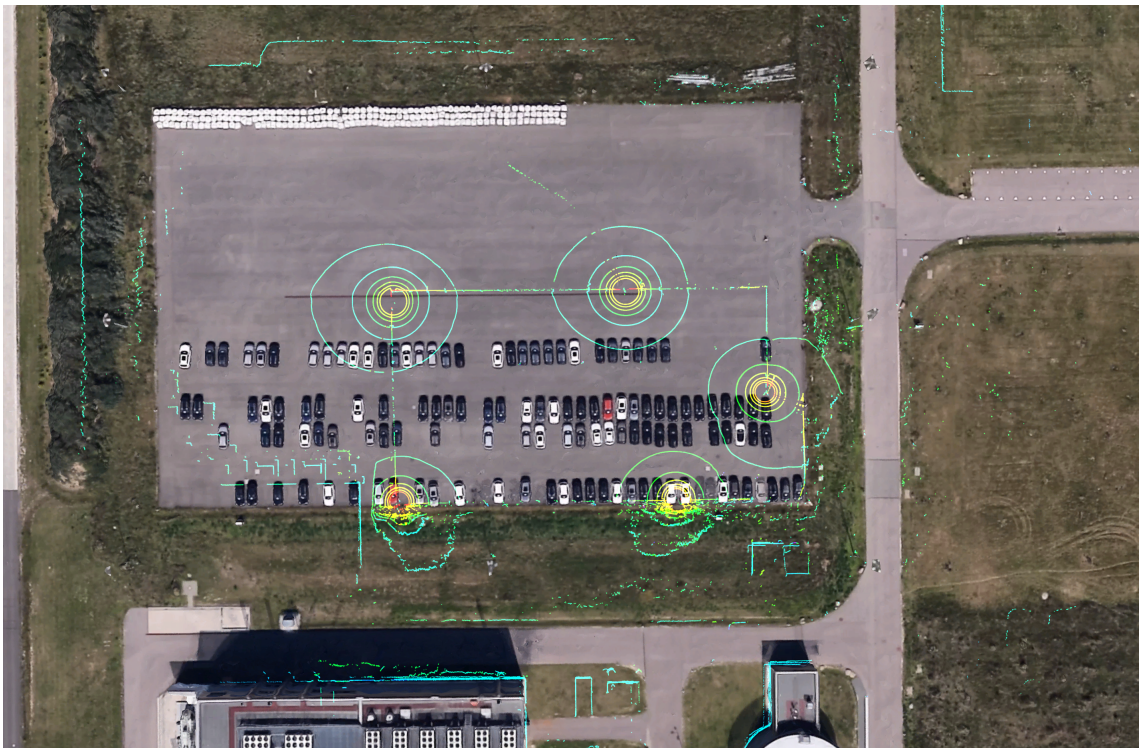


Figure 5.1: Outside area $\approx (80m \times 40m)$ where the experiment has taken place overlayed with the sensor measurements.


Figure 5.2: Illustration of Sensor Height in regard to the ground panel. Sensors are mounted such that they can scan horizontal over the complete area.

Each sensor spinning at a $20Hz$ rate outputs 8 layers with $\approx 2625$ points each. This means we are processing $\approx 8 * 2625 * 5 = 105000$ points each time step.

**Computational Power** For the experiments a Dell XPS laptop is used. The system does not utilizes any GPU-acceleration. The CPU of the laptop is an Intel Core i7-6700HQ CPU @ 2.6 GHz (up to 3.5GHz) and it has 8 GB RAM.

**Operating System and Programming Language** The operating system for the experiment is Linux Ubuntu 16.04 LTS with ROS-Kinetic. ROS, short therm for Robotic Operating System, is a flexible framework for writing robot software. It is a collection of tools, libraries and conventions that aim to simplify the development progress.

| Parameter | Specification |
|---|---|
| Wavelength | $905nm$ |
| Measurement Technique | Time of Flight (TOF) |
| Minimum Range | $1m$ (80% reflection) |
| Maximum Range | $> 150m$ (80% reflectivity) |
| Range Accuracy ($1\sigma$ at $50m$) | $< 3cm$ |
| Frame Rate (Update Frequency) | $5 - 20Hz$ |
| Angular Resolution | $0.003° - 0.2°$ dependent on frame rate |
| Detection Layers | 8 |
| Field of View (FOV) | Horizontal: 360°; Vertical: $20°(+3°/-17°)$ |
| Data Outputs | Angle, Distance, Intensity, Sync. Time Stamps |
| Returns | 3 |
| Output Rate | 420.000 points per second (1 return)<br>1.26M points per second (3 returns) |
| Operating Temperature | −20°C to +60°C (−4°F to +140°F) |

Table 5.1: Quanergy M8 Sensor Specification from manufacture [Qua]

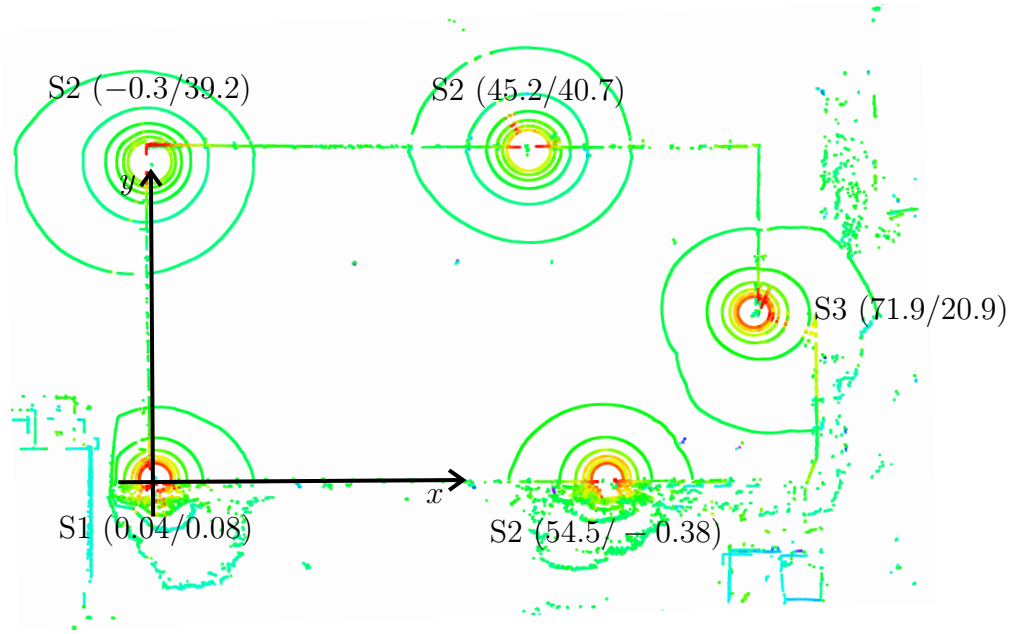



Figure 5.3: Sensor position and world coordinate system

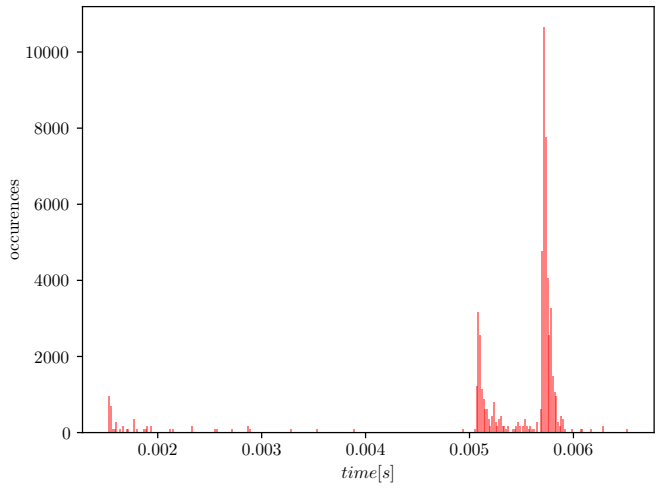



Figure 5.4: Duration of the background subtraction algorithm

Every method was implemented in C++ and built in release mode, using the ROS framework to enable the communication between the localization modules.

## 5.2 Results

Following the experimental results of the localization module. We evaluate the time consumption of the system, the positional accuracy of detected vehicles and the limiting cases for the system to produce reliable information. The accuracy is evaluated against human ground truth labelled data while the vehicle is not moving. Furthermore, the derivatives of the positions, speed and angular rate, are compared to the odometry of the car during a driving test. Finally, the localization hypothesis from the two localization modules are compared to the resulting position from the remote server, which fuses and filters the localization with the odometry of the car.

### 5.2.1 Time Consumption

**Pointcloud filtering and merging** Each sensor reading is labelled with the current system time when the pointcloud arrives.
The background subtraction is done parallel for each sensor. The processing time for the background subtraction with the sensors spinning at a rate of $20Hz$ is shown in Fig. 5.4. We achieve a mean time of $5.3ms$ and a standard deviation of $1ms$. In comparison, merging the pointclouds first and filtering the resulting unorganized pointcloud with a octree based spatial change detection, see Sec. 2.1, took around $100ms$.

| spinning rate | mean | std. deviation |
|---|---|---|
| $10Hz$ | $45.433ms$ | $11.442ms$ |
| $20Hz$ | $26.945ms$ | $5.830ms$ |

Table 5.2: Processing time for preprocessing (background subtraction and merging) the pointclouds compare to sensors spinning at $10Hz$ and $20Hz$.

The sensors are not time synchronized, resulting in the merging algorithm to wait for each sensor to output its measurements, see Sec. 4.3. This waiting time ($\approx 20ms$) takes up most of the time consumption for preprocessing the data, which is background subtraction and merging.
After these preprocessing steps we have the merged foreground pointcloud $\mathcal{C}_{\text{foreground}}$.

**Localization hypothesis and tracking** We measure the time difference between when the object list is present compared to the time stamp of the preprocessed pointcloud. This means we have to subtract the preprocessing time to get the time consumption of the localization modules.
It is important to mention that there is only one object present in the scene. The duration of the two localization modules are presented in Fig. 5.5.
The *Fast Localization* object list is in mean $27ms$ older than the used data. By subtracting the preprocessing time, we achieve a mean time of $1ms$ or less.
The *Precise Localization* object list is in mean around $78ms$ older than the used measurements. This results in an update time of around $52ms$ with a high std. dev. time of $41ms$. The high std. dev. comes because as soon as an update finishes, another update is done using the same pointcloud, if no new pointcloud arrived till then. This means that sometimes two updates are done with the same pointcloud. Only one update takes in mean $30.9ms$ with a std. dev. of $15.0ms$ to compute. It is still important to keep in mind that the duration for multiple tracks for the *Precise Localization* scales linear, as they are updated iteratively one by one in the current implementation. This is limited by the computational power and can be paralleled because every update of a track is independent to others.
The bottleneck to fasten the *Fast Localization* is the pointcloud merging. If the data would come in time synchronized, we could achieve a processing time of around $6ms$ which corresponds to a frequency of up to $166Hz$. The *Precise Localization* takes longer, which is fine because it was designed to be computational heavier and to support the *Fast Localization* for scenarios where it fails.

### 5.2.2 Localization Accuracy

Following the evaluation of the positional accuracy of the system. We employ three different tests. First, we research the noise output from the localization system if the vehicle is not move. A hand-labelled ground truth data is provided. We then compare the derivatives of the localization hypothesis with the odometry recieved

| Localization | mean | std. deviation |
|---|---|---|
| Fast Localization | 26.927$ms$ | 4.0668$ms$ |
| Precise Localization | 78.223$ms$ | 41.878$ms$ |
| Precise Localization (1 Iteration) | 55.929$ms$ | 15.030$ms$ |

Table 5.3: Time difference of the system time to the time stamp of the object list for the *Fast* and *Precise* localization module

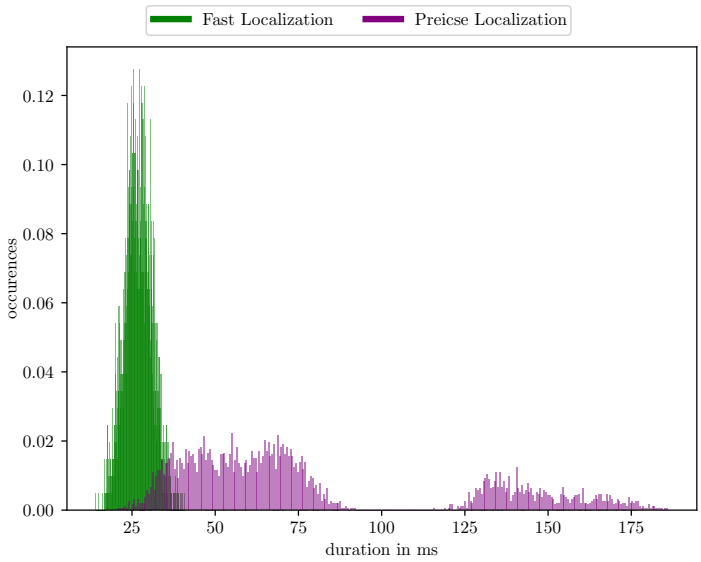



Figure 5.5: Duration of the *Fast* and *Precise* localization module. The two peaks from the *Precise* localization comes from doing iteratively two localization hypothesis with the same input.

from the car. Finally, we have a look at the difference between the filtered and fused localization from the remote server with the localization hypothesis of the proposed system.

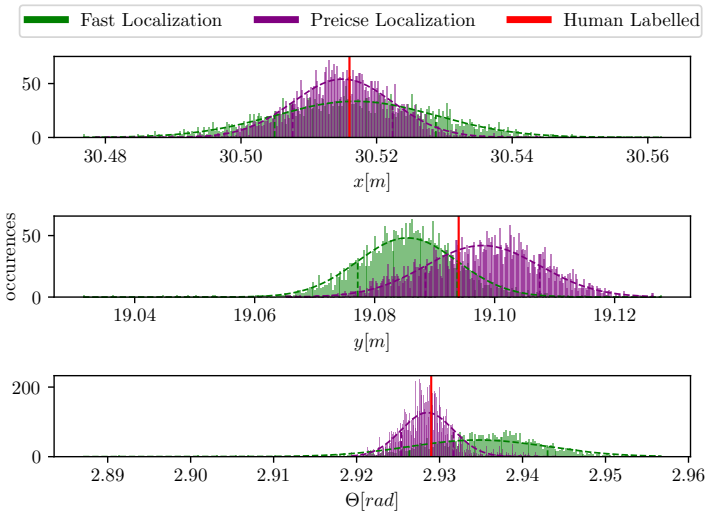
**Calibration errors** Because the software which should be employed to get the calibration of the sensors was not able to estimate the calibration on such high distances between the sensors, we had to overcome it by calibrating them by hand. We therefore assume that every sensor is perfectly horizontal (trying to achieve it with a level) and then calibrate the $x$, $y$ and $\theta$ for every sensor by hand, see Sec. 4.7. Because we can not align the sensors perfectly horizontal, the calibration has errors as well. These errors are noticeable as the located vehicle is sometimes bigger/smaller in e.g. the width then the actual width of the vehicle. We tried to keep the overall error as small as possible (fixing one side of the area makes the result on the other side worse), but as we drove over the complete area, they are noticeable. The resulting measurements for a car in the bottom right corner of the area can be obtained in Fig. B.4.

### Stationary tests

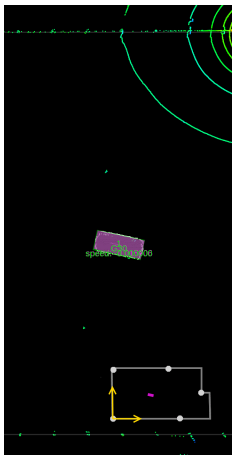
Following two tests are presented, where the vehicle was standing still and we collected the localization hypothesis over time.

**Test 1** The vehicle was placed in the middle of the area leading to an average distance between the vehicle and each sensor of $\approx 34m$. The mean and standard deviation of the estimated positions for the car can be seen in Fig. 5.6 and in Tab. 5.4. The distance from the vehicle to each sensor is shown in Tab. 5.5. We collected the localization hypothesises over a time period of $113s$. The *Fast Localization* returned 2281 measurements running at a frequency of $20Hz$. The *Precise Localization* returned 3504 measurements during the time period, resulting in a frequency of $31Hz$ for the module.

**Test 2** The vehicle was placed in the bottom right corner of the area. The vehicle is not surrounded with sensors from each side. The size of the purple box in Fig. 5.7(b) uses the vehicle model size parameters and is bigger in regard to the box size implied by the sensor data. This comes from the previously described calibration errors. This makes estimating the ground truth by hand labelling hard as well. The mean and standard deviation of the localization hypothesis can be seen in Fig. 5.7 and in Tab. 5.6. The distance from the vehicle to each sensor is shown in Tab. 5.7. We collected the localization hypothesises over a time period of 236. The *Fast Localization* returned 4735 measurements, running at a frequency of $20Hz$. The *Precise Localization* returned 4599 measurements during the time period, resulting in a frequency of $19Hz$.




(a) Histogram of the localization results collected over $113s$



(b) Position of the vehicle (middle of the area)

Figure 5.6: Histogram of localization results for a stationary vehicle (stationary test 1)

| Stationary test 1 | | $x[m]$ | $y[m]$ | $\theta[rad]$ |
|---|---|---|---|---|
| Ground Truth: | | 30.516 | 19.094 | 2.929 |
| *Fast Localization* | Mean $\mu$: | 30.517 | 19.085 | 2.935 |
| | Std. dev. $\sigma$: | 0.0119 | 0.0083 | 0.0083 |
| *Precise Localization* | Mean $\mu$: | 30.515 | 19.098 | 2.929 |
| | Std. dev. $\sigma$: | 0.0073 | 0.0095 | 0.0032 |

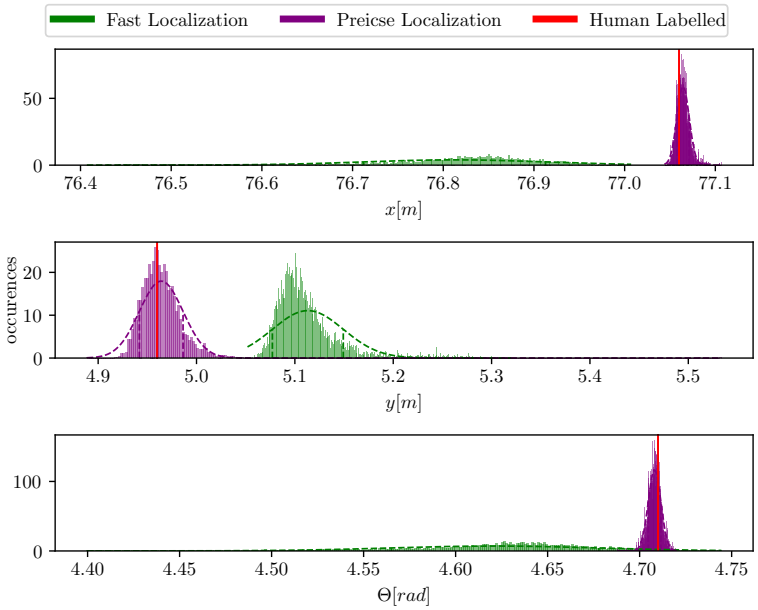
Table 5.4: Mean and std. dev. results for the *Fast* and *Precise* localization modules for a stationary vehicle (stationary test 1)

| Sensor: | S1 | S2 | S3 | S4 | S5 |
|---|---|---|---|---|---|
| Distance: | $36.1m$ | $30.9m$ | $41.4m$ | $26.1m$ | $36.98m$ |

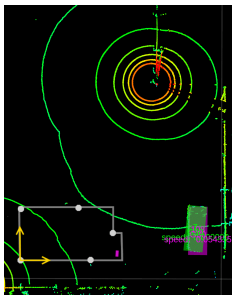
Table 5.5: Distance of the vehicle to each sensor (stationary test 1)

| Stationary test 2 | | $x[m]$ | $y[m]$ | $\theta[rad]$ |
|---|---|---|---|---|
| Ground Truth: | | 77.06 | 4.96 | 4.71 |
| *Fast Localization* | Mean $\mu$: | 76.812 | 5.113 | 4.621 |
| | Std. dev. $\sigma$: | 0.101 | 0.036 | 0.056 |
| *Precise Localization* | Mean $\mu$: | 77.064 | 4.964 | 4.708 |
| | Std. dev. $\sigma$: | 0.0068 | 0.0222 | 0.0034 |

Table 5.6: Mean and std. dev. results for the *Fast* and *Precise* localization for Stationary test 2




(a) Histogram of the localization results collected over $236s$



(b) Position of the vehicle (bottom right of the area)

Figure 5.7: Histogram of localization results for a stationary vehicle (stationary test 2)

| Sensor: | S1 | S2 | S3 | S4 | S5 |
|---|---|---|---|---|---|
| Distance: | $77.31m$ | $23.18m$ | $16.75m$ | $47.88m$ | $84.61m$ |

Table 5.7: Distance of the vehicle to each sensor for stationary test 2

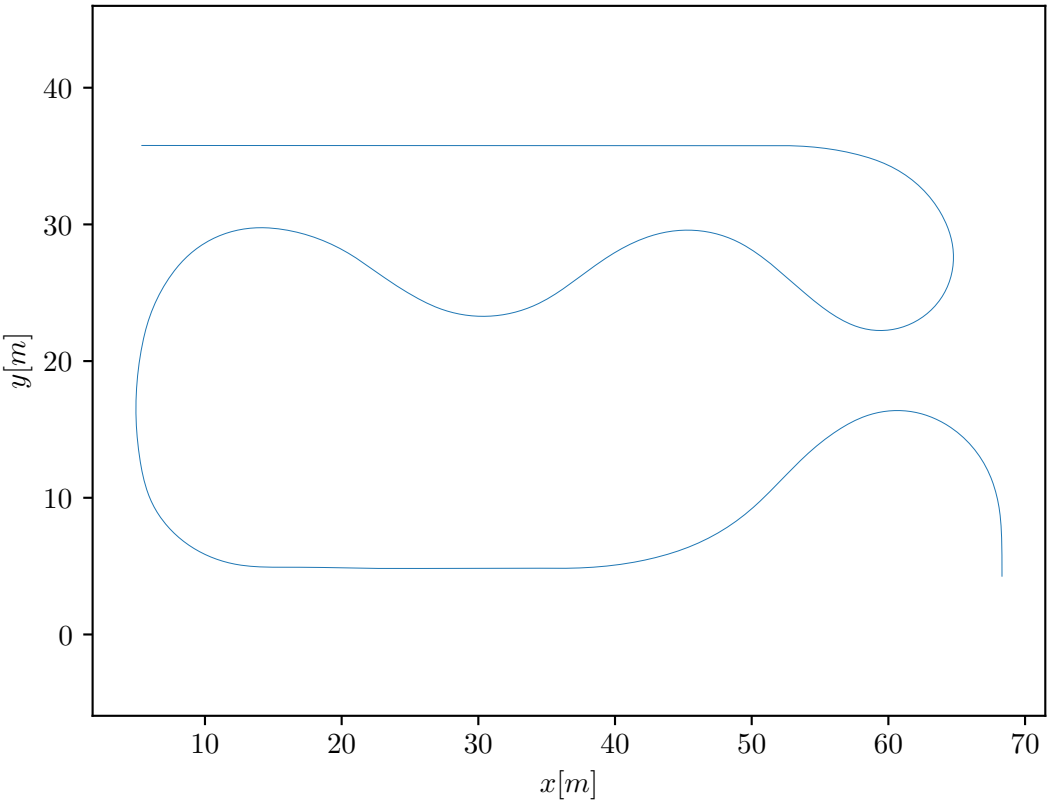



Figure 5.8: Path for the driving scenario. Start point is top left and end point is bottom right. The car shall drive forward with a speed of up to $4\frac{m}{s}$ on straight sections and $2\frac{m}{s}$ on curves.

### Comparision of Localization Results in Driving Scenario

Following the results of a test drive on the area. The path of the test drive can be obtained in Fig. 5.8. Start point is top left and end point is bottom right. It consists of straight sections where the car is allowed to drive up to $4\frac{m}{s}$ and manoeuvring parts where the car is allowed to drive a maximum of $2\frac{m}{s}$. We compare the results of the localization with the odometry by computing the derivatives of the localization results. Finally, we evaluate the unfiltered localization results from our system with the fused filtered results from the remote server. The remote server fuses the localization hypothesis of the *Fast* and *Precise Localization* module with the odometry of the car, see Sec. 4.7.

**Comparision to Odometry** We collected the localization hypothesis from the environment module, the odometry of the vehicle and the remote server output

(fused result from the environment module and the odometry). We use the odometry as ground truth. Odometry is precise in short terms, but the error of the odometry will accumulate in the absolute position $(x, y, \theta)$ over time. The standard deviation of the relative error is less than 5% [SPA15] when the speed is over $1.5\frac{m}{s}$. We mainly operate in a speed range between $1.8\frac{m}{s}$ and $4\frac{m}{s}$, leading to a standard deviation of $0.09\frac{m}{s}$ to $0.2\frac{m}{s}$.
The collected localization positions plotted over $x$ and $y$ can be obtained in Fig. 5.14. The derivatives of the localizations were calculated by using two following localization hypothesis and dividing the difference with the time difference $\Delta t$. The time difference $\Delta t$ has to be at least $20ms$. The speed is calculated by the euclidean distance of $\Delta x$ and $\Delta y$ divided by $\Delta t$. The estimated localization derivatives and the odometry of the vehicle recorded during the experiment can be obtained in Fig. 5.9 and Fig. 5.10.
The differences of the localization derivatives from the odometry, $\Delta v$ and $\Delta\dot{\Theta}$, are shown in Fig. 5.11 and the mean and standard deviation can be obtained in Tab. 5.8. For each localization derivative, we compare it to the odometry result by interpolating the odometry to match the timestamp with a linear function. The odometry is recieved at a refresh rate of $50Hz$.
Figure 5.10 and 5.11 shows that the deviation of the estimated orientation from the *Precise Localization* differs less to the odometry than the *Fast Localization*. This means the assumption to trust on the detected line and do not optimize on the orientation is valid. Sometimes, we experiment higher noise, which mostly appears when a vehicle corner was near a sensor and the density of measurement points on the corners of the car was really high. This sometimes leads to choosing the wrong line because of the high density of points on the corner. This means a line through the corner has a lot of inlier, making it possible to outscore the correct line. The effect is illustrated in Fig. 5.12 and a resulting wrong localizaiton hypothesis can be obtained in Fig. B.3.

**Comparison to the server output** Finally, we compare the unfiltered results from the infrastructure based localization with the filtered and fused results of the remote server. The localization hypothesis plotted over $x$ and $y$ can be obtained in Fig. 5.14. We simply calculate for each update step the difference $(\Delta x, \Delta y, \Delta\Theta)$ between the kalman filter result and the input localization hypothesis. The differences can be obtained in Fig. 5.13 and Tab 5.9. These values have to be considered with caution, because the filtered and fused result from the remote server is affected by the localization errors from the localization module.

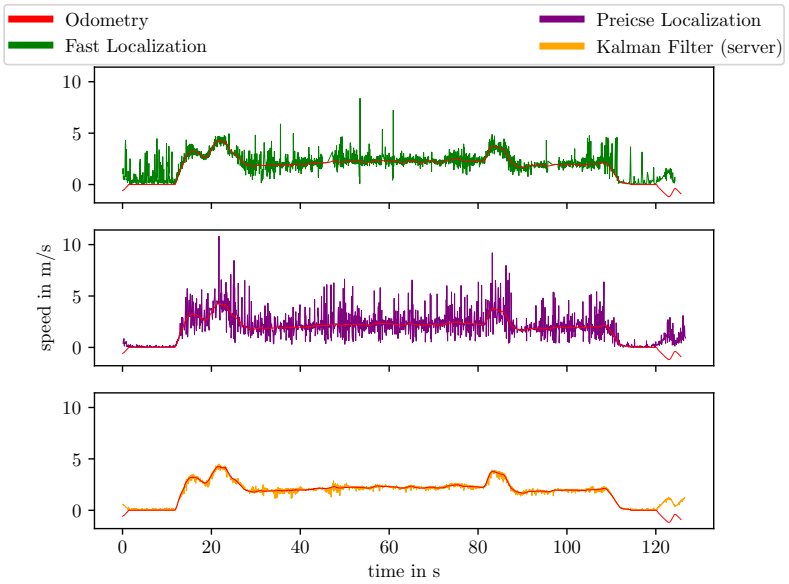



Figure 5.9: Speed plotted over time for the driving scenario. The speed for the localization modules got calculated at a rate of $20Hz$. The odometry (red) is used as ground truth.

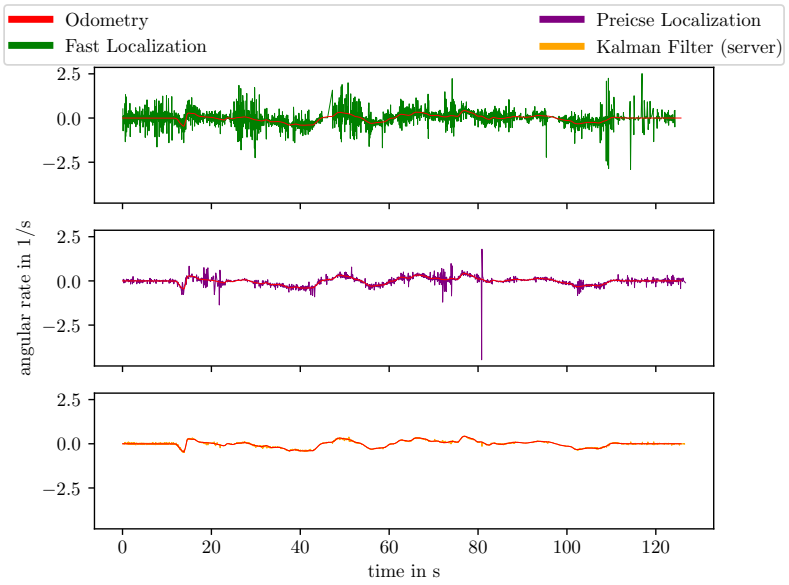



Figure 5.10: Orientation change (yaw rate) plotted over time for driving scenario. Orientation changes for the localization modules got calculated at a rate of $20Hz$. The odometry (red) is used as ground truth.

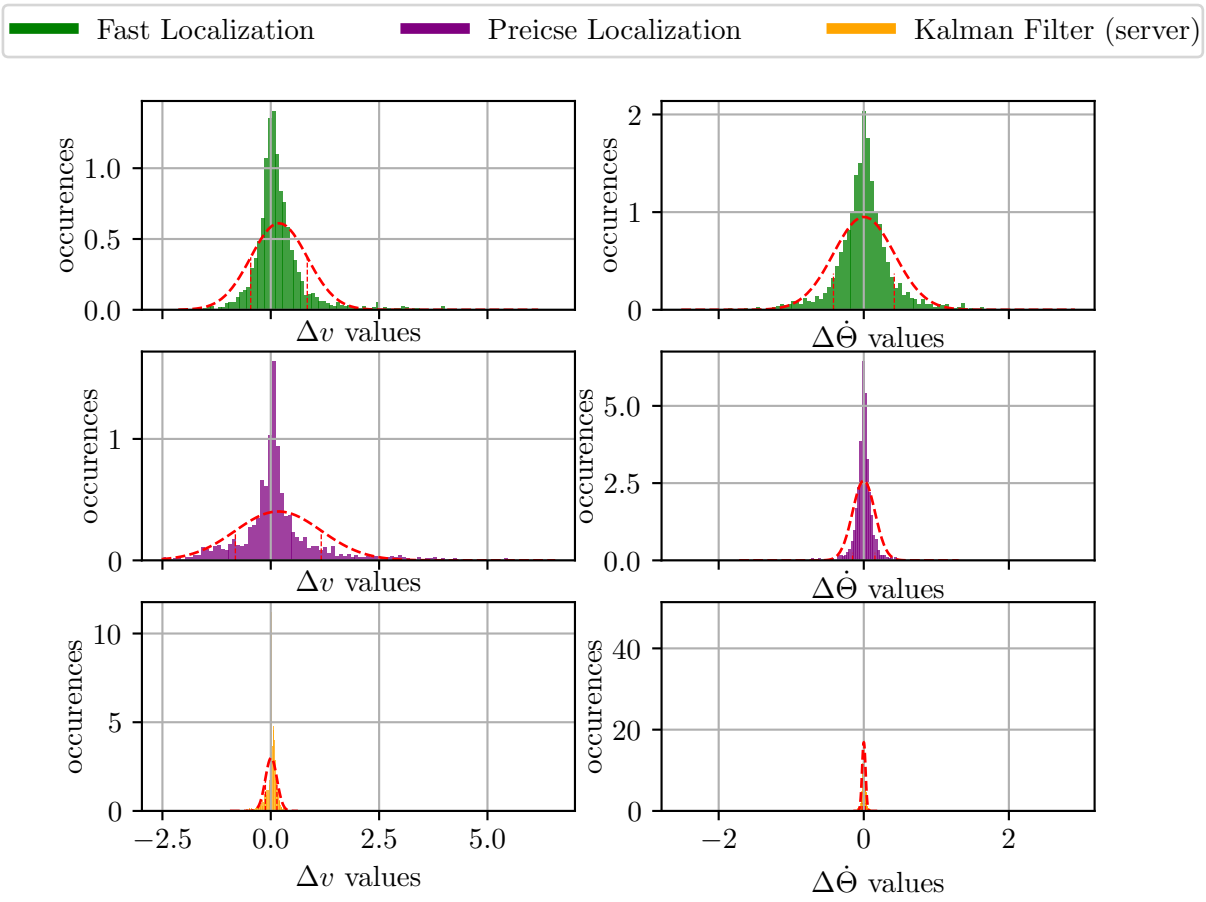



Figure 5.11: Histogram of the difference between the *Fast* and *Precise* localization derivatives compared to the odometry of the vehicle for the driving scenario.

| Module | $\mu_{\Delta v}[\frac{m}{s}]$ | $\sigma_{\Delta v}[\frac{m}{s}]$ | $\mu_{\Delta\dot{\Theta}}[\frac{rad}{s}]$ | $\sigma_{\Delta\dot{\Theta}}[\frac{rad}{s}]$ |
|---|---|---|---|---|
| Fast Localization | 0.1899 | 0.6526 | 0.00159 | 0.4192 |
| Precise Localization | 0.1739 | 0.9906 | −0.00047 | 0.1540 |
| Kalman Filter (server) | 0.0091 | 0.1332 | 0.00056 | 0.02359 |

Table 5.8: Mean ($\mu$) and std. dev. ($\sigma$) for the differences of the odometry compared to the derivatives of the *Fast* and *Precise* localization and the fused and filtered server results

| Module | $\mu_{\Delta x}[m]$ | $\sigma_{\Delta x}[m]$ | $\mu_{\Delta y}[m]$ | $\sigma_{\Delta y}[m]$ | $\mu_{\Delta\Theta}[rad]$ | $\sigma_{\Delta\Theta}[rad]$ |
|---|---|---|---|---|---|---|
| Fast Localization | 0.0656 | 0.2554 | −0.0515 | 0.1655 | 0.00419 | 0.04380 |
| Precise Localization | 0.0258 | 0.1483 | −0.0063 | 0.0936 | −0.0000325 | 0.02398 |

Table 5.9: Mean ($\mu$) and std. dev. ($\sigma$) for the differences of the filtered and merged server results compared to the *Fast* and *Precise* localization results for the driving scenario

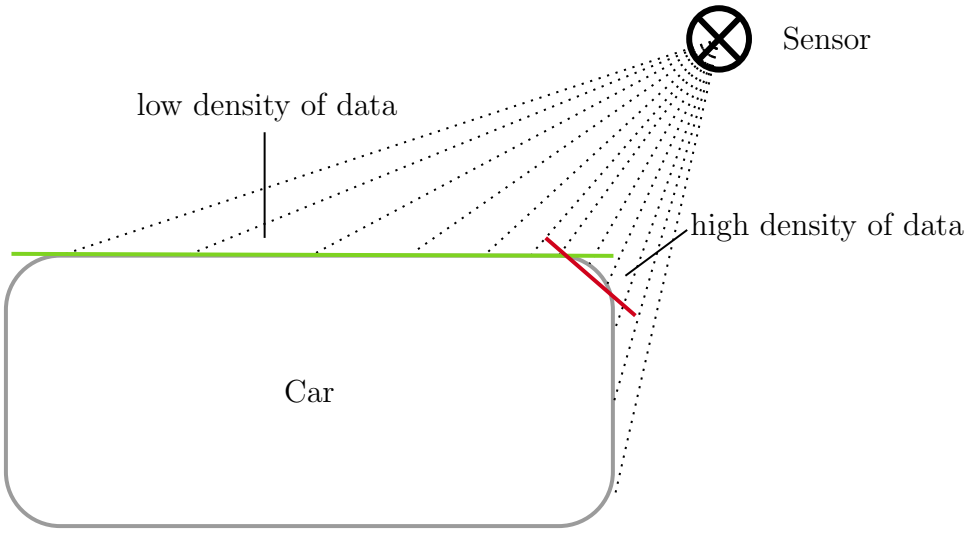



Figure 5.12: Illustration of how the density of data affects the fitted line. We take into consideration the inlier and the length of the line to choose the best. Sometimes the wrong line (red) outscores the correct line (green) if the corner of the car is near to a sensor, because the density of data points at the corner is way higher than on the side. The closer the car to the sensor the more noticeable the effect is. This can result into the red line outscoring the green line because of the sheer amount of inlier it has.

### 5.2.3 Limits of the Localization Modules

Following the limitations of the *Fast* and *Precise Localization* modules.

**Fast Localization** The clustering based *Fast Localization* fails to successfully estimate vehicle positions if cluster merge together. Cluster can merge together as soon as two objects are closer than double the occupancy grid map resolution $r$ times $\sqrt{2}$, see Sec. 4.5.1, which was set to $0.5m$ for our experiment. Through sensor noise the cluster might even merge faster. Lets assume we have a grid resolution $r$ which means each grid cell has a size of $r \times r$. Lets assume we have a point in the bottom right corner of a cell and one point in the top left corner of the adjacent cell on the top left of this cell. Because we are using 8-connectivity, the connected component labelling alg. will assign both cells the same label. The distance between the two points is twice the diagonal of a cell, $2\sqrt{2}r$. On the other side, we can only be sure two points are assigned to the same cluster if the distance is less than $r$ between both.

Furthermore, as described in Sec. 4.5.2, the *Fast Localization* highly depends on the measurement data to sense the car from at least three sides. Figure 5.15 shows another example of a false aligned bounding box.

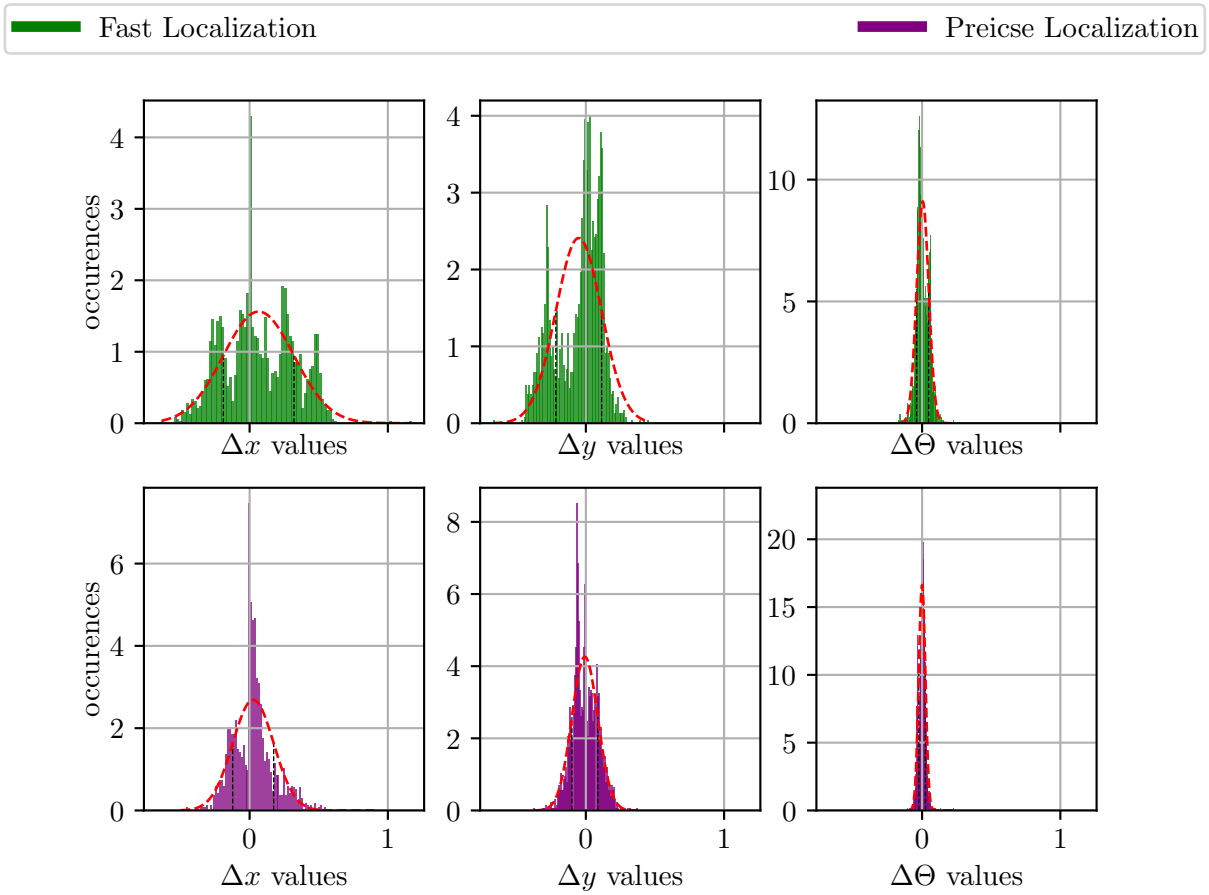



Figure 5.13: Histogram of the difference between the *Fast* and *Precise* localization hypothesises compared to the fused and filtered localization hypothesises from the remote server for the driving scenario

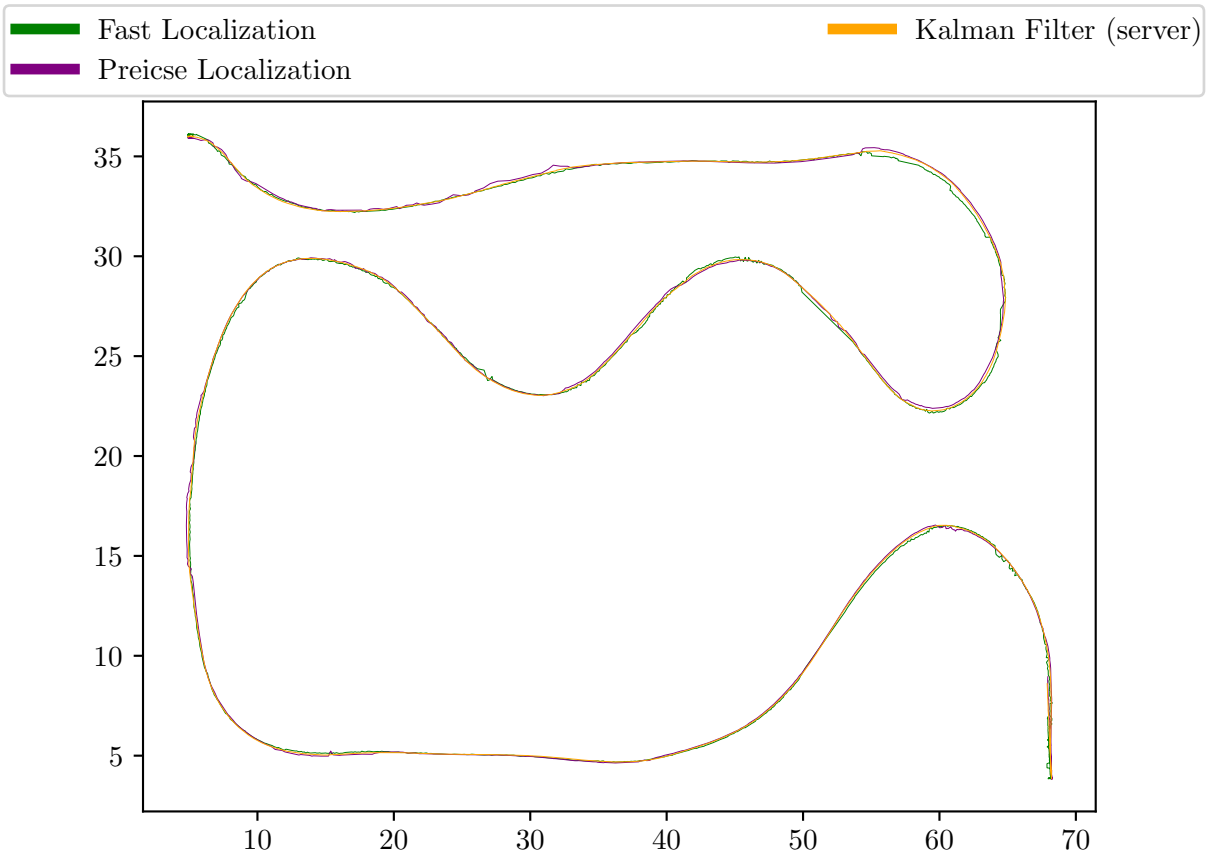



Figure 5.14: Localization results for the driving scenario plotted in 2D (x-y-axis)

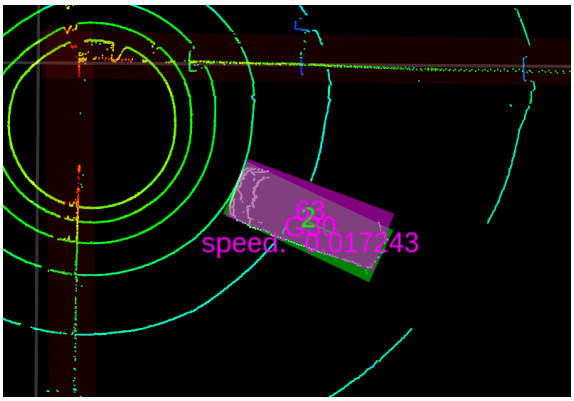



Figure 5.15: Limits of the Fast Localization Module (green box). Because of the measurement data, the box is not aligned with the sides of the car, resulting in a wrong detected position.

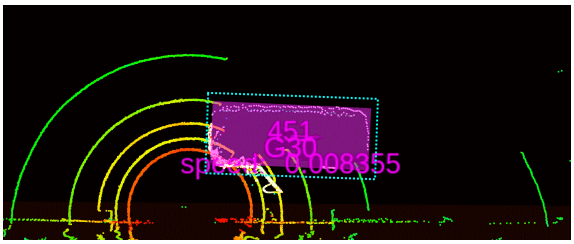



Figure 5.16: Car next to a sensor. The turquoise dotted rectangle illustrates the area to retrieve the points to update the track. Because of this, most of the door is not seen by the algorithm, otherwise the strongest line would lie on the door, resulting in wrong localizations.

**Precise Localization** To achieve a high density of parked cars, the localization module has to be capable of detecting cars which are close together. In our experiment, see Fig. 5.17, we had one car stationary (ID 1415 and 1549) and another car moving around it. In the first test, it approached and passed it on the side and drove a circle around the non moving car (top and mid row of the pictures). In the second test, it parked behind the other car (bottom row). It is important to notice that the driving car model was not known to our system. This leads to the algorithm thinking that the car is longer than it actually is (purple box is longer than what the measurements imply). We experienced effects of the other car on the localization results as soon as the distance between the two cars was less than $50cm$. This was achieved by trusting on the track and assuming we drive slowly while near other obstacle and only retrieving a subset of the points around the track which gets updated, see Sec. 4.6.2. This means only a small set, if any, of the points from the other car are used to update the current car track.
Figure 5.16 shows a experiment where the car was standing next to a sensor with an open door. The turquoise dotted rectangle illustrates the area to retrieve the points to update the track. We have a high difference between the density of the points in the measurement data. If all points would be seen by the updating function, the strongest line would be aligned through the door. Because we only retrieve a small subset of points around the track, most of the door is not seen by the updating function, resulting in the correct localization position. The size of the rectangle depends on the time of the last update as well as the speed of the vehicle. Through this, we are still able to robustly estimate the correct position of the car.

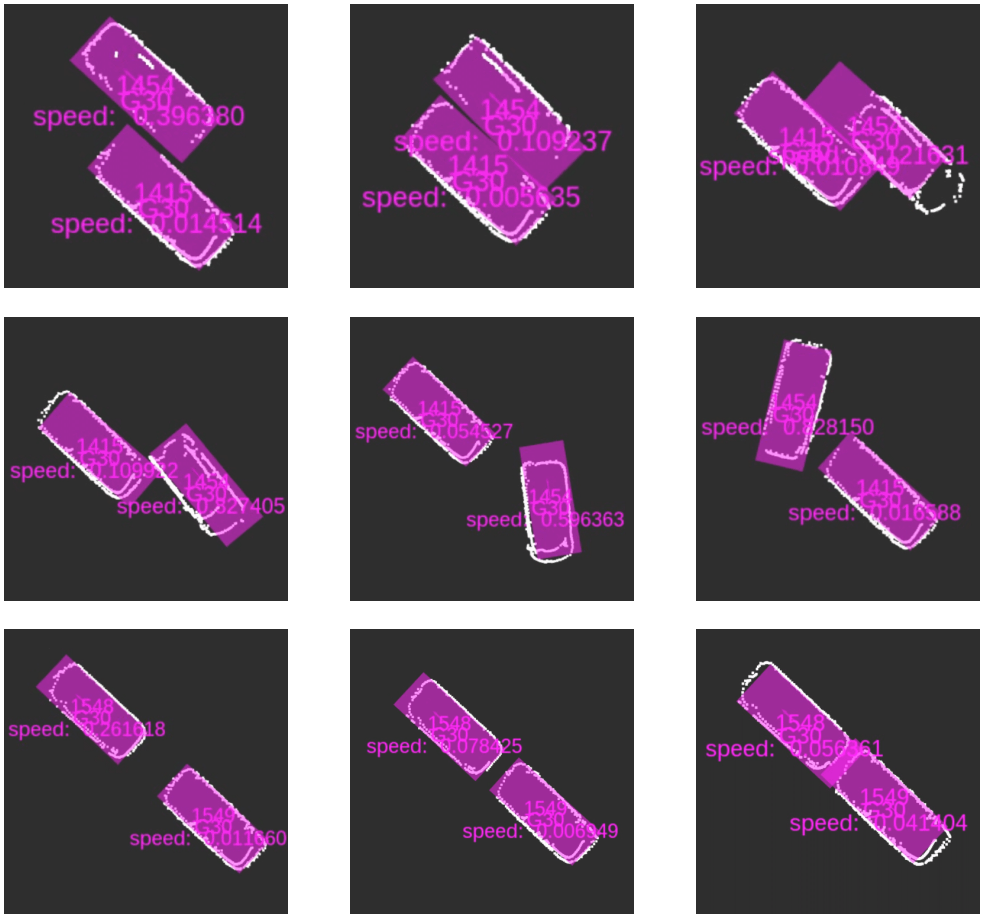



Figure 5.17: Limits of the Precise Localization module. One car was standing still (Id 1415 and 1549) while the other was slowly driven by a human. The localization hypothesis are affected by the other car if the distance between the two cars is less than $50cm$. In the top and mid row the car is passing the other car on the left side and then making a turn. In the bottom row, the car is approaching the other car from behind.

# Chapter 6

# Discussion

The developed system is used to demonstrate the possibility to have vehicles drive autonomously on constraint scenarios and to proof the concept of autonomous valet parking. We achieved a more generic approach by scanning for the autobody and cover an area of $\approx 80m \times 40m$ with five LIDAR scanners.
We experienced a robust system with reproducible results, using it for over more than hundreds of test drives, driving forward and backwards. The other test paths, excluding the one from the driving scenario test, Fig. 5.8, can be found here A.
During testing phase, we marked the driving path by adding cones to both sides, allowing us to test the reproducibility of the system. The car constantly "dodged" the cones with around $50cm$ of free space.

**Computational Time** We achieved a system which runs at a rate of $20Hz$, which is limited by the input rate of the sensors, not utilizing GPU-acceleration.
The latency of the system can be split up into three parts:

- Background substraction: $5ms$
- Merging pointclouds: $20ms$
- Fast Localization: $1ms$
- Precise Localization: $35 - 55ms$

The time of an update cycle for the *Precise Localization* highly depends on how the measurement data is presented. If we inspect the two stationary tests, on test 1, we achieved an update rate of $31Hz$, see Tab. 5.4. On test 2, we achieved a rate of $19Hz$, see Tab. 5.6, which is slightly lower than the input rate. This is coupled by the optimization function, which differs depending on the input. In the second test, the calibration errors are more noticeable, which might affect the optimization function in such a way, that it has more local minima in regard to if no calibration errors are present. Functions with a lot of local minima are harder to optimize for the optimization algorithm.

**Accuracy of the Localization Hypothesis** Because of the calibration errors described in Sec. 5.2.2 it is hard to evaluate the systems accuracy. This also affects the hand labelling of the ground truth data, where it is often not clear where exactly the car is, see Fig. B.4.
The stationary test results include the noise of the sensors but miss two important aspects:

- The sensors are spinning. Therefore, there is a time difference of up to $50ms$ between the sensing for a point in the same pointcloud (for $20Hz$ spinning rate). E.g. the first sensed point has timestamp $t_f$, the last sensed point $t_l = t_f + \Delta t_{spin}$ with $\Delta t_{spin} = \frac{1}{f} = 50ms$. If the objects are not moving, the time difference in sensing them does not matter.

- The calibration errors. Because we do not move the object from its place, the calibration error stays the same for this object all the time. While moving, the calibration error changes depending on the distances to the sensor, making it hard to predict the effects.

We see that we can achieve a standard deviation of $0.01m$ for a vehicle which distance is at least $26.1m$ away from each sensor. In regard, the sensors state a standard deviation of $0.03m$ for a range of $50m$. We also experience a higher deviation for the *Fast Localization*. This might depend on the sensor noise, as it always constructs the rectangle such that every point, also the noisy outlier, is inside of it, leading to a more noise in the estimated positions.
In the stationary test 2, see Sec. 5.2.2, we obtain a huge difference of the estimated position between the two localization modules. This can also be seen in Fig. 5.7(b). The difference and high deviation of the *Fast Localization* $x$ output highly corresponds to the high deviation of the orientation output $\theta$. This comes as depending on how many points are sensed on the right side of the car, the orientation of the box differs, because we simply want to minimize its area. Another example of this effect is shown in Fig 4.6. This also affects the center-point of the rectangle, used to representing the $x$ and $y$ values of the pose. We obtain a higher deviation into the $x$ direction than the $y$ direction from the *Fast* module. This makes sense as the false orientation of the rectangle has more impact on the $x$ value (left to right) than on the $y$ value (bottom to top) in this case. We successfully overcome this issue with the *Precise* module by sensing for a strong line. We obtain a higher deviation into the $y$-axis for the *Precise* results. This can be explained as the vehicle does appear to be shorter in the measurement data than in the real world ($y$-direction, purple box (actual size of the car) appears bigger than the size implied by the points obtained from the sensors).
By comparing the results with the odometry, we get following findings:

- The noise level of the speed is higher in the *Fast* module while it is lower in the *Precise* module if the vehicle is standing still.

- The noise level of the speed is lower in the *Fast* module while it is higher in the *Precise* module if the vehicle is moving.

- The noise level of the orientation is overall lower in the *Precise* module in regard to the *Fast* module.

The first results corresponds to the lower standard deviation in the position for the *Precise* module than for the *Fast* module in the stationary tests. The lower the standard deviation, the less the position changes (for stationary objects), leading to less phantom speed for the object. Phantom speed is referred to as the speed which is calculated by the noise of the position, $|v_{sensed} - v_{real}| = v_{phantom}$.
The second finding might be related to the calibration issues. The *Precise Localization* uses a fixed size for the rectangle to estimate the position in regard to the *Fast Localization* which uses the measurement data to find the size of the rectangle. If the car now appears to be bigger or smaller in the measurement data than for the real world, the *Fast Localization* adapts its size to the measurement. The *Precise Localization* keeps the size fixed, leading to optimize a smaller or bigger rectangle to the data. Depending on which side of the rectangle is aligned to the borderline of the car implied through the sensor data, the estimated position differs. This can lead to sudden jumps in short time periods, which are highly noticeable in the derivatives. This effect can be seen e.g. in Fig. 5.14 on the left side, where the path goes down. The last comparison is just used to give a hint over the absolute accuracy of the system in a driving scenario.
The calibration errors, which introduce unknown errors at the beginning to the system and the usage of hand labelled ground truth data, also containing some errors induces some inaccuracy for the evaluation. Alternatively one could gain more accurate ground truth from differential GPS which was not available for our experiments.

**Limits of the Localization Modules** We experienced good results with running both localization modules in parallel. If one localization module fails, the *Fast Localization* because of clusters merging/breaking up or limited view on the car or the *Precise Localization* because of high differences in the density of the measured data leading to wrongly fitted lines, see Fig. 5.12, the other was producing reliable results. This makes the system robust to failures. The minimal distance between two cars to still retrieve reliable localization positions is $50cm$, which is sufficient to achieve a high parking density. We never tested the system with more than two cars, parking a car between two other cars might affect the sensor readings in a way that this system does not produce reliable position estimations any more.

**Sensors** We experienced different behaviour of our used sensor during the experiment. The quality of the provided measurement data highly corresponds to the surface of the car/object. Metallic car bodies are more reflective than frosted ones

Figure 6.1: Measurement data of a black car. Right image a picture from the car in the real world, left image shows the sensor readings.

making it harder for the sensor to sense the car, resulting in less measurement points in the provided data for metallic cars. Furthermore, black cars in regard to white ones have the same effect, see Fig. 6.1. The Quanergy M8 scanner emits three rays into each direction aiming to minimize false detections through e.g. rain or fog. Because of this, the sensor maybe filters the non consistent measurements for metallic or black surfaces automatically on the sensor level. On the other side, when it was snowing, snow close to the LIDAR was present in the produced data, see Fig. B.2. This is less a problem than the absence of measurements, because it is random over time and removed during the filtering process of our system.
On one side, we want to make the sensor sensible enough, to sense every car independent of the surface, but also want to minimize false detections in the sensor data. With this in mind, the sensors should be tested in outdoor scenarios on different surfaces, on different ranges and weather conditions, to make sure they fit in every situation.

**Future Work** One of the main issues of our system is the lack of automatic calibration for the LIDAR sensors. For such a system, it is necessary to calibrate on all unknown parameters without the assumption that every sensor is perfectly horizontal, to make the calibration error as small as possible. Therefore, developing a semi automatic calibration software is mandatory. It should be able to adjust small changes in the calibration over time, as we can not assume that the sensors never move, especially in outdoor scenarios.
The LIDAR sensors used were not ideal for this task as well. A higher resolution in the vertical field of view would be advisable. Another drawback is the layer based scanning technique, which is especially noticeable if we never move the sensor, in regard to if it is mounted on a moving platform. Having the sensor stationary leads it to always sense the same area, and if e.g. a object happens to not get hit by any layer from the sensor, it will never measure it unless the object moves. This means not all objects in the area have to be present in the sensor data. This can

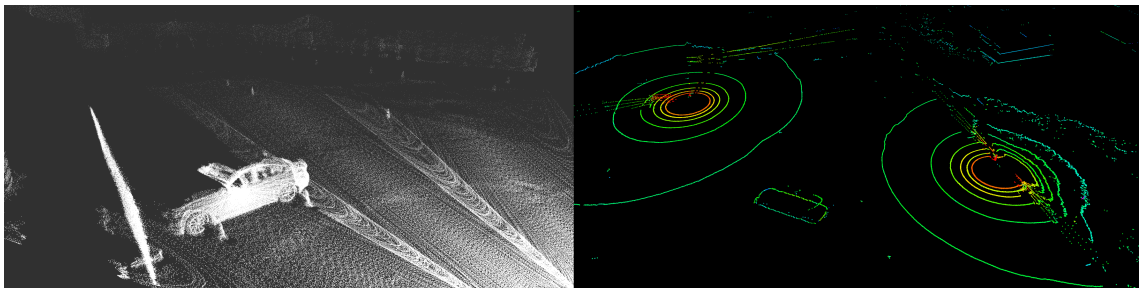


Figure 6.2: Comparision of non-repetetive scanning LIDAR (Livox Mid, left) and layer based scanning LIDAR (Quanergy M8, right) output. It is important to note that we collected the data on the left side over a period of $1sec$, on the right side only one scanning phase which takes $50ms$.

be overcome by the assumption that every object has to move into the area at some point and can not suddenly disappear, therefore still has to be at that location even if it is not sensed at all any more. With the rise of new LIDAR sensors, using non repetitive scanning phases, e.g. the Livox Mid [DJI], scanning the complete FoV over time, see Fig. 6.2, this problem might be solved. It is important to note that the data of the Livox Mid presented in the picture is accumulated over $1s$ in regard to the data from the quanergy M8 sensor, which is one scanning phase, taking $50ms$. Measurement data collected over $100ms$ for a slowly driving car is shown in Fig. B.1. These new sensors opens the ability to control the accumulation process depending on the object speed and on how preciously we have to localize the object. Accumulating the data of a layer based sensor does not yield much information if we are not moving, because it would sense the same points over and over again, only minimizing noise. With such data, it would also make sense to not reduce the dimension of the cloud to 2D, as it holds more information in the z-Axis through the higher resolution in this direction. One approach with these sensors would be for example to match a CAD-model [MKYT08, KPRB09] for the controlled vehicles in the measurement data. This might introduce higher computational costs, processing 3D data in regard to 2D, making it necessary to use GPU-acceleration or having the kalman filter in the remote server compensate for the time difference [SL15]. These sensors would make the calibration task also simpler, as finding corresponding features over different sensors is easier in a dense pointcloud.

Another big topic is occlusion. We set the sensors in a way aiming to hit the vehicles all the time with the sensors. This is most probably not the go to solution, because this way we have a lot of occlusion as well. Mounting the sensors higher and having them scan diagonally down yields the problem with the layer based scanning, that we might not hit objects at all or producing output which is hard to analyse for an algorithm. Here, non repetitive scanning sensors might help as well to come up with a good solution.

In the first iteration, we wanted to use a particle filter for the *Precise Localization*

module. The weight function for each particle was the distance from each point in the pointcloud to the closest point of a recorded down sampled pointcloud of the vehicle transformed to the corresponding position of the particle. The evaluation function was to costly to run fast enough. It could be researched how the particle filter performs in regard to the current system using the optimization function as evaluation function for the particles.
For the *Fast Localization* module, at some point it might be necessary to compute bounding boxes of arbitrary shapes because rectangles can be a poor choice for some objects, resulting into blocking a path even if it is free. Furthermore, the classification should be capable of classifying more classes. A machine learning approach utilizing the set of points assigned to each cluster, especially if we use 3D data, should provide valuable information about the objects.
For scalability of the system, it might be necessary to separate the complete area into smaller sub areas, which process the informations on their own and only exchange data if some objects pass from one area to another. This would result into a overall linear scaling.

# Chapter 7

# Conclusion

In this thesis we present a environment based localization and tracking module. It is responsible to detect every object in the area with the focus on precisely detecting vehicles on an outdoor parking area using LIDAR sensors. The localization system is part of a bigger system which enables autonomous driving on the parking area. The system provides reliable positional data.
In the proposed processing chain, first the measurements of every sensor are divided into fore and background. The foreground gets merged into a common world coordinate system. This data is used by two different localization modules:

- The *Fast Localization*, responsible to detect every object in the area and to be fast in computational time. It is a clustering based algorithm generating minimal area rectangular boundingboxes for each detected cluster to estimate their position. It fails in different scenarios for producing reliable positions for vehicles, e.g. if two objects are assigned to the same cluster.

- The *Precise Localization*, responsible to boost the accuracy of the detected positions for vehicles and to overcome the drawbacks of the *Fast Localization*. This is important because for controlling vehicles closed loop we also have to have a reliable position. It only maintains existing tracks, which are initialized by the *Fast Localization* when it detects a new car. Each track corresponds to a vehicle and they are getting updated independently by using the measurement data around the updating track position. It does a Hough Line transform to fit a line in the retrieved data. This line is assumed to be one side of the car. The line sets the orientation of a rectangle with the size of the current track car model, which gets optimized on its position (x,y) to minimize a cost function. The cost function aims to minimize the distance of every point in the measurement data to the borderline of the rectangle. It uses the global derivative free optimization algorithm CRS2 with local mutation.

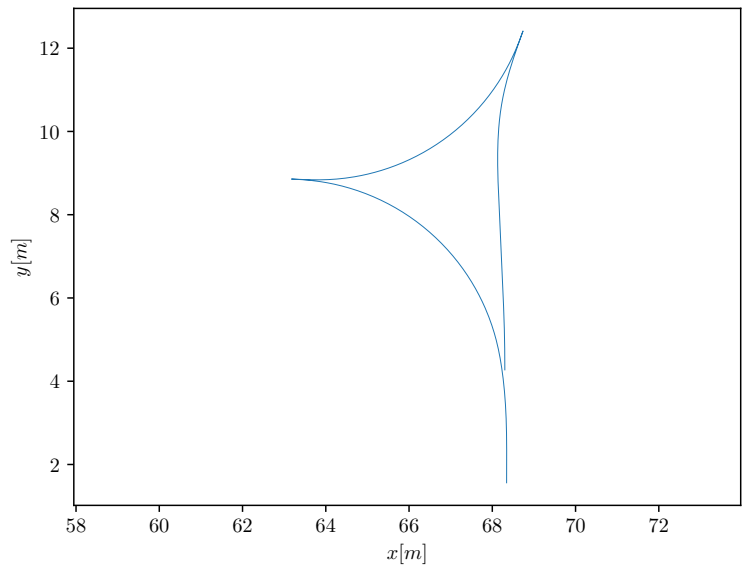
In our experiments we showed that our system is suitable to provide data which can be used to remotely control multiple vehicles on a predefined outdoor area. We

used five LIDAR sensors on an area of around $80m * 40m$. The reproducibility of the position where shown by autonomously driving a fixed path multiple times and marking the path of the car with cones. Furthermore, a fail-safe scenario where a person is in the path of the vehicle, making it stop, was successfully tested as well. The *Fast Localization* module could technically achieve a processing rate of up to $166Hz$ and was running at the sensor rate, $20Hz$. The *Precise Localization* is capable of correctly detecting vehicles in more complex scenarios, e.g. a car with open doors or two cars close together.
An advantage of our system is that the autonomously driving car does not need any specific sensors, making every car with a internet connection suitable for the system. Following a summary of the proposed future work:

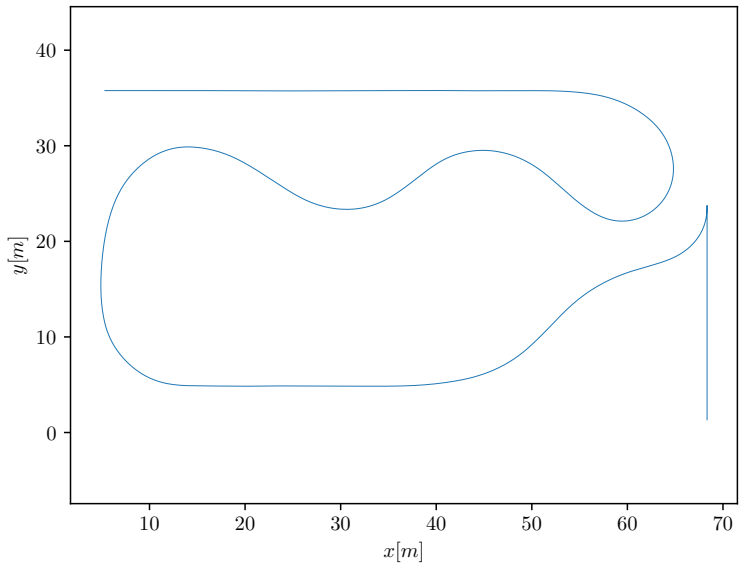
- (Semi)-automatic calibration software for high distances between the LIDAR sensors
- Adapt the system to non-repetitive scanning LIDARs
- Research on how to minimize the occlusion and how the sensor setup effects the localization system
- Particle Filter for *Precise Localization* utilizing the developed cost function
- Extend the bounding box generation to generate arbitrary shapes for the objects
- Introduce a machine learning based classification algorithm
- Research on the scalability

# Appendix A

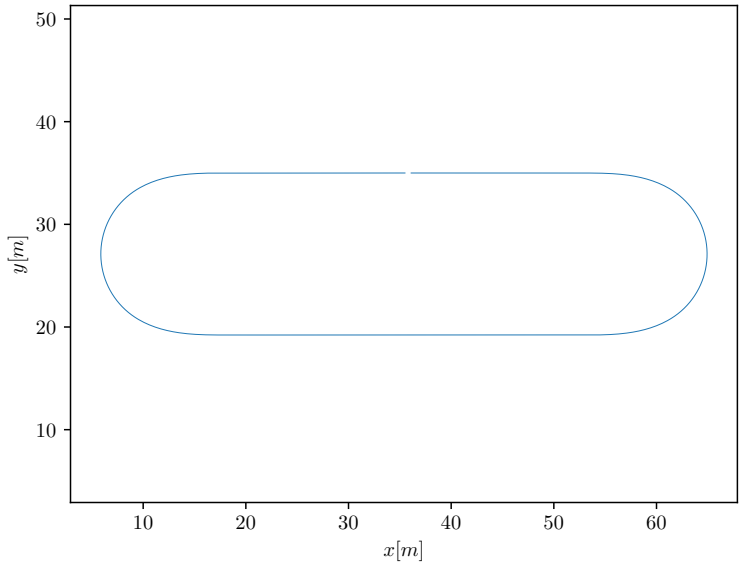
# Test paths




Path for turning on the spot. We start at the bottom and drive backwards, then turn around and park at the same spot backwards.




We start at the bottom right with the front of the car pointing upwards. We then drive forward till we reach the turning point. From there, we drive the driving scenario from Fig. 5.8 backwards.




We start somewhere on the path and we will circle around it forever. The direction is counter clockwise. This was used to test the fail safe [Sch19].

# Appendix B

# Figures

Figure B.1: Non repetitive scanning LIDAR sensor (Livox Mid). The data is accumulated over $100ms$.

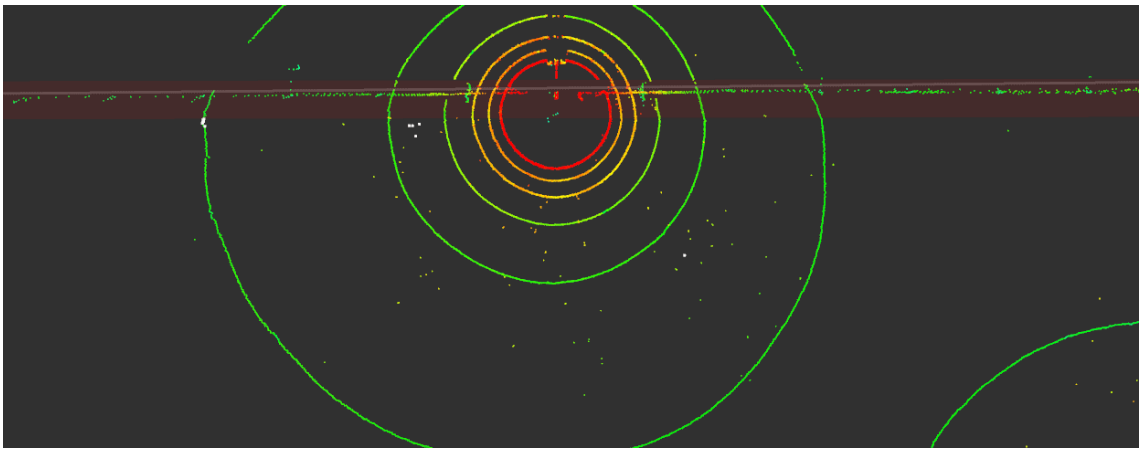


Figure B.2: Sensor readings during snowing. The dots around the sensors origin in the top middle of the picture are caused by the snow.


Figure B.3: Resulting localization hypothesis of the Precise Localization if a wrong line is choosen

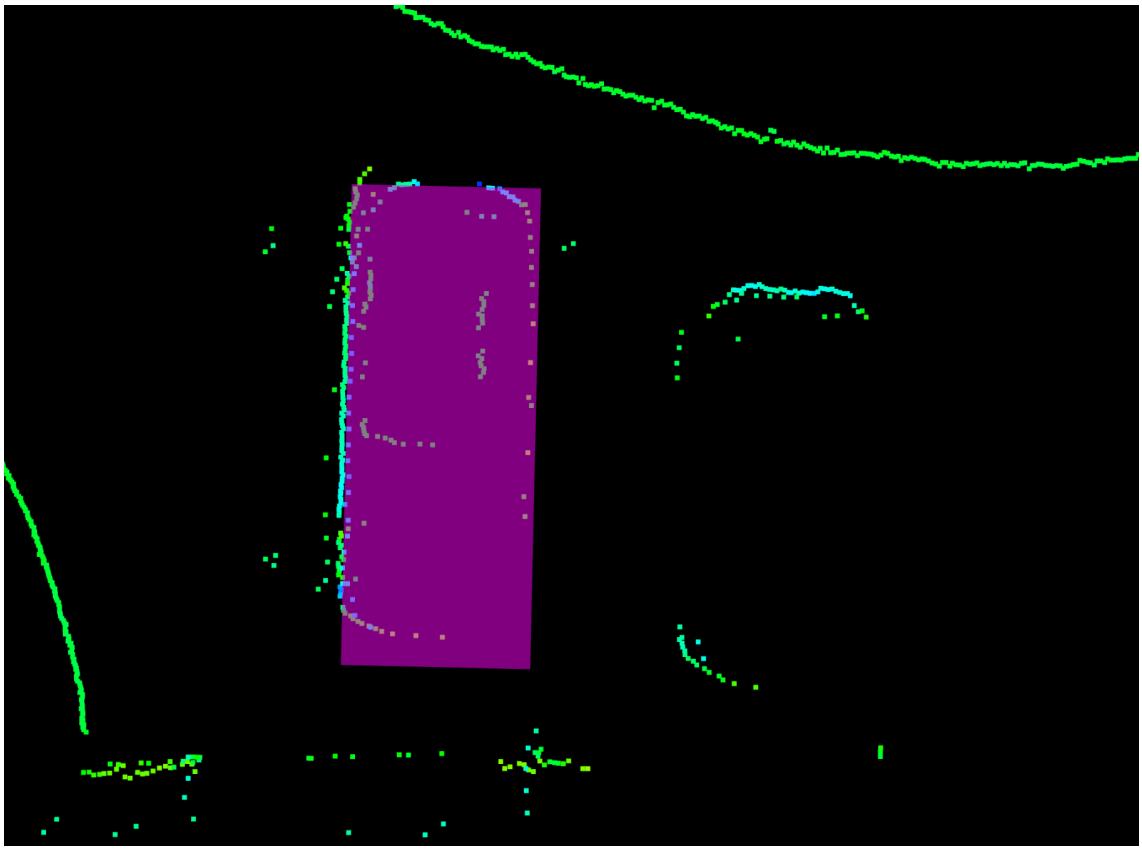


Figure B.4: Measurements of a car in the bottom right corner of the area. Because of the calibration errors, the car position is not clearly.

# Appendix C

# Research: RANSAC Model Fitting for Precise Localization

First, we wanted to do a RANSAC modelfitting of a rectangle to find the localization position. We chose three random points, assuming the first two correspond to one line and the other one to one of the orthogonal lines. With these, we calculate the rectangle and count the inlier by taking all points close the borderline of the rectangle. The assumption to find these three points was very unlikely, making us discard this approach. The results in different scenarios can be obtained in Fig. C.1

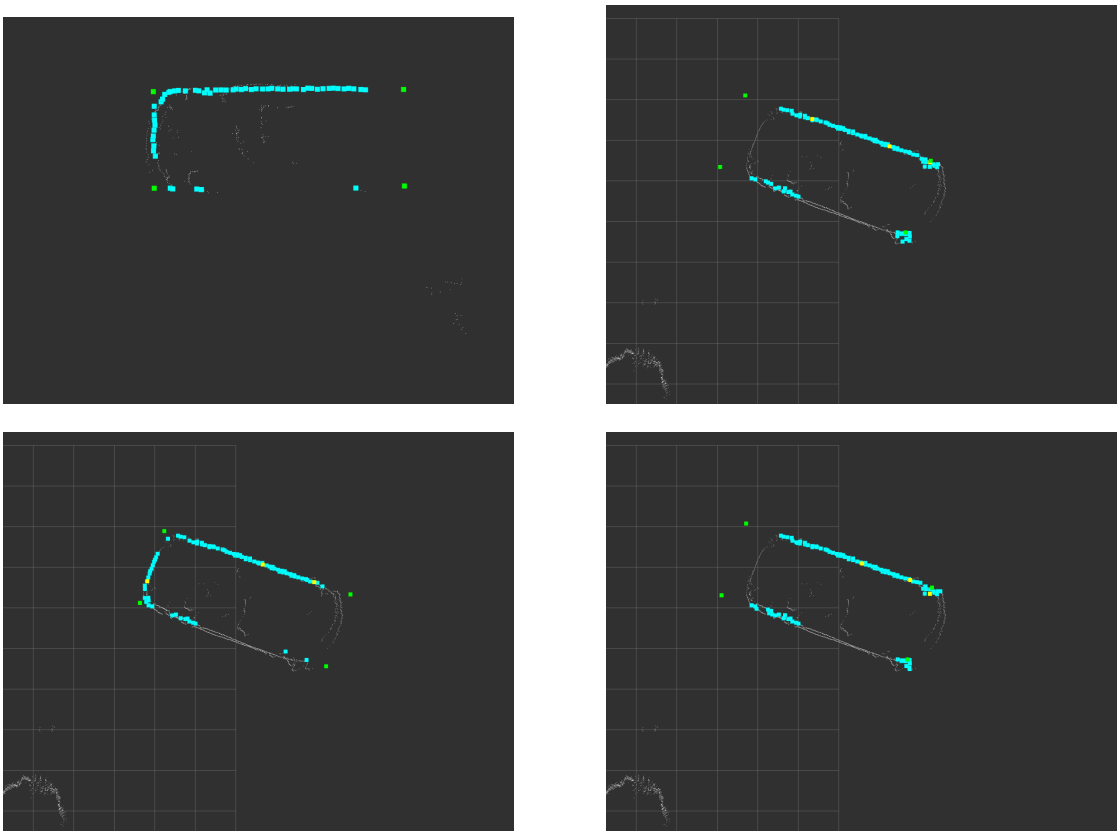


Figure C.1: RANSAC rectangle fit. The yellow dots are the chosen points used to construct the resulting rectangle, defined by the four green dots as the corners. The turquoise points are the inlier.

# List of Figures

# Acronyms and Notations

**AVP** Autonomous Valet Parking

**DLT** Direct Linear Transformation

**LTE** Long Term Evolution

**LIDAR** Light Detection and Ranging

**FOV** Field of View

**ROS** Robot Operating System

**PCL** Point Cloud Library

# Bibliography

[AA71] Y. I. Abdel-Aziz. Direct linear transformation from comparator coordinates in close-range photogrammetry. 1971.

[APN15] A. Asvadi, P. Peixoto, and U. Nunes. Detection and tracking of moving objects using 2.5d motion grids. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 788–793, Sep. 2015. doi:10.1109/ITSC.2015.133.

[BKKZ04] Christian Böhm, Karin Kailing, Peer Kröger, and Arthur Zimek. Computing clusters of correlation connected objects. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 455–466, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1007568.1007620, doi:10.1145/1007568.1007620.

[BNKZ17] Holger Banzhaf, Dennis Nienhüser, Steffen Knoop, and J Marius Zöllner. The future of parking: A survey on automated valet parking with an outlook on high density parking. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1827–1834. IEEE, 2017.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[BW15] Michele Bertoncello and Dominik Wee. Ten ways autonomous driving could redefine the automotive world. *McKinsey & Company*, 6, 2015.

[DJI] DJI. Non repetitive lidar sensor livox mid. Website. Online available at https://www.livoxtech.com/mid-40-and-mid-100; visited at 12th Mar 2019.

[DKKR09] Darko Dimitrov, Christian Knauer, Klaus Kriegel, and GÃ$\frac{1}{4}$nter Rote. Bounds on the quality of the pca bounding boxes. *Computational Geometry*, 42(8):772 – 789, 2009. Special Issue on the 23rd European Workshop on Computational Geometry. URL: http://www.sciencedirect.com/science/article/pii/S0925772109000200, doi:https://doi.org/10.1016/j.comgeo.2008.02.007.

[DMC90] Ernst D Dickmanns, Birger Mysliwetz, and Thomas Christians. An integrated spatio-temporal approach to automatic visual guidance of autonomous vehicles. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1273–1284, 1990.

[Ebe15] David Eberly. Minimum-area rectangle containing a set of points. 2015.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996. URL: `http://dl.acm.org/citation.cfm?id=3001460.3001507`.

[GBK+12] Klaus Greff, Andre Brandao, Stephan Krauss, Didier Stricker, and Esteban Clua. A comparison between background subtraction algorithms using a consumer depth camera. volume 1, 02 2012.

[GMK99] C Galamhos, Jose Matas, and Josef Kittler. Progressive probabilistic hough transform for line detection. In *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, volume 1, pages 554–560. IEEE, 1999.

[Goo99] Tomassia Goodrich. Cs16 convex hull. Website, 1999. Online available at `http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf`; visited at 6th Feb 2019.

[HKMS17] Kersten Heineke, Philipp Kampshoff, Armen Mkrtchyan, and Emily Shao. Self-driving car technology: When will the robots hit the road. *McKinsey & Company*, 5:2017, 2017.

[ISA+13] André Ibisch, Stefan Stümper, Harald Altinger, Marcel Neuhausen, Marc Tschentscher, Marc Schlipsing, Jan Salinen, and Alois Knoll. Towards autonomous driving in a parking garage: Vehicle localization and tracking using environment-embedded lidar sensors. In *Intelligent Vehicles Symposium (IV), 2013 IEEE*, pages 829–834. IEEE, 2013.

[Joh] Steven G. Johnson. The nlopt nonlinear-optimization package. Website. Online available at `http://ab-initio.mit.edu/nlopt`; visited at 6th Feb 2019.

[KA06] P Kaelo and MM Ali. Some variants of the controlled random search algorithm for global optimization. *Journal of optimization theory and applications*, 130(2):253–264, 2006.

[KPRB09] Ulrich Klank, Dejan Pangercic, Radu Bogdan Rusu, and Michael Beetz. Real-time cad model matching for mobile manipulation and grasping. In *2009 9th IEEE-RAS International Conference on Humanoid Robots*, pages 290–296. IEEE, 2009.

[Kuh55] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[KWB08] K. Klasing, D. Wollherr, and M. Buss. A clustering method for efficient segmentation of 3d laser data. In *2008 IEEE International Conference on Robotics and Automation*, pages 4043–4048, May 2008. doi:10.1109/ROBOT.2008.4543832.

[LPOZ15] A. Leigh, J. Pineau, N. Olmedo, and H. Zhang. Person tracking and following with 2d laser scanners. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 726–733, May 2015. doi:10.1109/ICRA.2015.7139259.

[MDG+05] Christoph Mertz, David Duggins, Jay Gowdy, John Kozar, Robert MacLachlan, Aaron Steinfeld, Arne Suppé, Charles Thorpe, and Chieh-Chih Wang. Collision warning and sensor data processing in urban areas. 2005.

[Mea82] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129 – 147, 1982. URL: http://www.sciencedirect.com/science/article/pii/0146664X82901046, doi:https://doi.org/10.1016/0146-664X(82)90104-6.

[MHH17] Daniel D Morris, Regis Hoffman, and Paul Haley. A view-dependent adaptive matched filter for ladar-based vehicle tracking. *arXiv preprint arXiv:1709.08518*, 2017.

[MKYT08] Kenichi Maruyama, Yoshihiro Kawai, Takashi Yoshimi, and Fumiaki Tomita. 3d object localization based on occluding contour using stl cad model. In *2008 19th International Conference on Pattern Recognition*, pages 1–4. IEEE, 2008.

[MNR+16] Marie-Anne Mittet, Houssem Nouira, Xavier Roynard, Francois Goulette, and J-E Deschaud. Experimental assessment of the quanergy m8 lidar sensor. In *ISPRS 2016 congress*, 2016.

[NSMH06] Luis Navarro-Serment, Christoph Mertz, and Martial Hebert. Predictive mover detection and tracking in cluttered environments. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, 2006.

[Ooi87] Beng C. Ooi. Spatial kd-tree: A data structure for geographic database. In H.-J. Schek and G. Schlageter, editors, *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 247–258, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[Qua] Quanergy. Quanergy m8 lidar sensor datasheet. Website. Online available at `hhttps://autonomoustuff.com/wp-content/uploads/2017/08/M8_Datasheet.pdf`; visited at 12th Mar 2019.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[RY05] Thomas Runarsson and Xin Yao. Search biases in constrained evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 35:233 – 243, 06 2005. `doi:10.1109/TSMCC.2004.841906`.

[Sch19] Fabian Schoenert. Online verification of autonomous driving in parking scenarios using set-based prediction, 2019.

[SG99] C. Stauffer and W. E. L. Grimson. Adaptive background mixture models for real-time tracking. In *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, volume 2, pages 246–252 Vol. 2, June 1999. `doi:10.1109/CVPR.1999.784637`.

[Skl82] Jack Sklansky. Finding the convex hull of a simple polygon. *Pattern Recognition Letters*, 1(2):79–83, 1982.

[SL15] Nico Steinhardt and Stefan Leinen. Datenfusion für die präzise lokalisierung. In *Handbuch Fahrerassistenzsysteme*, pages 481–511. Springer, 2015.

[SPA15] Xiaotong Shen, Scott Pendleton, and MH Ang. Accurate odometry for autonomous vehicles using in-car sensors. In *International Conference on Electronics, Information, and Communication (ICEIC)*, 2015.

[TBF] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. Probabilistic robotics (intelligent robotics and autonomous agents series).

[THKS87] Charles Thorpe, Martial Hebert, Takeo Kanade, and Steven Shafer. Vision and navigation for the carnegie-mellon navlab. *Annual Review of Computer Science*, 2(1):521–556, 1987.

[Tho] Martin Thoma. Kalman filter tikz example. Website. Online available at https://github.com/MartinThoma/LaTeX-examples/tree/master/tikz/kalman-filter; visited at 12th Mar 2019.

[Tou14] Godfried T Toussaint. The rotating calipers: An efficient, multipurpose, computational tool. In *The International Conference on Computing Technology and Information Management (ICCTIM2014)*, pages 215–225. Citeseer, 2014.

[unk17] unknown. Automated valet parking mercedes-benz. Mercedes-Benz, 2017. Online available at https://www.mercedes-benz.com/en/mercedes-benz/innovation/avp-bosch-and-daimler-show-driverless-parking-in-real-life-traffic/?shortener=true&csref=yt-AVP-Daimler-Bosch; visited at 14th March 2019.

[Wei] Eric Weisstein. Point.line distnace–2-dimensional. Website. Online available at http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html; visited at 12th Mar 2019.

# License