Institute of Parallel and Distributed Systems
Machine Learning and Robotics Lab

Bachelorarbeit

Monte Carlo Localization in dynamic environments based on an Automotive Lidar Sensor Cocoon

Felix Weitbrecht

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Marc Toussaint
Supervisor:	Prof. Dr. rer. nat. Marc Toussaint, Dr. Daniel Hennes, Dr. rer. nat. Jens Honer
Commenced:	2017-02-01
Completed:	2017-08-01
CR-Classification:	G.3, I.2.9

Abstract

Autonomous driving and driver assistance systems require accurate information about the vehicle and its surroundings to perform tasks such as robust path planning. An occupancy grid map can provide such information, but it too requires precise information about the vehicle's location. We present an approach to Monte Carlo Localization on an occupancy grid map based on an automotive lidar sensor cocoon providing 360° measurements around the vehicle using five Valeo SCALA sensors. Standard MCL is enhanced through an alternative particle weighting function and separate alpha filters are used to incorporate odometry measurements. Additionally, scan point sampling is introduced into the particle weighting function to select scan points most representative of pose estimation quality. Compared to paths reconstructed from only the vehicle's odometry signals, the mean squared error in heading angle and position is reduced by 93-97% and 86-96%, respectively. Investigated scenarios include urban roads, factory roads, elevated country roads and highways.

Zusammenfassung

Autonomes Fahren und Fahrassistenzsysteme benötigen akkurate Informationen über das Fahrzeug und seine Umgebung um Aufgaben wie zum Beispiel robuste Bahnplanung durchzuführen. Occupancy Grid Maps können solche Informationen liefern, aber auch sie benötigen präzise Informationen über den Standort des Fahrzeugs. Wir präsentieren auf Basis einer Occupancy Grid Map einen Ansatz für Monte Carlo Localization. 360° Messungen der Umgebung werden von einem Automobil-Lidar-Sensor-Cocoon geliefert, der aus fünf Valeo SCALA-Sensoren besteht. Standard-MCL wird durch eine alternative Gewichtungsfunktion für Partikel verbessert und separate Alpha-Filter werden verwendet, um Odometriemesswerte zu integrieren. Zusätzlich wird Scanpunkt-Sampling in der Gewichtungsfunktion für Partikel eingeführt um Scanpunkte auszuwählen, die am repräsentativsten für die Schätzungsqualität einer Pose sind. Die mittlere quadratische Abweichung in Richtungswinkel und Position wird 93-97% und 86-96% verbessert gegenüber Bahnen, die nur aus den Odometrie-Signalen des Fahrzeugs berechnet wurden. Untersuchte Szenarien beinhalten Innerortsstraßen, Fabrikgelände, erhöhte Landstraßen und Autobahnen.

Contents

Glossary

This short glossary is intended to explain abbreviations and terminology used frequently in this work. It is incomplete in the sense that it does not contain all abbreviations and terminology, this list only includes items that occur frequently without an explanation obvious from context.

Host Vehicle the OGM and SLAM is working with

MCL Monte Carlo Localization

MSE Mean squared error

OGM Occupancy Grid Mapping

Pose Position and heading angle of the host; different from *position* which only refers to location

Scan point cloud Set of scan points gathered by a sensor in one time step

SCALA Lidar scanner, often referred to simply as *sensor*

SLAM Simultaneous Localization and Mapping

Trace File containing recording of all data necessary for OGM and SLAM provided by the host during a test drive

1 Introduction

1.1 Motivation

Autonomous driving is rapidly gaining importance. Replacing human drivers with autonomous vehicles promises significant advantages for everyday life, safety and the environment. Eliminating humans as a source of accidents would significantly decrease traffic related injuries and deaths. A step below autonomous driving are driver assistance systems which assist with or fully automate certain parts of driving, such as staying in one's lane or keeping a safe distance to cars traveling in front. Reacting to and interacting with its environment requires a vehicle to know as much as possible about its surroundings. Lidar sensors enable vehicles to see their surroundings through sets of detections reported for certain ranges and angles relative to the sensors. Combining these detections over time allows the vehicle to construct a map of its surroundings in which it can track its movement. To make the map useful the vehicle location on the map needs to be known as precisely as possible. The process of simultaneously constructing a map and localizing oneself in it is called *SLAM*, short for *simultaneous localization and mapping*. It assists in providing fundamental information about the vehicle's environment such as the shape and extent of the road, the location and motion of other vehicles, and of course its own position relative to obstacles. Reliable information like that is necessary for various parts of autonomous driving such as robust path planning. To maximize the information gathered by the vehicle one can place a multitude of sensors around the vehicle so that the sensors, when their measurements are combined, provide a 360° view around the entire vehicle.

Typical SLAM algorithms assume a static environment, making robust localization especially difficult in highly dynamic environments. There, other vehicles going in various directions, pedestrians and other smaller dynamic obstacles make it hard to obtain accurate localization results sufficient for reliable fully autonomous driving. That is because their changing location causes scan point clouds to depict environments that are different from previously established environment information.

1.2 Goals

We aim to provide improved localization estimates in dynamic environments by utilizing a *Monte Carlo filter* working on an *occupancy grid map*. Its performance is amplified by various modifications including multiple ways of sampling scan points used for determining the quality of estimations.

The particle filter we set up is based on an existing occupancy grid map. The filter provides vehicle location updates to the grid map to help it better line up new scan point clouds to the existing map. We improve the performance of the particle filter first by investigating different particle weighting functions and using alpha filters to estimate odometry signals. Then we implement a series of scan point sampling strategies aimed at finding a set of scan points containing the scan points most important for localization. We obtain a particle filter that is able to reliably track the vehicle in a variety of environments.

1.3 Outline

The rest of this work is structured as follows.

Chapter 2 – Related Work discusses important related work concerning *SLAM*, automotive lidar and occupancy grid mapping.

Chapter 3 – Background presents background theory concerning *Bayes filters*, *particle filters*, *occupancy grid mapping* and the hardware required for this work.

Chapter 4 – Problem Description and Setup explains the system we are working with and its limitations.

Chapter 5 – Approach describes the techniques we use to solve the problem. An alternative particle weighting function is introduced and a variety of scan point sampling strategies used in the particle filter's weighting function are presented.

Chapter 6 – Evaluation and Results formulates a method of evaluation based on matching scan point clouds to satellite imagery and uses it to show and compare the results of our work. Paths estimated by various configurations of our particle filter are compared against paths calculated based on only odometry data.

Chapter 7 – Conclusion concludes the work, concisely summarizing our approach, highlighting key achievements and outlining some areas for future work to expand on.

2 Related Work

There exist many different approaches to SLAM. Extended Kalman Filters have been popular from the beginning [SC86]. They maintain a Gaussian distribution which is updated recursively as new measurements become available. A detailed summary of the history of and many approaches to SLAM is given in [BD06; DB06]. Another popular approach is Monte Carlo Localization [DFBT99], proposed in 1999 to localize and track mobile robots in known environments. In MCL, a particle filter maintains a set of samples drawn from the state space to represent the belief distribution. A key advantage of particle filters over Kalman Filters is their ability to represent arbitrary belief distributions. Combining MCL with a mapping algorithm yields a solution to SLAM.

SLAM has also been studied in automotive scenarios. [LT10] uses a combination of GPS, IMU and lidar to build a non-volatile map capable of being updated on secondary visits. [RJMZ16] simultaneously perform MCL on multiple map layers built from data provided by a 3D lidar sensor.

Other approaches to automotive SLAM exist that don't rely on particle filters. [VAA07] uses a local grid map to keep track of the vehicle's environment; scan matching based on samples drawn according to the motion model is used for position updates.

It's important to distinguish between 2D lidar sensors and 3D lidar sensors. The former are currently the only type of hardware viable for automotive applications, however they often have parts of their field of view obstructed by dynamic objects; the latter, more expensive, 3D lidar sensors, mounted above the vehicle, allow one to specifically select scan points belonging to the ground plane, resulting in measurements free of most dynamic and semi-dynamic obstacles [CQB+13; LMT07].

Other approaches use cameras for localization [Wu16], for example [CMC+17] uses a camera pointed at the road to improve lateral position estimations using road markings.

3 Background

This chapter explains the basics of the standard techniques and methods, and presents the hardware built into the test vehicles used for this work. Specifically, SLAM (Simultaneous Localization and Mapping), Bayesian Filtering, (Monte Carlo) Particle Filters, Occupancy Grid Mapping, and SCALA are covered.

Note that in this chapter the term *robot* will be used to describe the machine on which sensors are installed and algorithms are executed. This term is not meant to be limited to actual robots, instead it is used in the context of techniques developed with them in mind, but also applicable to our purposes, automobiles.

Large parts of sections 3.1 through 3.3 are based on [TBF05].

3.1 SLAM: Simultaneous Localization and Mapping

SLAM is the practice of concurrently constructing a consistent map of a robot's surroundings and continuously determining the robot's location in this map as it is generated.

There are various different scenarios in which SLAM finds application: There can either be one robot or there can be multiple robots cooperating to construct the map, navigating through the same environment at the same time. The map may be partially or fully constructed beforehand (with some or no errors), with the robot first having to localize itself inside this map before being able to contribute to it in a meaningful way. The accuracy and detail provided by the sensors used to describe the robot's movement and its vision of the environment may vary greatly. The map may be limited to a certain area, either fixed in the environment or relative to the robot. With larger or unlimited maps, one may encounter the issue of loop closure, the robot visiting a location it has visited before. There, ideally, the robot's localization in the map should be the same for both visits so the scans of the environment line up between the two visits.

In general, there are two approaches to SLAM:

- Smoothing: Given all measurements up to and including the current time step, what is the most likely sequence of robot states leading up to and including the current time step?
- Filtering: Given the previous robot state and map as well as a new set of measurements, what is the most likely state of the robot for the current time step?

For our purposes, there exists only one robot, building a square map centered on itself. Depending on the configuration of the map, the point farthest from the robot may be up to about 170 meters away from it. Considering the accuracy of even untreated odometry data and the scale of typical highway and urban environments, loop closure is not an issue we need to address specifically. Global localization isn't an issue we need to deal with either because the map is not known beforehand, making this a pure tracking application.

Typically Bayes Filters are used for SLAM. [DB06]

3.2 Bayesian Filtering

A Bayesian Filter estimates a state vector x_t at time t by updating a belief distribution *bel* over x_t whenever a new set of measurement data becomes available. It functions in two steps: prediction and measurement update. Measurement data consist of control inputs or (or, in our case, measured odometry data) u_t and a set of measurements z_t. Based on the previous time step's belief $bel(x_{t-1})$, the so called *prior belief*, a new belief $bel(x_t)$, the so called *posterior belief*, is then calculated.

When an update is applied, the belief is updated for every single x_t. In the prediction step

$$bel(x_t) \leftarrow \int p(x_t \mid u_t, x_{t-1})\; bel(x_{t-1})\; dx_{t-1} \tag{3.1}$$

the probability to arrive at each state x_t according to only the odometry data is calculated. Note that while the left side of the equation does read $bel(x_t)$, this is not the final $bel(x_t)$ - it will be overwritten again in the measurement update directly afterwards.

In the measurement update

$$bel(x_t) \leftarrow \eta\; p(z_t \mid x_t)\; bel(x_t) \tag{3.2}$$

the belief is multiplied by the probability of the measurement z_t having been observed in the predicted state x_t. η is a normalization factor that appeared after applying Bayes'

Theorem to calculate the posterior. There, it replaces the denominator on the right-hand side:

$$p(x_t \mid z_{1:t}, u_{1:t}) = \frac{p(z_t \mid x_t, z_{1:t-1}, u_{1:t})\, p(x_t \mid z_{1:t-1}, u_{1:t})}{p(z_t \mid z_{1:t-1}, u_{1:t})} \tag{3.3}$$

The $a_{b:c}$-notation indicates the sequence of a_b through a_c. For the full derivation of the filter, the reader is encouraged to look up chapter 2.4.3 of [TBF05].

To use the filter, an initial belief $bel(x_0)$ needs to be specified. Typically, the initial state is either fully known, for example in tracking applications, or entirely unknown, like in many localization applications. The obvious choice then will be to assign all probability to the known value or to choose a uniform distribution, respectively.

3.2.1 Particle Filters

Particle filters approximate the posterior $bel(x_t)$ with a finite set of state samples randomly chosen from $bel(x_t)$. This allows one to represent almost arbitrary distributions. Typically, the number of state samples, also called particles, M should be large enough to represent the belief accurately to some extent. Usually M is chosen between 100 and 1000 [TBF05]. Optionally, this number may be varied over time or depending on other parameters. We denote the set of particles at time step t

$$X_t = \{x_t^{[i]} \mid 1 \leq i \leq M\} \tag{3.4}$$

Then each particle $x_t^{[i]}$ is a hypothesis about the actual state at time step t. As X_t approximates $bel(x_t)$, the probability of a hypothesis x_t to be included in X_t should be proportional to its posterior $bel(x_t)$. Assuming the number of particles to be sufficiently large, this means that parts of the state space densely populated by particles are more likely to contain the true state.

Algorithm 3.1 Particle filter

```
1: function PARTICLE_FILTER($X_{t-1}$, $u_t$, $z_t$)
2:     for all $i \in \{1, ..., M\}$ do
3:         sample $x_t^{[i]} \sim$ p($x_t$ | $u_t$, $x_{t-1}^{[i]}$)
4:         $w_t^{[i]}$ = p($z_t$ | $x_t^{[i]}$)
5:     end for
6:     $X_t = \emptyset$
7:     draw $M$ particles $x_t^{[j]}$ from the $x_t$ with probability $\propto w_t^{[j]}$ and add them to $X_t$
8:     return $X_t$
9: end function
```

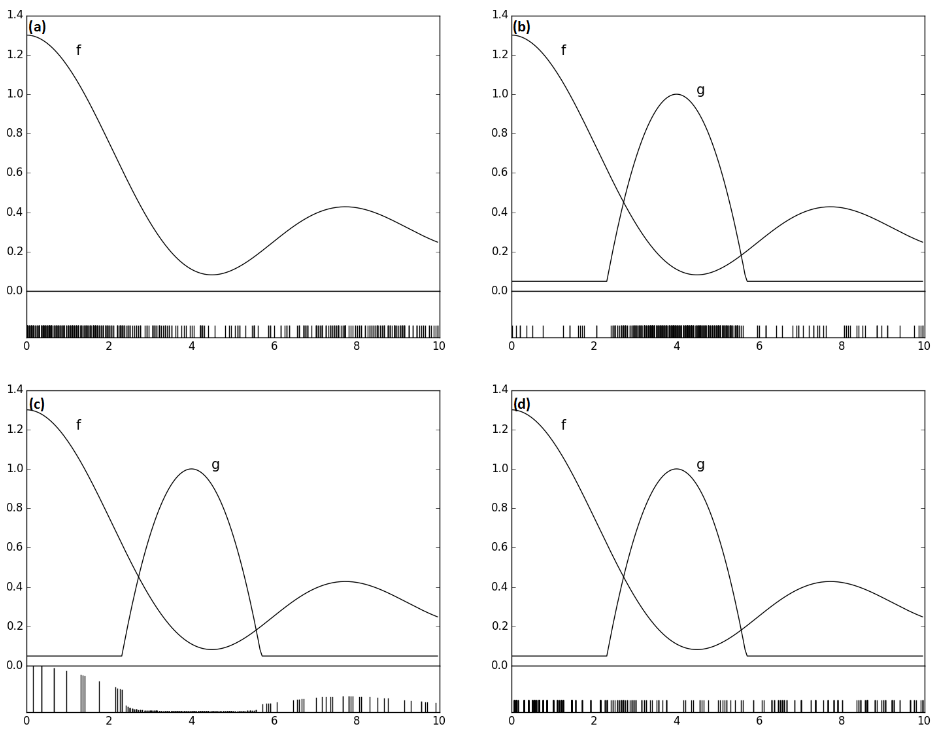

Figure 3.1: The resampling step used in particle filters. Under each graph, the current set of particles is displayed with different heights indicating their weights. **(a)** The density f we aim to approximate. **(b)** Unable to sample from f, we'll have to make do with a different density, g. **(c)** Giving importance factors $f(x)/g(x)$ to the particles, we obtain an approximation of f. **(d)** After resampling from the particle set. Note that the particle set now contains more duplicates, making dense areas appear sparser than they are. f stands for $bel(x_t)$ and g stands for the proposal distribution particle set before the resampling step.

Algorithm 3.1 describes a simple implementation of the particle filter's recursive update step performed whenever a measurement update along with a new set of odometry data becomes available. Just like in equation 3.1, first a temporary set of $x_t^{[i]}$ is constructed based on the odometry data before taking into consideration the measurement update to obtain the new posterior.

The prediction step is implemented by the loop in lines 2 through 5, each particle receives an update based on the odometry data. For this, one state hypothesis is generated for each particle based on its state and the state transition distribution. Also, an *importance factor* $w_t^{[i]}$ is calculated for each newly sampled hypothesis. It is given by the so-called *measurement probability*, the probability of the measurement given the newly sampled

Algorithm 3.2 MCL

```
function MCL($X_{t-1}$, $u_t$, $z_t$, $m$)
    for all $i \in \{1, ..., M\}$ do
        $x_t^{[i]}$ = SAMPLE_MOTION_MODEL($u_t$, $x_{t-1}^{[i]}$)
        $w_t^{[i]}$ = MEASUREMENT_MODEL($z_t$, $x_t^{[i]}$, $m$)
    end for
    $X_t = \emptyset$
    draw $M$ particles $x_t^{[j]}$ from the $x_t$ with probability $\propto w_t^{[j]}$ and add them to $X_t$
    return $X_t$
end function
```

hypothesis. Interpreting these importance factors as non-normalized weights of the particles, the set of particles approximates the posterior $bel(x_t)$. This will be used in the next step which is usually called *resampling*.

Here, in lines 6 and 7, M particles are chosen from the set of just-sampled hypotheses proportionally to their importance factors determined in the previous step. This yields X_t, our posterior approximation. Note that X_t generally will contain duplicates, taking the places of particles that weren't drawn in line 7 because they have evolved into less likely hypotheses. This step is important to avoid spending a lot of computational effort on particles located in areas of low likelihood within the state space as well as to keep a significant number of particles in the area(s) of higher likelihood. These duplicates will not remain identical forever, in the next time step they will diverge from each other as they each have the prediction step applied to them.

Figure 3.1 visualizes how the resampling step allows us to approximate one distribution by drawing samples from a different distribution and appropriately weighting them.

Again, for the mathematical derivation of the particle filter, the reader is encouraged to look up chapter 4.3.3 of [TBF05].

3.2.2 Monte Carlo Localization

MCL, short for *Monte Carlo Localization* [DFBT99], uses a particle filter to perform localization. In this section, we will take a short look at a basic MCL algorithm and illustrate its functionality with a simple one-dimensional example.

Algorithm 3.2 describes how the MCL algorithm works. Note that m is the map and that M is the amount of particles. The main difference between the general particle filter (algorithm 3.1) and MCL is the substitution of the distributions in the loop with a motion model and a measurement model. These provide a way to sample from the motion model and a way to ascertain how well a given measurement matches a given map. Chapter 5 will go into more detail on them.

There are several common straightforward improvements for MCL that should be mentioned even though they aren't used in this work.

- Random particles: Particles tend to bunch up into one location once they become lost. It's highly unlikely for the filter to recover from such a situation on its own. To work against this possibility, a certain number of particles, sampled uniformly from the entire state space, is added into the set of particles at each time step. This also helps recover from getting *kidnapped*, having the true pose changed without odometry data indicating so.
- Resource-adaptive M: To best make use of available resources, one can, instead of setting a fixed M in line 7 of algorithm 3.2, sample more and more particles until u_{t-1} and z_{t-1} become available.
- KLD-Sampling: Using the Kullback-Leibler distance, the approximation error of the particle filter is measured and used to dynamically set M to be smaller for very focused distributions and higher for uncertain ones. [Fox02]

Figure 3.2 shows how the Monte Carlo Filter works by the example of a robot, represented by the black sphere, which is free to move left and right in a one-dimensional, previously known environment. It possesses a sensor allowing it to distinguish between solid walls and air (open doors, for example). Initially (3.2(a)), the robot doesn't know where it is. Its belief is distributed equally over the whole state space, as seen in the set of particles displayed below the robot's environment. In 3.2(b), the robot's sensor has reported air. The belief $bel(x_t)$ is assigned importance factors for every particle according to the measurement probability $p(z|x)$, which, for air readings, exhibits peaks by the three holes the robot would sense air at.

In 3.2(c), the robot moves a few meters to the right. This step shows the particles after they've been resampled and moved in accordance with the odometry data. The belief displays three peaks corresponding to the three locations in the world the robot believes itself to be located at most likely; these three locations all looked the same at the time the only sensor reading so far was processed. In 3.2(d), weights are again assigned to the particles, this time assigning much more likelihood around the correct pose of the robot because the sequence of sensor readings and odometry data so far best matches that pose. In the following iterations, the particle set will most likely converge towards the correct pose.

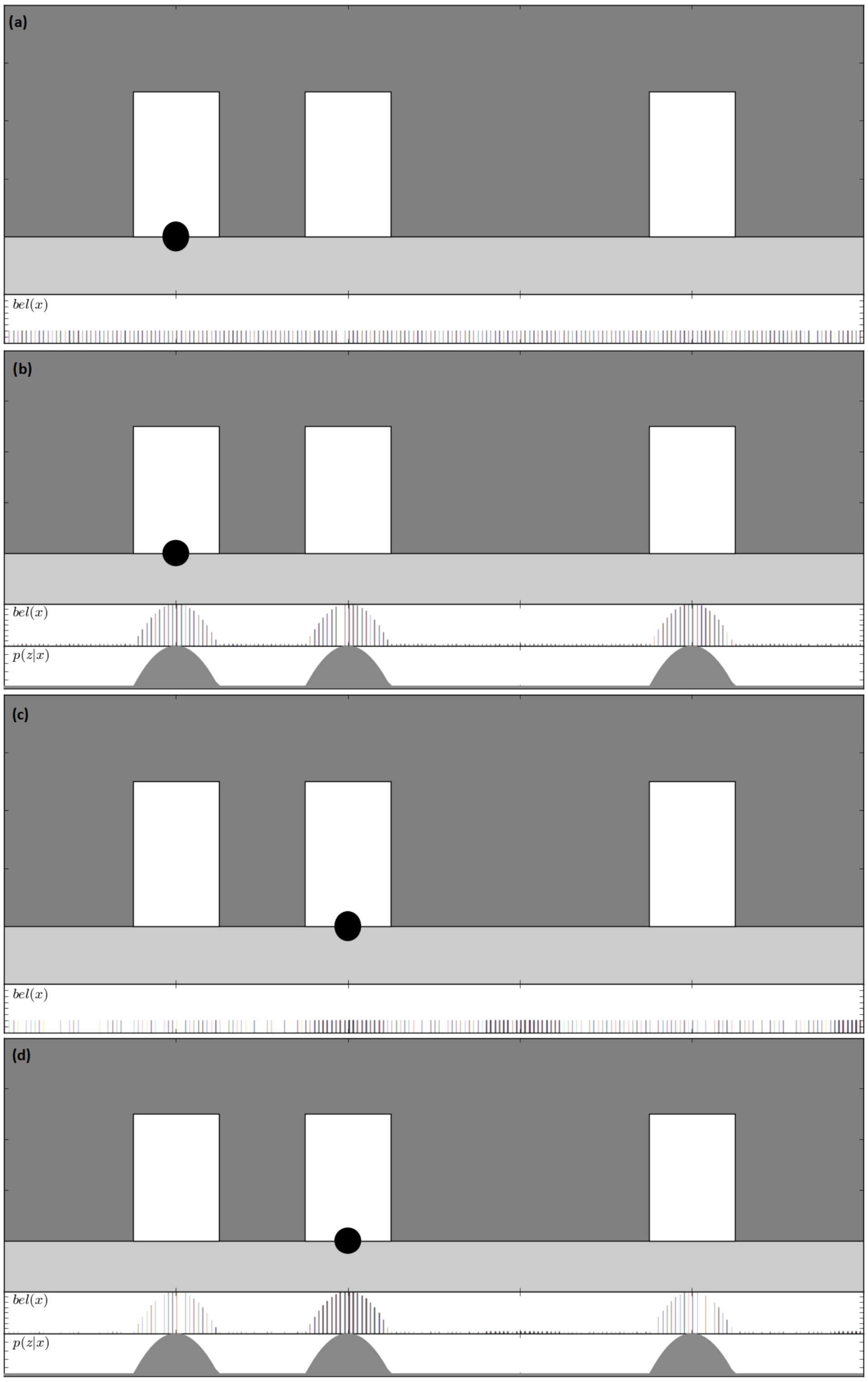

Figure 3.2: One-dimensional MCL, figure inspired by [TBF05]

Algorithm 3.3 Alpha filter

```
1: function ALPHA_FILTER($X_{t-1}$, $X_{m,t}$)
2:     $r \leftarrow X_{m,t} - X_{t-1}$
3:     $X_t \leftarrow X_{t-1} + \alpha r$
4:     return $X_t$
5: end function
```

3.2.3 Alpha Filters

An Alpha filter is a very simple implementation of a Bayes filter. It provides a basic approximation of a single variable X supported by measurements X_m of this variable. Measurements are assumed to be noisy but unbiased. Estimates are updated by shifting the estimate towards the newest measurement by a constant factor α, also called the *gain*. This is in contrast to *Kalman filters* which calculate updates using dynamic factors [Kal+60].

Algorithm 3.3 describes the update step. Line 2 calculates the residual, the difference between the last time step's prediction and the new measurement. Line 3 updates the estimate by adding α times the residual onto the previous one. Rewriting line 3 as

$$X_t \leftarrow (1 - \alpha)X_{t-1} + \alpha X_{m,t} \tag{3.5}$$

demonstrates that it is simply linearly interpolating between the measurement and the old estimate using weights α and $(1 - \alpha)$, respectively. Thus α determines how quickly the estimate reacts to changes in the measured signal.

This filter is closely related to the *alpha beta filter* which maintains two state variables of which one is the other's integral over time. In the context of distance and velocity, the filter would receive distance measurements and, using the residual of the distance prediction, update distance and velocity estimates using gains α and β, respectively [JZ98].

3.3 Occupancy Grid Mapping

OGM, short for Occupancy Grid Mapping, is a common method of storing information about the environment a robot sees. It discretizes robot surroundings into a set of grid cells m_i of a certain size. Usually, these cells are all equally-sized and square. While three-dimensional grid maps exist and have been utilized for vehicle navigation/tracking purposes [HR16], two-dimensional grid maps are most prevalent for ground navigation. Those only operate on a part of the environment, typically a horizontal thin slice-like structure, which, if necessary, is projected down into two dimensions to provide a top-down view of the area.

The goal is to calculate a posterior over the map m given the measurements and the path taken by the robot:

$$p(m \mid z_{1:t}, x_{1:t}) \tag{3.6}$$

As the number of grid cells and thus the number of dimensions is unpractically large for any non-trivial grid map, this posterior is impossible to calculate in practice. Instead, it has become standard practice to look at each cell separately to obtain a posterior:

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(m_i \mid z_{1:t}, x_{1:t}) \tag{3.7}$$

Note that cells are not correlated in this view, an assumption that does not fully match the real world. It also reduces the problem to a set of binary estimation problems, each with static state. Cells which are likely to be occupied will have values closer to 1, cells which are likely to be unoccupied will have values closer to 0 and cells we have little information about or we are unsure about will have values close to 0.5.

Another optimization that will prove convenient is to make use of *log odds*.

3.3.1 Log Odds

When working with binary states, *log odds* are very useful to store the belief. Denoting the two possible states occ and $\neg occ$, the simple relation between their probabilities can be described in the following way:

$$p(occ) = 1 - p(\neg occ) \tag{3.8}$$

Thus we can keep track of only one of the two probabilities without losing any information. Even better, we can use the *log odds ratio* of the two probabilities:

$$log \frac{p(occ)}{1 - p(occ)} \tag{3.9}$$

In the context of commonly used limited precision floating point numbers, this provides two main advantages. Most importantly, precision errors near 0 and 1 are avoided. Secondly, the entire domain of the number format is used because, as $p(occ)$ approaches 0 or 1, the log odds ratio given in equation 3.9 approaches $-\infty$ or ∞, respectively. It also turns what previously were multiplications into additions.

Then, to update the map after a measurement has arrived, all cells covered by the field of view of the sensor the update originated from are updated. Typically, an *Inverse Sensor Model* is used to update the probabilities.

$$l_{t,i} = l_{t-1,i} + ism(m_i, x_t, z_t) - l_0 \tag{3.10}$$

In this equation, $l_{t-1,i}$ and $l_{t,i}$ refer to the log odds representations of the prior and posterior belief of cell i, the function ism is the *Inverse Sensor Model* which will be explained in the following paragraphs, and l_0 is the log odds representation of the prior of occupancy. In applications in which the map is entirely unknown in the beginning, l_0, the log odds ratio of a probability of 0.5, will be 0 and can therefore be omitted.

The occupancy probability can be calculated from the log odds l using

$$p(occ) = \frac{e^l}{1+e^l} \tag{3.11}$$

which is because

$$\frac{e^l}{1+e^l} = \frac{\frac{p(occ)}{1-p(occ)}}{1+\frac{p(occ)}{1-p(occ)}} = \frac{\frac{p(occ)}{1-p(occ)}}{\frac{(1-p(occ))+p(occ)}{1-p(occ)}} = \frac{\frac{p(occ)}{1-p(occ)}}{\frac{1}{1-p(occ)}} = \frac{p(occ)(1-p(occ))}{1-p(occ)} = p(occ) \tag{3.12}$$

3.3.2 Inverse Sensor Model

In the *inverse sensor model*, cells are represented in a probabilistic manner under the assumption of independence of each other. As discussed in the context of occupancy grid mapping (section 3.3), this allows for significantly easier calculation of posteriors.

The inverse sensor model gives the probability of a cell being occupied based on the measurement update. It assigns higher probabilities of occupancy to cells close to detections, lower probabilities to cells between detections and the sensor, and the prior for occupancy to cells behind the measurement and cells out of measurement range. One algorithm implementing this functionality is given in [TBF05].

In general, these functions can and should differ depending on the type of sensors used. For example, one may take into account the distance from the sensor to a cell to determine how likely it is to be unoccupied. Other details to adjust include for example the expected thickness of detected objects and the accuracy of the sensor based on distance to a detection. Another option is to learn a function from sensor data instead of manually calculating or estimating parameters. [TBB+96]

Figure 3.3: SCALA lidar sensor housing. Image: Valeo.

An alternative is the forward sensor model:

$$p(z_{1:t} \mid m, x_{1:t}) \tag{3.13}$$

which is a maximization problem. Here, no assumption of independence between cells is made, allowing the model to handle contradictory detections. However, this comes at the price of considerably higher computational complexity. It is also not possible to recursively run this model as new measurements become available.

[Thr03] goes into more detail on the forward sensor model.

3.4 SCALA

SCALA [GG14], a lidar sensor suitable for mass production produced by *Valeo*, is the type of sensor installed on our test vehicles. Its size is 10.5 by 6 by 10 cm. It has a horizontal field of view of 145° with a resolution of 0.25°. Vertically, it generates measurements on four different layers, each covering 0.8° vertically. Range readings with a resolution of 10 cm or better are provided up to 150 meters, even farther for exceptional objects. Sensor readings are provided at a frequency of 25Hz, alternating between the bottom

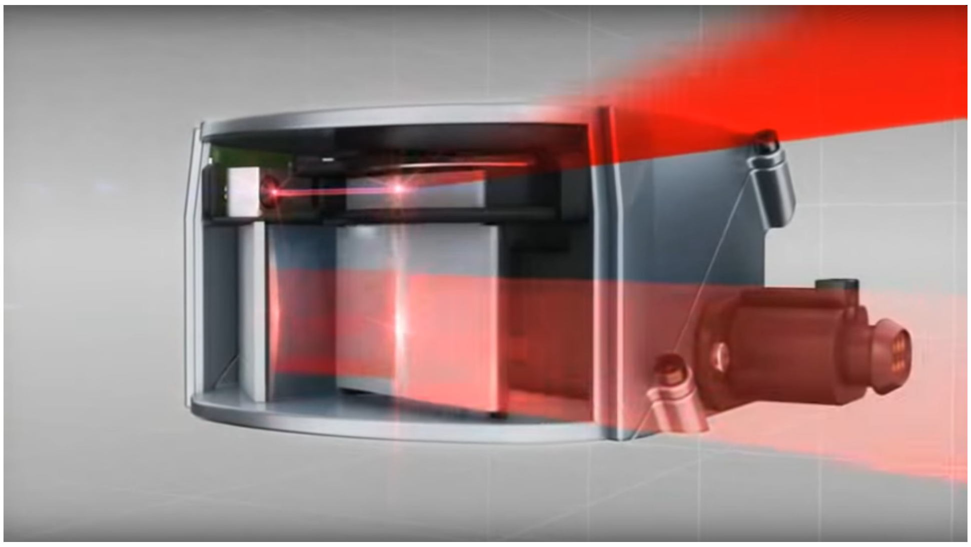

Figure 3.4: SCALA lidar sensor containing two back-to-back rotating mirrors. Image: Valeo.

and top three out of four layers. Figure 3.4 shows one of the two rotating mirrors used to direct the laser beams, which are emitted from a fixed source mounted in the top left in the image.

The set of scan points obtained in a scan spanning three layers typically consists of 1500 individual scan points across the whole field of view.

3.5 DBSCAN

DBSCAN [EKS+96], short for *density-based spatial clustering of applications with noise*, is a simple to use clustering algorithm able to detect arbitrarily shaped clusters without having to provide the number of such clusters in advance. This section serves to outline its basics. DBSCAN classifies data points into three categories:

- *Core points*: points with at least $minPts$ points within a radius of ϵ
- *Reachable points*: points within the ϵ-neighborhood of a core point
- *Outliers*: remaining points

Then, a cluster is defined as a core point and all points, possibly including other core points, reachable from it.

It requires two parameters: ϵ, the maximum distance between points for them to be considered connected, and $minPts$, the amount of points necessary within an ϵ-neighborhood of a point for it to be considered a core point.

The underlying algorithm is given and explained in more detail in [EKS+96]. For this work we will be using the DBSCAN implementation of *scikit-learn* [PVG+11].

4 Problem Description and Setup

This chapter will explain the physical setup we are working with as well as its digital interfaces and outline the objective of this work. Because the grid map we are working with has been developed independently of this thesis, with only minor adjustments made to it for our purposes, it will also be described in this chapter.

4.1 SCALA Cocoon

Like the protective cover some insects develop during part of their life, the term *cocoon* here also refers to surrounding something. Here, a vehicle is equipped with multiple sensors in such a way that the area covered by their fields of view spans around the whole vehicle; only some small coverage holes exist right by the car. Specifically, we are using five SCALAs mounted in the back, on the sides and on the front of the vehicle. Thanks to the sensor's horizontal field of view of 145°, five sensors together can be optimized for full 360° coverage with only five minimal blind spots. The sensors are

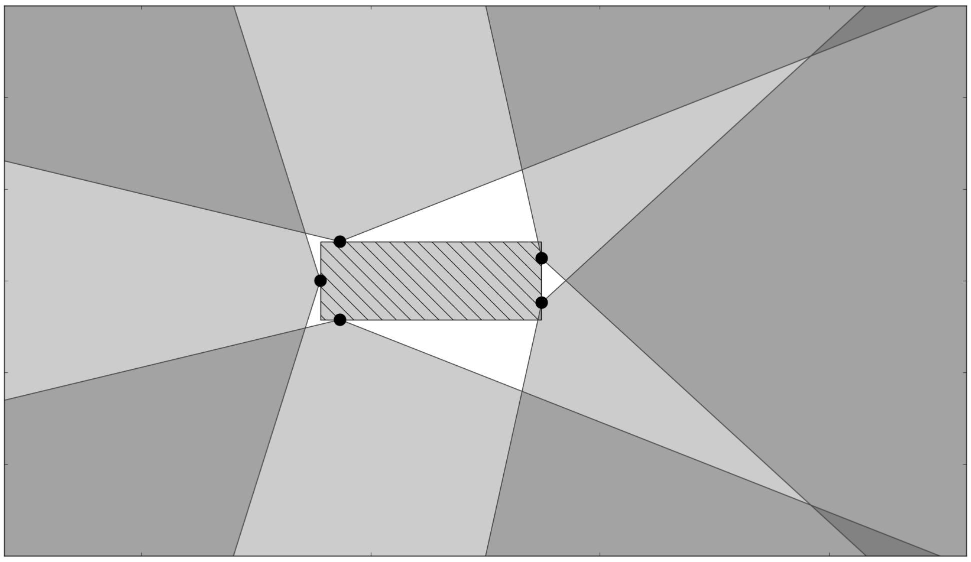

Figure 4.1: SCALA Cocoon coverage of the car's surroundings, car facing right

all mounted at heights between 40 cm and 55 cm above the ground, where most other vehicles and obstacles can be seen from without a vertical angle offset.

Figure 4.1 shows the setup we are working with. The car, which is facing to the right, is represented by a rectangle in the middle, and the SCALA mounting positions are represented by black circles. The gray wedges originating from them visualize their fields of view, with darker areas showing where two or more sensors have overlap in their field of view and white areas showing area not covered by any sensor's field of view. In front of the car, two sensors cover the area being driven towards. On the other three sides, most of the area is covered by a single sensor. The white areas indicating coverage holes all lie within at most two meters of the vehicle, close enough to not play a significant role in localization scenarios in typical highway or urban street environments.

4.2 Odometry

Odometry data are provided through a decoder on the *CAN* bus (*Controller Area Network*), independently of SCALA scans. There, an extractor provides measurements of the car's current velocity v in m/s and the car's current yaw rate ω (the rate at which the vehicle's heading angle changes) in rad/s at a frequency of 100Hz. However, the actual frequency at which that data is read is lower. This is explained further in section 4.3.1. Note that this does not make a significant impact: both signals are smooth enough for there to be no significant loss of information when sampling with a frequency of 25Hz.

4.3 Occupancy Grid Map

This section seeks to document our implementation of an occupancy grid map. For the theoretical background we refer to section 3.3.

The occupancy grid map we are working with has square cells and square shape, but allows tweaking resolution and size of the map through res and $cells$ parameters giving the length of a cell's side in meters and the amount of cells extending from the origin in all four cardinal directions, respectively. The car, which we will also be referring to as *host*, is always in (or very near to) the center of the map. Specifically, it always is at most half a cell away from the origin in both dimensions. Of course, as the car is moving, it will change position eventually. This is reflected in two key properties of the map: the host's position is saved as an offset from $(0, 0)$, and the map is shifted to compensate for vehicle motion whenever the host position exceeds a certain threshold. This will be explained in detail in section 4.3.3.

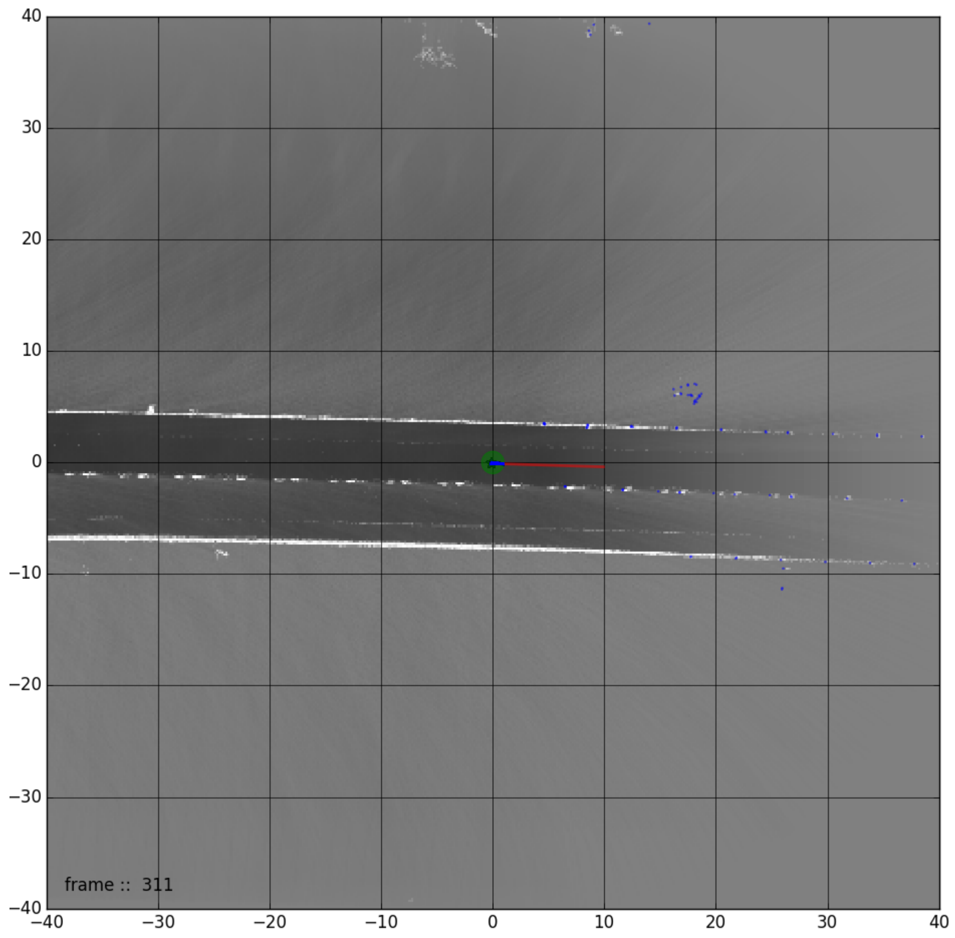

Figure 4.2: A typical view of the occupancy grid map displaying a scenario in which the car is driving on a relatively straight bit of road.

Some of the features whose only purpose is to increase performance will be omitted in this section as they have no relevance to the functionality.

Figure 4.2 shows a typical view of the occupancy grid map. In the center, the host is represented by a green circle. On top of that, the arrangement of blue star shapes represents the particles used for the Monte Carlo particle filter described in section 5.2, one star per particle. A red line originating from the green circle visualizes the direction the host is heading towards currently. On the map, black areas are free space, white areas are occupied and gray areas are unknown. Overlaid on the map in blue is the last set of scan points, in this case delivered from the SCALA mounted on the front left of the car. A frame counter is displayed in the bottom left corner.

The occupancy grid map works in two steps. In the *prediction* step (sections 4.3.2 and 4.3.3), the host movement is applied and in the *correction* step (section 4.3.4), the map is updated with new measurements.

4.3.1 Input Data

The occupancy grid map receives a new set of input data after every cycle of 40 ms, triggered by a designated SCALA sensor providing its data set. On this trigger, current odometry data of the car (velocity v in m/s and yaw rate ω in rad/s) is read from a buffer along with scan data of each sensor, including their mounting position (section 4.1) and a UNIX timestamp in microseconds. The mounting position, provided in vehicle coordinates (section 4.3.2), consists of an x-offset, an y-offset and a ϕ-offset.

We split this time step up into five time steps, one for each sensor, and calculate a new vehicle pose for each of them to gain more accurate projections. In section 4.2 we noted that odometry data is not read with its actual frequency. Instead, it is only read once per set of input data, i.e. once every 40 ms. Then, as the five sensors generally are not synchronized to complete their scans at the same time, the odometry data is used for five different time steps. This, along with the errors in the odometry data itself, are causes of incorrect paths being calculated for the host. While this isn't always noticeable on the constructed map it still isn't a desirable effect.

The scans are provided as a polar histogram in which scan points have been sorted into a number of angle bins (columns) and range bins (rows). Scan points are given in polar coordinates (distance and angle) in the sensor's coordinate system. In this coordinate system, the sensor is placed at $0, 0$ and is facing towards positive x. Each scan point comes with a $layer$ attribute (see section 3.4) of 0, 1, 2, or 3, with 0 denoting the lowest layer and 3 denoting the highest layer. As part of this work, the OGM was changed to immediately discard layer 0 scan points due to their tendency to include significantly more ground detections.

This histogram will be used to update the occupancy grid map. For this, it is projected onto the occupancy grid map and occupancy probabilities are updated. Note that binning scan points into a histogram which is then projected onto the occupancy grid map introduces slight inaccuracies. However, those have little impact if the histogram consists of a sufficient number of bins in both dimensions.

Note that gathering scan points on different levels (in 3D) leads to scan points with the same range reading corresponding to different distances on the 2D map. For this, the scan points are projected down into two dimensions in a preprocessing routine, allowing us to treat them equally.

4.3.2 Motion Model

To describe the motion of the car, a suitable motion model needs to be chosen, considering the trade-offs in accuracy of simpler models and disadvantages in computational complexity of more complex models [LJ00; LVL14; SRW08]. Depending on the data available to the algorithm, one may structure the motion model in an arbitrarily complex manner. However, we have the advantage of receiving odometry measurements twenty-five times per second, alleviating the need for intricate model designs. Instead, we can use the *constant turn rate and velocity* model [BF08] which assumes the turn rate and velocity to be constant between odometry measurements, describing a circular path in combination with a simple *bicycle model* [LVL14]. The bicycle model is a very simplistic model of dynamic vehicle motion, entirely disregarding many influencing factors, such as the lateral direction [Cha04]. While not perfect, this model allows very good approximation because fundamentally, a car's movement can be broken down into a series of arcs due to the limited freedom of movement of the wheels.

The state vector $x_{host,t}$ used by the occupancy grid map to keep the pose of the vehicle up to date within its coordinate system consists of three components:

$$x_{host,t} = \begin{bmatrix} x_t \\ y_t \\ \phi_t \end{bmatrix} \tag{4.1}$$

Here, t is a time index and the heading angle ϕ_t is measured in radians. Before setting up the motion model, we should take a look at the two coordinate systems used here.

The main coordinate system is the map's coordinate system, it's properties are apparent in figure 4.2 and the text describing it. The host state is saved in this coordinate system. We define a heading angle ϕ of 0 to be heading parallel to the x-axis towards positive x. Increasing the heading angle ϕ corresponds to a counter-clockwise turning motion (in figure 4.2). The host's position is measured in the middle of the rear axle. Thus, a state of $[0,\ 0,\ \frac{\pi}{2}]^T$ has the car positioned at the center of the map, with most of its body above the x-axis, pointing upwards.

The host coordinate system is a right-handed coordinate system with its origin located in the center of the rear axle and its x-axis pointing towards the front center of the vehicle. This coordinate system is used as a reference coordinate system in the calculation of the vehicle's new pose and thus the new host coordinate system for the following time step. This means that the transformation matrix to convert coordinates from host coordinates into world coordinates is different for each time step as the host navigates the world.

With this distinction made, we can set up the motion model: The function, given in algorithm 4.1, first calculates the motion of vehicle car in its own coordinate system,

Algorithm 4.1 OGM motion model

```
1: function MOTION_MODEL($x_{host,t-1}$, $\triangle t_t$, $v_t$, $\omega_t$)
2:     $\triangle\phi \leftarrow \omega_t \triangle t_t$
3:     if $\omega_t = 0$ then
4:         $\triangle x \leftarrow v_t \triangle t_t$
5:         $\triangle y \leftarrow 0$
6:     else
7:         $\triangle x \leftarrow \frac{v_t}{\omega_t} sin(\triangle\phi)$
8:         $\triangle y \leftarrow \frac{v_t}{\omega_t}(1 - cos(\triangle\phi))$
9:     end if
10:    $x_{host,t} \leftarrow x_{host,t-1} + \begin{bmatrix} cos(\phi_{t-1}) & -sin(\phi_{t-1}) & 0 \\ sin(\phi_{t-1}) & cos(\phi_{t-1}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \triangle x \\ \triangle y \\ \triangle\phi \end{bmatrix}$
11:    return $x_{host,t}$
12: end function
```

then transforms the motion into the map's coordinate system and returns the map's new state of the host. The algorithm receives four input parameters:

- $x_{host,t-1}$, the old host state
- $\triangle t_t$, the time difference since the last update in seconds
- v_t, the velocity towards the car's positive x-direction in meters per second
- ω_t, the yaw rate in radians per second

In line 2 the difference in heading angle is calculated to later be added onto the previous heading angle and to be used for turn calculation in the next few lines. Lines 3-9 calculate the distance in x- and y-direction driven on the curve described by $\triangle t_t$, v_t and ω_t according to their equations. See section 4.3.2.1 for the derivation of these equations. In the case of the yaw rate being 0 (for perfectly straight motion), the equations given in section 4.3.2.1 don't work as they would require dividing by zero. Instead, we can simply calculate the distance driven in x-direction as the total distance driven, given by the product of time and velocity $\triangle t_t v_t$. The distance driven in y-direction (to the side) then clearly is 0.

Line 10 transforms the motion back into the map coordinate system. Before being added onto the old coordinates, the x- and y-components are rotated by the old heading angle so the curve continues in the direction the car was heading up to this time step. The heading angle has the difference in heading angle applied to it.

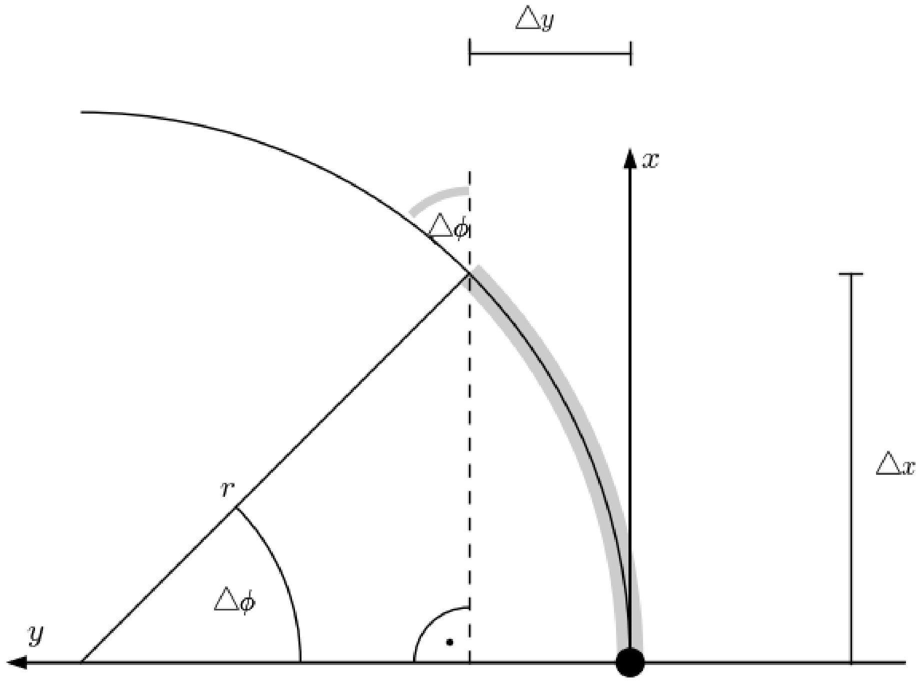

Figure 4.3: Visualization of the turn model used for the host motion in the OGM

4.3.2.1 Derivation of the equations for curved motion

We will take a look at where the equations used in algorithm 4.1 come from. Time indices will be omitted for the sake of readability. For this, we have to imagine the car driving on a circle such as the one in figure 4.3. Note that the x-axis here is pointing upwards and the y-axis is pointing to the left. In the figure, the car is represented by the black dot on the bottom, facing towards the top of the figure. In this time step, its heading angle has changed by $\triangle\phi$. Without a change in heading angle, the car, going straight, would have followed along the direction of the arrow pointing upwards. Instead, it drives along the highlighted part of the circle, an arc with angle $\triangle\phi$. The forward change in position we intend to calculate is labeled $\triangle x$, the sideways change in position $\triangle y$.

To understand the equations, we will consider the circle a unit circle while we calculate the changes of position and apply an appropriate scaling factor later. The unit circle assumption allows for simple equations when calculating the length of the sides of the triangle completed by the dashed line in the center of the figure, which will prove essential in a second. It now becomes clear that $\triangle x$ is given by the height of that triangle, $sin(\triangle\phi)$; slightly more tricky is $\triangle y$, which is given by the difference between the radius of the circle and the base length of the triangle , $1 - cos(\triangle\phi)$. At the end, we simply apply the scaling factor $\frac{v_t}{\omega_t}$ to convert the distances from radians to meters, obtaining the equations

$$\triangle x = \frac{v_t}{\omega_t} sin(\triangle \phi) \tag{4.2}$$

$$\triangle y = \frac{v_t}{\omega_t} (1 - cos(\triangle \phi)) \tag{4.3}$$

4.3.3 Map Shift

To keep the host centered in the map we shift the map and the host opposite the way the host is moving whenever it leaves the center of the map. In effect, the map boundaries are shifted along with the host movement. This is done right after the motion update is applied, before the map is updated with new scan data.

The map shift is described in algorithm 4.2. Once the new host state $x_{host,t}$ is known, its coordinates are divided by the resolution res and rounded to the nearest integer to get the cell index of the host in lines 2 and 3. The map is then shifted the opposite way. Cells newly added to the map are initialized with the map prior. At last, the host is moved back towards the center; specifically it is returned to at most half a cell away from the origin in both dimensions.

Algorithm 4.2 Map shift

```
1: procedure SHIFT_MAP($x_{host,t}$)
2:     $\triangle cells_x = round(\frac{x_t}{res})$
3:     $\triangle cells_y = round(\frac{y_t}{res})$
4:     Shift whole map $\begin{bmatrix} -\triangle cells_x \\ -\triangle cells_y \end{bmatrix}$ cells, discard cells past map edges and fill new cells with map prior
5:     $x_{host,t} = \begin{bmatrix} x_t - res\ \triangle cells_x \\ y_t - res\ \triangle cells_t \\ \phi_t \end{bmatrix}$
6: end procedure
```

4.3.4 Correction step

The *correction* step receives as its input a polar histogram of the scan points. It is used to update the log odds of the cells covered by the sensor's field of view. We will discuss our inverse sensor model (see also section 3.3.2) and look at how it is implemented in combination with the polar histogram. We don't assume detected objects to have a specific thickness, instead only the cell a scan point is mapped into will be considered occupied.

The histogram contains the number of scan points that fall into a cell for each cell. Each scan point is evidence for its cell being occupied. To integrate this evidence into the OGM, the histogram is multiplied with the log odds factor determined by the detection probability p_d, this way each detection causes the same amount of evidence to be applied to the OGM later. That log odds factor influences how much the occupation probability increases. Detecting a scan point at range r from the sensor provides evidence that there are no obstacles at ranges smaller than r in that direction. So cells between the sensor and the first detection in an angle bin should decrease their occupancy probability. The free space part of the inverse sensor model is computed in advance. It assigns a likelihood of being empty to every range bin, allowing us to simply copy part of this data set (in log odds) into the histogram up to the first detection. The unknown space on the outer edge, having been 0, doesn't need to be changed explicitly. That is because log odds of 0 represent our maximum entropy state - we have no information about this part of the map, so the occupation probability is 0.5 and the log odds are 0.

Figure 4.4 visualizes how cells of the polar histogram correspond to cells of the occupancy grid map, which uses a Cartesian coordinate system.

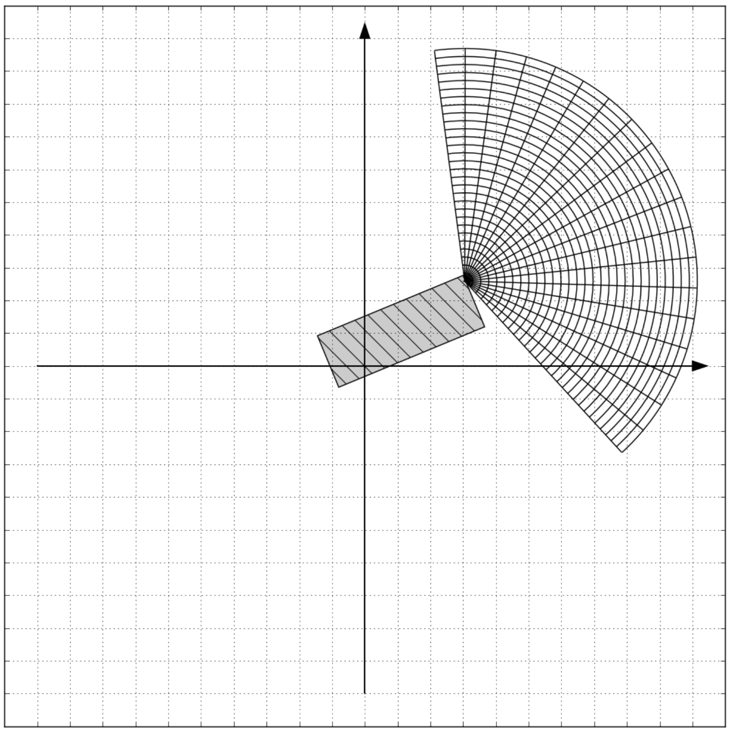

Figure 4.4: Polar histogram of a front-mounted sensor's field of view mapped onto the occupancy grid map

Algorithm 4.3 Correction

```
1: procedure CORRECT_MAP($h_{polar}$)
2:     $h_{polar} \leftarrow log\_odds\_factor\ h_{polar}$
3:     for $\phi \in angles(h_{polar})$ do
4:         if obstacles detected in angle bin $\phi$ then
5:             Fill $h_{polar}$ with $ism$ up to the nearest detection
6:         else
7:             Fill $h_{polar}$ with $ism$
8:         end if
9:     end for
10:    Project $h_{polar}$ onto $m$ and add the occupancy values together
11: end procedure
```

Algorithm 4.3 describes how this process is integrated with the log odds (see also section 3.3.1) to update the occupancy grid map. In line 2, the histogram is multiplied by the log odds factor determined by p_d. Then, in lines 3 through 9, each column of the histogram is traversed along the range axis to allow for simple application of the inverse sensor model on a per-angle basis. The precomputed values of the inverse sensor model are copied over, stopping at the first detection if there are any. Finally, the polar histogram is projected onto the occupancy grid map and its occupancy probability log odds are added to those of the occupancy grid map in line 10.

The precomputed part of the inverse sensor model was designed to place more probability of cells not being occupied on cells closer the the sensor, because objects closer to the sensor generate scan points more reliably. This is also the reason the probability of not being occupied tends to 50% as distance increases past the point of reliable sensor detections. Figure 4.5 shows a plot of the precomputed values.

4.4 Objective

We are given odometry data and scan point clouds generated by five SCALA sensors mounted all around the car; both of these inputs are imperfect. Odometry measurements are not free of noise and biases are observable, especially on the yaw rate signal. Furthermore, the measurements do not fully account for physical circumstances. Scan point clouds are flawed in that false negatives are inherent to the system. Obstacles are less likely to be detected with increasing distance to the vehicle. Small laser beam reflection angles also cause obstacles to not be picked up. False positives also occur, albeit less frequently. They are usually caused by highly dynamic objects such as rain drops or foliage.

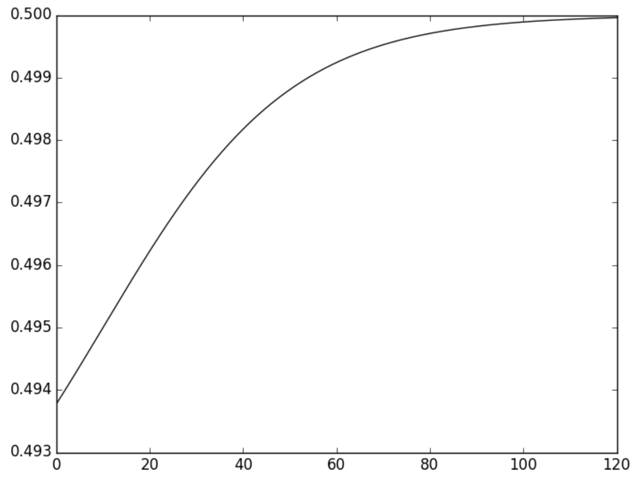

Figure 4.5: Precomputed inverse sensor model placing more probability of not being occupied on cells near the sensor

Based on these inputs, a local occupancy grid map is maintained. Our task is to build and improve a particle filter working with this existing occupancy grid map to run live Monte Carlo Localization. Here, *live* refers to the fact that pose estimates provided by the particle filter are fed back into the occupancy grid map. This will improve the OGM's motion compensation step in which the host advances along its path on the map, in turn producing a cleaner map which can be used for other applications such as path planning. More importantly, it will also give us a better estimate of the path taken by the vehicle.

5 Approach

This chapter will present the base implementation of a particle filter used for SLAM in combination with the occupancy grid map presented in section 4.3 and continue to build on that to explore various ways of improving estimates produced by the filter. Both the mentioned occupancy grid map and the particle filter have been implemented in Python 2.7, however this work will stick to pseudo code to explain important concepts.

We will begin by setting up a classic particle filter and properly connecting it to the occupancy grid map. After that has been accomplished we will turn our focus towards improving the particle filter in terms of estimate accuracy.

Note that from here on, time indices, whenever possible, will be omitted for the sake of readability or simply because there is no need to keep the whole data sequence around when we only need the latest data set.

5.1 Environment

Python 2.7 is used for the implementation. Various packages that provide important functionality to some of the features presented in this chapter are used:

- *SciPy* [JOP+01] provides many packages useful for scientific computing, including all the other items on this list, as well as packages for common linear algebra use cases and some coordinate mapping functionality
- *NumPy* provides efficient multi-dimensional arrays [WCV11], randomness in many different formats and arbitrary data types
- *Matplotlib* [Hun07] provides a framework for plotting data
- scikit-learn [PVG+11] provides the clustering algorithm we use, see also section 3.5

The listed features are representative of what these packages are used for in our implementation, they are not meant to be a complete list of features.

Algorithm 5.1 Particle filter iteration

```
1: procedure ITERATE_PARTICLES($v, \omega, scan$)
2:     if $particles$ uninitialized then
3:         SPAWN_PARTICLES($v, \omega$)
4:     end if
5:     EVOLVE_PARTICLES()
6:     UPDATE_PARTICLE_WEIGHTS($v, \omega, scan$)
7:     RESAMPLE_PARTICLES()
8: end procedure
```

5.2 Particle Filter Implementation

In this section we are constructing the foundation of this work; we are implementing a particle filter based on section 3.2.2, specifically algorithm 3.2. For the beginning, we will keep its interface minimalistic. Interfaces irrelevant to its functionality, such as those for visualization, will not be mentioned at all. In fact, in the beginning we will only have a single function interfacing with the occupancy grid map, the ITERATE_PARTICLES($v, \omega, scan$) function which receives current odometry data and a set of scan points. The $scan$ parameter is provided in the form of a list of scan points in polar coordinates. This function executes one iteration of MCL and handles initial particle creation, as can be seen in algorithm 5.1. Note that the odometry parameters are *not* passed to the particle evolution function but instead to the particle weighting function. This is because we consider them measurements, just like scan point clouds. The odometry parameters are part of what we want to estimate and therefore part of the pose saved with each particle. Apart from that, everything else is comparable to algorithm 3.2; other missing parameters are explained by the necessary data being accessed in other ways due to implementation details.

We define a particle to be an array [1] of length six, with the first five of the components used for position, angle and velocities, and the last component for the weight. The pose will also contain v and ω besides the obligatory x, y and ϕ. Due to the similarity of 'ω' and 'w' we will use $weight$ for the last component. Then, a particle will look like this: $[x\ y\ v\ \phi\ \omega\ weight]^T$. Furthermore, we will use n to denote the number of particles. The following subsections describe some of the functions used in the iteration loop in more detail.

[1]Throughout this work, arrays are zero-indexed, meaning an array of length three consists of items 0, 1 and 2.

Algorithm 5.2 Particle spawning

```
1: procedure SPAWN_PARTICLES($v, \omega$)
2:     $particles \leftarrow$ array of length $n$
3:     for $i \in [1, ..., n]$ do
4:         $particles[i] \leftarrow \begin{bmatrix} 0 & 0 & v & 0 & \omega & \frac{1}{n} \end{bmatrix}^T$
5:     end for
6: end procedure
```

5.2.1 Particle Spawning

Before working with the particles we need to initialize them in a meaningful way. Here, we are in luck: As the particle filter is running in parallel with the occupancy grid map, we essentially are performing *tracking*: we know the initial pose of the vehicle and our task is to keep the pose updated as the vehicle moves through the world. In the OGM, the initial vehicle pose is known to be at $(0, 0)$ with a heading angle of zero. Algorithm 5.2 describes the initialization process.

5.2.2 Particle Evolution

In this step, the path determined by the particles' states is followed for a time period of $\triangle t$ and then some noise is applied to the resulting poses. First, an initial prediction is made for each particle using the motion model described in section 4.3.2 and algorithm 4.1. Afterwards, we need to introduce noise into the particle motion. To apply process noise we draw inspiration from a motion model more complex than the one used for initial predictions. Note that this second model is *only* used for *process noise*. Process noise is used to handle influences on the state vector that are not encompassed by the motion model.

[RHG14] assumes the host is driven by acceleration inputs $a\ [\frac{m}{s^2}]$ and $\alpha\ [\frac{rad}{s^2}]$ for v and ω, respectively. They distinguish between two models:

- Acceleration inputs act as an impulse on v and ω just before the end of a time step. This approach does not affect x, y and ϕ.
- Acceleration inputs are constant over the course of a time step, also known as *zero-order-hold discretization*. This approach adjusts x, y and ϕ in accordance with the acceleration inputs by integrating a and α over the time step's length.

We will use the more realistic and common zero-order-hold discretization. Given an acceleration vector $acc = [a\ \alpha]^T$, the following equations can be used to update the host state hypothesis given by one particle, with MOTION_MODEL being a suitable motion model matching the state vector, such as the one in section 4.3.2:

$$G \leftarrow \begin{bmatrix} \frac{(\triangle t)^2}{2} cos(particles[i][3]) & 0 \\ \frac{(\triangle t)^2}{2} sin(particles[i][3]) & 0 \\ \triangle t & 0 \\ 0 & \frac{(\triangle t)^2}{2} \\ 0 & \triangle t \end{bmatrix} \tag{5.1}$$

$$particles[i][:5] \leftarrow \text{MOTION\_MODEL}(particles[i][:5]) + G\ acc \tag{5.2}$$

Note the indexing in equation 5.2, here we use Python's slice notation to denote which part of an array we are accessing. Specifically, $[:5]$ refers to the first five indices, stopping before index 5. MOTION_MODEL is an implementation of algorithm 4.1.

We will sample a and α from zero mean Gaussians with standard deviations $\sigma_a = 5.0$ for a and $\sigma_\alpha = 2.5$ for α. These values have proven to work in practice. Using acceleration inputs acc sampled from these distributions gives us the desired particle evolution functionality as described in algorithm 5.3. The algorithm loops over all particles and evolves them individually. Lines 14 and 15 sample acceleration inputs and line 16 calculates the state vector update using G (equation 5.1). Finally, line 17 applies the state vector update.

5.2.3 Particle Weighting

We need to calculate weights for every particle. They will be determined by finding importance factors both for odometry measurements and for scan points and normalizing their product. In the end, we normalize them to get our final set of weights which can be used for resampling. See section 3.2.1 for more background information.

5.2.3.1 Weighting based on scan points

We start with the likelihoods for the detections. Looking back on sections 3.2.2 and 3.3, this seems rather straightforward: To calculate a particle i's importance factor, given a scan $scan = s_1, s_2, s_3, ...$, project the scan points into the occupancy grid map based on the particle's pase and multiply the values of all the cells hit by scan points together to obtain the particle's importance factor:

$$particles[i][5] \leftarrow \prod_{pt\ \in\ scan} p(\text{map cell hit by } pt) \tag{5.3}$$

Algorithm 5.3 Particle evolution

```
 1: procedure EVOLVE_PARTICLES()
 2:     for $particle \in particles$ do
 3:         $\triangle\phi \leftarrow particle[4]\ \triangle t$
 4:         if $particle[4] = 0$ then
 5:             $\triangle x \leftarrow particle[2]\ \triangle t$
 6:             $\triangle y \leftarrow 0$
 7:         else
 8:             $\triangle x \leftarrow \frac{particle[2]}{particle[4]} sin(\triangle\phi)$
 9:             $\triangle y \leftarrow \frac{particle[2]}{particle[4]} (1 - cos(\triangle\phi))$
10:         end if
11:         $particle[0] \leftarrow particle[0] + cos(particle[3])\triangle x - sin(particle[3])\triangle y$
12:         $particle[1] \leftarrow particle[1] + sin(particle[3])\triangle x + cos(particle[3])\triangle y$
13:         $particle[3] \leftarrow particle[3] + \triangle\phi$
14:         $a \leftarrow$ sample from $\mathcal{N}(5.0^2, 0.0)$
15:         $\alpha \leftarrow$ sample from $\mathcal{N}(2.5^2, 0.0)$
16:         $d \leftarrow G \begin{bmatrix} a \\ \alpha \end{bmatrix}$
17:         $particle[:5] \leftarrow particle[:5] + d$
18:     end for
19: end procedure
```

However, there are a lot of technicalities this simple approach doesn't consider: Scan points may fall out of map bounds, different particles may end up with different numbers of scan points inside the map bounds, and floating point accuracy may be insufficient.

Algorithm 5.4 shows one way to implement particle weighting without these issues. Lines 5 through 7 convert the scan points from polar coordinates in the sensor's coordinate system to Cartesian coordinates in the vehicle's coordinate system. Line 8 turns the OGM's log odds into probabilities (see equations 3.11 and 3.12) and then calculates the logarithm of them. This reduces floating point inaccuracy issues normally encountered during multiplication of many small numbers between 0 and 1. Lines 9 through 25 iterate over all particles, converting the scan points from vehicle coordinates into world coordinates based on the pose of particle i and then iterating over all scan points. Every scan point that is within the OGM's boundaries is looked up inside it; the resulting logarithmic probability is added to the particle's weight so far and the scan point is counted towards the number of scan points considered for this particle's weight.

Afterwards, particles that didn't have any scan points end up inside the OGM boundaries will be assigned a weight of 0, the initial value of $weight$. For other particles, the final weights are calculated. Because different particles are positioned differently in the world, the set of scan points will be projected into the OGM differently for each of them. This causes different particles to have different amounts of scan points end up inside the OGM bounds. We need to avoid giving higher weights to particles with fewer scan points inside the OGM, so we normalize the importance factors against the number of scan points used to calculate each factor. Let n_{pts} be the number of scan points used in the calculation of a particle's importance factor. We then normalize by taking the n_{pts}th root of the particle's importance factor. In logspace this corresponds to division by n_{pts}. This functionality is described in line 21. Lastly, we recover linear values from the logarithmic ones in line 22 and normalize what have been importance factors until now to obtain weights in line 26.

This approach is known as the *likelihood field* model ([TBF05], section 6.4). It does not consider free space information gained from sensor readings and it does not account for dynamic obstacles. The model can not directly be motivated through physical sensor properties but it works well in practice, is relatively simple to calculate and produces smoother weighting functions than the physically motivated *beam model* they [TBF05] present in section 6.3.

5.2.3.2 Weighting based on odometry

To incorporate the odometry measurements into particle weighting, we assume them to have a normally distributed error and assign them weights accordingly, both for v and for ω. For reference, v typically ranges from 0 to 20 $\frac{m}{s}$ for city driving and from 20 to 60 $\frac{m}{s}$ for highway driving while ω typically ranges up to $\pm$ 0.7 $\frac{rad}{s}$ for turning.

The choice of using normal distributions to calculate the weights is near. Using a standard deviation $\sigma_v = 5$ for v and $\sigma_\omega = 0.04$ for ω, we will be working with $\mathcal{N}(5^2, v)$ and $\mathcal{N}(0.04^2, \omega)$, respectively. Using their probability density functions to calculate importance factors gives us algorithm 5.5. It calculates the two weights for all particles, multiplies them with the existing weights and finally normalizes them again. This allows particles to have odometry deviating from the odometry input data as long as their pose estimates make up for the lower odometry weights by providing better scan matching weights.

Figure 5.1 shows how different values of v and ω in the poses give different importance factors for an odometry measurement of $v = 20\frac{m}{s}$ and $\omega = 0.1\frac{rad}{s}$.

Algorithm 5.4 Particle weighting

```
 1: procedure UPDATE_PARTICLE_WEIGHTS($scan$)
 2:     if no scan points in $scan$ then
 3:         return
 4:     end if
 5:     Convert $scan$ into Cartesian coordinates
 6:     Rotate $scan$ by sensor angle offset
 7:     Offset $scan$ by sensor mounting position
 8:     $log\_map \leftarrow log($EXTRACT_MAP$(map))$
 9:     for $i \in \{1, ..., n\}$ do
10:         $scan_w \leftarrow scan$ converted into world coordinates based on $particles[i]$'s pose
11:         $weight \leftarrow 0$
12:         $n_{pts} \leftarrow 0$
13:         for $pt \in scan_w$ do
14:             if $pt$ not outside $map$ boundaries then
15:                 $idx \leftarrow$ OGM_INDEX_FROM_COORDINATES$(pt)$
16:                 $weight \leftarrow weight + log\_map[idx]$
17:                 $n_{pts} \leftarrow n_{pts} + 1$
18:             end if
19:         end for
20:         if $n_{pts} > 0$ then
21:             $weight \leftarrow \frac{weight}{n_{pts}}$
22:             $weight \leftarrow e^{weight}$
23:         end if
24:         $particles[i][5] \leftarrow weight$
25:     end for
26:     NORMALIZE_PARTICLE_WEIGHTS()
27: end procedure
```

Algorithm 5.5 Particle weighting (odometry part)

```
1: procedure UPDATE_PARTICLE_WEIGHTS_ODOMETRY($v, \omega$)
2:     for $i \in \{1, ..., n\}$ do
3:         $1_v \leftarrow \frac{1}{\sqrt{2\pi\sigma_v^2}} e^{-\frac{(particle[i][2]-v)^2}{2\sigma_v^2}}$
4:         $weight_\omega \leftarrow \frac{1}{\sqrt{2\pi\sigma_\omega^2}} e^{-\frac{(particle[i][4]-v)^2}{2\sigma_\omega^2}}$
5:         $particles[i][5] \leftarrow particles[i][5]\ weight_v\ weight_\omega$
6:     end for
7:     NORMALIZE_PARTICLE_WEIGHTS()
8: end procedure
```

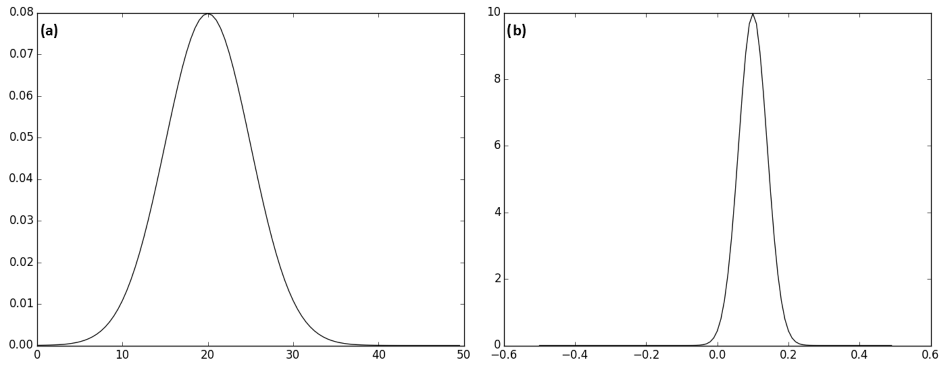

Figure 5.1: Odometry-based weighting function. **(a)** $v = 20\frac{m}{s}$, **(b)** $\omega = 0.1\frac{rad}{s}$.

5.2.4 Particle Resampling

In this step, the particles are resampled according to their weights. One difference to the standard implementation pattern is how the weights are handled: Instead of normalizing them after resampling, we keep them with the particles as a quality measure of the particles' estimates, and use them during pose extraction in section 5.3. This does not cause any problems to the rest of the particle filter because the weights aren't used elsewhere; afterwards they are overwritten by the UPDATE_PARTICLE_WEIGHTS$(v, \omega, scan)$ function before being accessed again.

5.3 Pose Extraction

Here, we introduce another interface to our particle filter before we can connect it to the occupancy grid map. A key component of the relationship between the OGM and the MCL is the exchange of data: Every time step, MCL requires odometry data, timestamps and scans from the sensors, and in return the OGM receives updates to the pose of the vehicle. For this, we need some way of extracting a pose from the particle filter, which, under the hood, has n different poses to choose from.

There exist many different ways to handle pose extraction; a few common ones are compared in [LR09], however not all of them fit our tracking purposes. Four different approaches to pose extraction were considered:

- Choose the best particle. This approach is simple but it exhibits a huge problem: As particle weights are only based on the last set of measurements - which come from different sensors with slight calibration errors and cover very different areas of the map - particle weights are not very stable. A particle with large weight may be deemed sub-average in the next time step.

- Choose the weighted mean of all particles. Another simple approach, but also a flawed one: it doesn't handle multimodal particle distributions with which averaging all particles would produce a pose somewhere between the peaks of the distribution.
- Choose the weighted mean of the best 10% of all particles. In an attempt to combat multimodality and uneven distributions of outlier particles, we restrict the particle set to the $\lceil n/10 \rceil$ best particles before computing their weighted mean. However, this approach will still select particles from different peaks of the particle distribution.
- Cluster the particles in the xy-subspace, choose the weighted mean of the cluster with the best weight. This approach has no problem handling multimodal distributions and in practice produces stable pose sequences. DBSCAN (section 3.5) with parameters $\epsilon = 0.001$ and $minPts = 3$ is used to cluster the particles.

We will choose the clustering approach for the reasons mentioned above.

5.4 Connecting the Particle Filter to the Occupancy Grid Map

Now that we've set up our OGM and our MCL including some interfaces, we can connect the two. The OGM works in two steps: prediction and correction. In the correction step, the map has its occupancy probabilities updated, there is nothing MCL needs to do here. In the prediction step, the host is moved and, in our case, the map is shifted. That's where we need to tie in because with MCL, the pose updates of the host need to come from there. So, before the host is moved, MCL has to run its iteration. This also means that the occupancy grid map at this point will not have the newest sensor readings incorporated, and for good reason: MCL needs the map to be in its old state to best determine where the new sensor readings fit in and thus where the host has moved.

So, in the OGM's scheduling loop, instead of calling its motion model (section 4.1), we need to run the MCL iteration function (algorithm 5.1) followed by querying the particle filter's pose extraction function (section 5.3) for the pose the map's host state will be overwritten with. Next we'll need another interface between MCL and the OGM: After the map has been shifted, additionally to the map's host state we also need to shift the particles in the particle filter. For this, a SHIFT_PARTICLES($\triangle cells_x, \triangle cells_y$) function is called that implements the pose update part (line 5) of algorithm 4.2 for the particles.

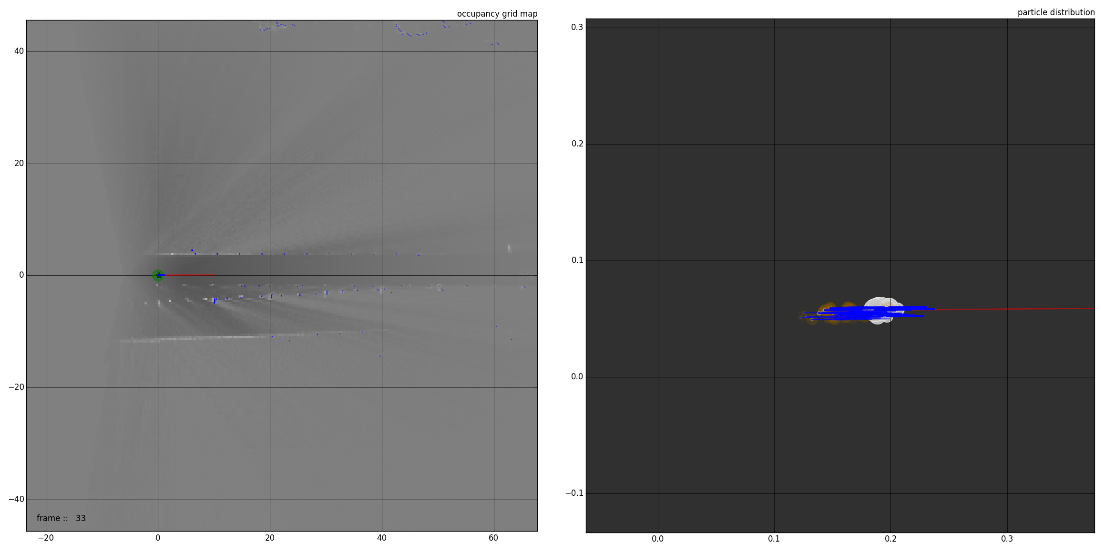

Figure 5.2: A view of the particles including their weights and the previous iteration displayed next to the OGM

Lastly, we changed the grid map's visualization to render the particles on top of the map (figure 4.2). The visualization now includes a second plot showing a closeup view of the particles represented by transparent white circles with their size corresponding to the particle's weight. Behind those particles, the particles of the previous time step are rendered with fixed size, less opacity and in yellow color. Figure 5.2 shows the visualization; note that one can also zoom in and out of both plots during the animation.

5.5 Expanding the Particle Filter

In this section we will discuss two changes made to the particle filter described in section 5.2 in order to improve its performance, i.e. the accuracy of the approximations given by it. The results of these changes will be discussed in chapter 6.

5.5.1 Alternative Particle Weighting Function based on Scan Points

[HR16] build a different scan point weighting function based on their inverse sensor model. They replace equation 5.3 with the more sophisticated

$$particles[i][5] \leftarrow \prod_{pt\ \in\ scan} \frac{1}{2} + p_d(p(\text{map cell hit by } pt) - \frac{1}{2}) \tag{5.4}$$

This model considers the confidence we have in detections; with increasing confidence in detections we should be devaluing diverging hypotheses more and with decreasing confidence they should be given a chance to prove themselves. This produces a less aggressive weighting function because all factors are moved to within $\pm\frac{p_d}{2}$ of 0.5; as p_d tends to 1.0, equation 5.4 tends towards the more aggressive equation 5.3. They compute p_d from the false alarm rate and the signal-to-noise ratio, however neither of those are known to us. Instead, we substitute a constant factor $\frac{1}{2}$ for p_d to make use of this model's dampening property. Preliminary tests have shown even smaller factors to negatively affect approximation quality.

The rest of algorithm 5.4 stays the same.

5.5.2 Utilizing Alpha Filters for Odometry

Here, we replaced the odometry part of the weighting function with an alpha filter (section 3.2.3) for each of the two odometry measurement parameters, reducing the weighting function to scan matching only. Two parameters, $\alpha_v = 0.05$ for the velocity gain and $\alpha_\omega = 0.4$ for the yaw rate gain were introduced. The yaw rate gain was chosen to be relatively high so the particle filter can quickly react to yaw rate changes which significantly alter the course of the vehicle. The velocity gain was chosen smaller in order to not unnecessarily restrict the particle filter. These values may seem counter-intuitive given that the yaw rate signal provided by the vehicle tends to be biased while the velocity signal is quite accurate. However they work very well in practice, with smaller α_ω parameters actually worsening path estimates.

This change removes algorithm 5.5. The newly introduced alpha filters, described by algorithm 5.6, are placed inside the particle evolution function (section 5.2.2), between lines 1 and 2 of algorithm 5.3. Since the alpha filters require odometry measurements we also need to adjust the parameter list for EVOLVE_PARTICLES() which previously didn't contain v or ω.

Algorithm 5.6 Alpha filters for odometry

```
1: procedure ALPHA_ODOMETRY_UPDATE($v, \omega$)
2:     for $i \in \{1, ..., n\}$ do
3:         $particles[i][2] \leftarrow particles[i][2] + \alpha_v(v - particles[i][2])$
4:         $particles[i][4] \leftarrow particles[i][4] + \alpha_\omega(\omega - particles[i][4])$
5:     end for
6: end procedure
```

5.6 Scan Point Sampling

In this section we aim to give scan points we deem important more influence on particles' weights and lessen the impact of scan points we deem unimportant, erroneous or clutter. We also want to reduce bias introduced by uneven scan point distributions. In [UT13], a couple of sampling strategies in scan matching are described. Listed are only those applicable to OGM SLAM based on SCALAs:

- No sampling
- Uniform subsampling
- Selecting points such that the distribution of normals among the selected points is as large as possible

We will draw inspiration from these and examine some other sampling strategies. Considering that we are primarily aiming to improve approximation quality and not performance, the first two approaches are not applicable either.

Our approach is simple: based on the original set of scan points, construct a different set of scan points by changing which scan points appear how often. Then change the weighting function to work with this modified set of scan points. In algorithm 5.4, this step would be inserted between lines 7 and 8 for sampling methods based on Cartesian coordinates, and between lines 4 and 5 for sampling methods based on polar coordinates. Here, *sampling* doesn't necessarily refer to sub-sampling, it just refers to the fact that we are using weights assigned to scan points to sample a certain number of scan points from the original set; sometimes with replacement, sometimes without; sometimes we end up with fewer scan points, sometimes we end up with more scan points - with some duplicates, of course. That is because we investigate what type of distribution is beneficial for the particle filter. To optimize the particle filter's runtime to some extent, the number of scan points could be further reduced through uniform subsampling without significantly impacting the quality of approximations.

Algorithm 5.7 Sample scan points based on distance to the car

```
1: procedure SCANPT_SAMPLING_DISTANCE($scan$)
2:     for $pt \in scan$ do
3:         if RANDOM_FLOAT() $< \frac{\text{range reading of } pt}{20}$ then
4:             drop $pt$
5:         end if
6:     end for
7: end procedure
```

5.6.1 Sampling based on Distance to the Car

Scan points at close distances are often undesirable: In general, scan points at small ranges make up a disproportionately large fraction of the entirety of scan points, making this method of sampling useful even for detections that in themselves aren't undesirable. Undesirable scan points can be caused by raindrops or spray close to the sensor, they can be caused by parts of the car itself if the sensor isn't installed properly or they could be other vehicles that are going to move elsewhere soon. This method of sampling discards some scan points with close distances to the car. Specifically, preliminary tests have indicated 20 meters to be a good threshold for that. The probability of a scan point surviving this step is then proportional to its distance from the car, up to the threshold. Scan points bearing a range reading of 0 will certainly be discarded and scan points at a distance of 20 meters or more will certainly be kept.

Denoting the number of scan points n_{pts}, algorithm 5.7 describes this procedure. RANDOM_FLOAT generates a random or pseudo-random floating point number between 0 and 1.

Figure 5.3 shows an example situation in which this filtering method removes some undesirable scan points as well as some scan points in densely populated areas of the scan point cloud. In the figure, one can see the car facing top-right. Some thirty meters away, road boundaries can be made out. A few cars have been detected from behind, most of them at a distance of 60 meters or more. 5.3(a) shows the scan point cloud before sampling. The three scan point clusters closest to the vehicle consist of significantly more scan points than those stemming from similar obstacles at greater distances: the road boundaries to the right of the vehicle at x=5 and at x=35 are structurally equivalent, however they generate very different amounts of scan points. In 5.3(b), after applying the sampling step, the three clusters closest to the vehicle have been removed entirely or weakened significantly. 5.3(c) displays the scan points discarded in this sampling step.

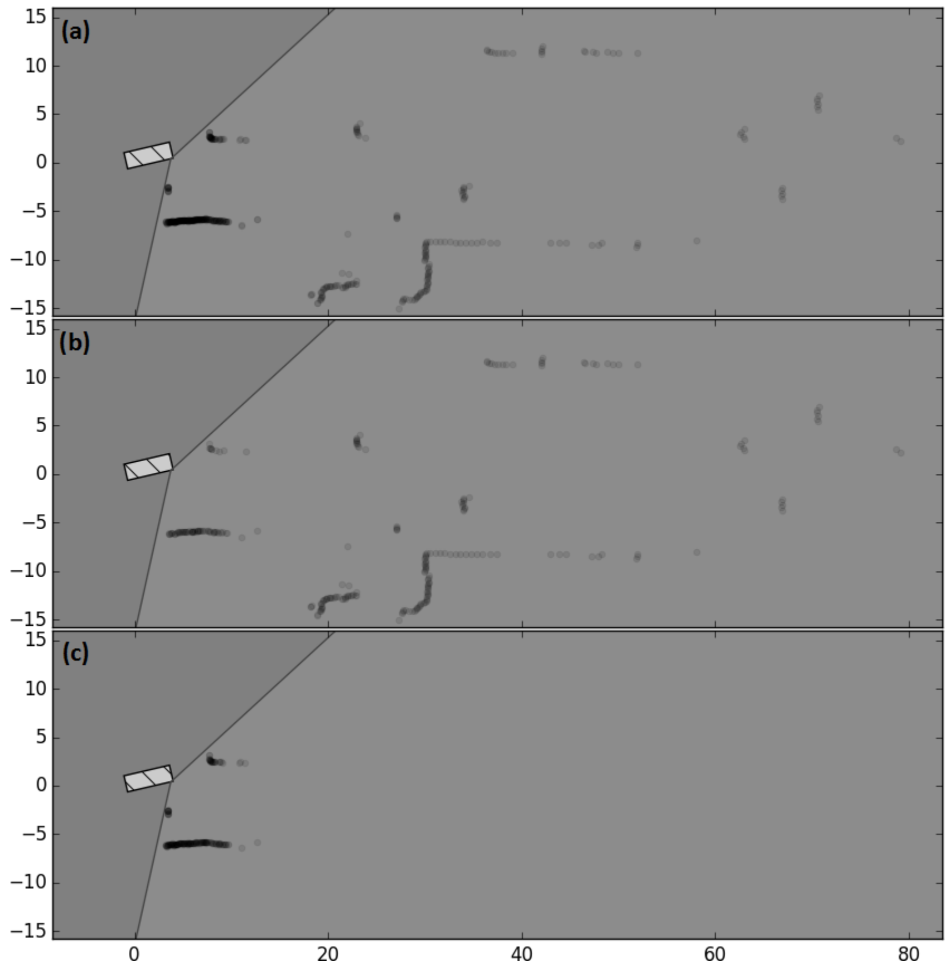

Figure 5.3: Range based sampling thinning out the scan point cloud near the car. **(a)** Before; **(b)** after; **(c)** discarded scan points.

A pleasant side effect of this method of sampling is that it, on a smaller scale, has the same effect as the sampling strategy for far detections which specifically increases the impact of scan points detected at large distances. That strategy is discussed in section 5.6.2.

5.6.2 Sampling for Far Detections

This method of sampling is also based on the distance between scan points and the car, however it serves a very different purpose. Here, scan points far away from the car are given significantly more influence by duplicating them. Of course, this also affects clutter or unrelated object detections, however those do not occur in large numbers. Mostly relevant detections are affected and we can afford to duplicate these scan points.

Specifically, scan points with a range reading of 80 meters and above are added to the set of scan points an additional ten times.

At these distances, guardrails by the side of straight roads are no longer detected, however buildings and other tall obstacles such as signs mounted above the highway will still be picked up by the top-most layer of scans (see also section 3.4). These obstacles are static and thus very good for reliable localization.

Detections at large distances are particularly useful for estimating the angle of the car. To understand this relation, consider the following hypothetical using a laser sensor that allows scan points an error of 0.5 meters to each side. A detection at a range of 10 meters could have been generated anywhere within a detection window of 5.73 degrees. The same detection with a range of 100 meters however will stem from a detection window of only 0.57 degrees.

This makes reliable detections at large ranges very desirable to us. Usually, they are vastly outnumbered by closer detections, but using this filter we can increase their numbers to a point at which they make an actual impact on the particle weights.

5.6.3 Sampling Scan Points based on Angle between Road and Obstacles

Straight, empty stretches of roads typically generate scan point clouds consisting mainly of scan points stemming from guard rails or other road boundaries. When few other obstacles are nearby, guard rails and other road boundaries look the same over long distances. This makes it difficult to localize the vehicle along the road in the direction of travel; reliable estimates of velocity become difficult to obtain. To remedy this situation we place more weight on scan points for obstacles that are orthogonal to the direction of travel. These obstacles are important for velocity estimation as they are not invariant with regards to the position of the vehicle along its predicted path. To avoid having to calculate a road or lane radius, we use the next best thing available, the vehicle's current heading angle. This approximation is quite appropriate on straight roads where the vehicle's heading angle usually closely matches that of the road.

To calculate the angle of obstacles relative to the vehicle we need to calculate their surface normals. Using the neighborhood of a scan point for surface normal calculation may accidentally include scan points from other obstacles. Therefore we need to cluster the scan point cloud. However, many surfaces are more complex than a simple line so we must consider the immediate neighborhood of a scan point within its cluster. Hence, this approach combines the advantages of local neighborhoods and clusters: we get the locality of neighborhoods without having the set of considered scan points contain scan points stemming from nearby separate structures.

Algorithm 5.8 Sample scan points based on angle between road and obstacles

```
 1: procedure SCANPT_SAMPLING_ROAD_ANGLE($scan$)
 2:     $particle \leftarrow$ POSE_EXTRACTION()
 3:     $clusters \leftarrow$ DBSCAN($scan, 1.0, 3$)
 4:     $weights_{scan} \leftarrow$ array of length $|scan|$, initialized with $0.0$
 5:     for $i \in \{1, ..., |clusters|\}$ do
 6:         for $pt \in clusters[i]$ do
 7:             $neighbors \leftarrow \{pt_{neighbor} \in clusters[i] \;\Big|\; ||pt - pt_{neighbor}|| < 2\}$
 8:             $mean \leftarrow \sum_{pt \in neighbors} \frac{pt}{|neighbors|}$
 9:             $deltas \leftarrow neighbors - mean$
10:             $scatter \leftarrow \sum_{delta \in deltas} delta \otimes delta$
11:             $vals,\ vecs =$ EIG($scatter$)
12:             if $vals[0] < vals[1]$ then
13:                 $smallest\_vec \leftarrow vecs[0]$
14:             else
15:                 $smallest\_vec \leftarrow vecs[1]$
16:             end if
17:             $weights_{scan}$[index of $pt$] $\leftarrow \left| smallest\_vec \begin{bmatrix} cos(particle[3]) \\ sin(particle[3]) \end{bmatrix} \right|$
18:         end for
19:     end for
20:     Normalize $weights_{scan}$
21:     Sample the amount of non-clutter scan points from the weighted scan points
22: end procedure
```

We start out by clustering the scan points using DBSCAN (section 3.5) with parameters $\epsilon = 1.0$ and $minPts = 3$. Then, for each scan point, we look at its neighborhood of scan points closer than two meters within the cluster. We calculate the surface normal of the surface described by that neighborhood. To obtain the surface normal, we calculate the eigenvector belonging to the smallest eigenvalue of the point neighborhood's scatter matrix. Assuming that neighborhood stems from a single surface, the resulting eigenvector is its surface normal. The cosine of the angles between surface normals and the vehicle's heading direction are used as weights. Acute angles between vehicle and surface correspond to angles near 90° or 270° between vehicle and surface normal. Thus the absolute value of the cosine matches our intent of giving smaller weights to surfaces parallel to the vehicle path. Scan points that weren't grouped into a cluster are considered clutter and will be discarded. In the end, the weights are used to resample the scan point cloud.

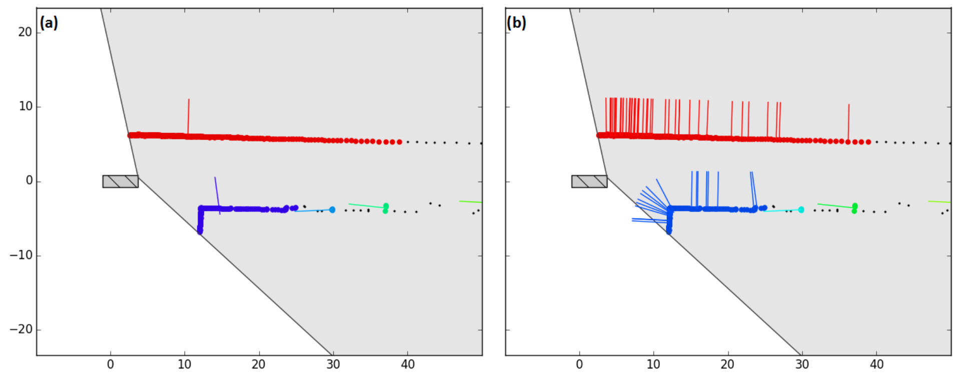

Figure 5.4: Effect of using neighborhoods within clusters for surface normal calculation. **(a)** clustering only, **(b)** clustering combined with neighborhoods.

Algorithm 5.8 outlines this approach. In line 2 the current pose estimate is fetched, acting as an approximation for the pose of the car. The pose is actually different for each particle, but for typical particle distributions, with only small deviations in angle and position, this approximation is harmless. This allows us to calculate the angles only once instead of for every particle. Lines 8 through 10 calculate the scatter matrix before fetching its eigenvalues and normalized eigenvectors in line 11. The following five lines select the eigenvector belonging to the smaller eigenvalue. That is used to calculate the absolute value of the dot product between the eigenvector and the vehicle heading direction. We take the absolute value because the surface normals may point either direction of the heading direction while we are only interested in the angle between the two. That yields the cosine of the angle between these two vectors. We do not need to normalize it because both vectors were normalized to begin with. The remaining clutter scan points keep their initial weights of 0.

Figure 5.4 shows how considering neighborhoods within clusters produces more accurate surface normals. The identical clusters produced in both versions are highlighted in different colors, outliers are displayed as black dots. In 5.4(a), the same surface normal is used for every scan point within a cluster, not accounting for clusters consisting of what should be considered multiple surfaces. In 5.4(b), each point has a separate surface normal, calculated based on its neighborhood within its cluster. Note that in 5.4(b), only a small subset of scan points are displayed with their surface normal for visibility reasons.

5.6.4 Grid-based Sampling

Proposed in [UT13], this strategy is designed to normalize the scan point density across the space of detections. It can be considered a generalization of the first sampling strategy based on the distance to the car presented in section 5.6.1. As explained there, the amount of scan points detected at a certain range decreases drastically as the range decreases. This is because with smaller distances, sensor beams are thinner and thus more beams can hit a certain area (see also figure 4.4). Another cause is the detection rate decreasing with increasing distance. This places a disproportional amount of relevance on detections at close ranges and very little relevance on far-out detections; however we'd like all objects on the map to be utilized equally.

The implementation of this strategy is straightforward: Start by binning all scan points into a Cartesian grid laid over the scan point cloud. Then, cells are classified into populated and sparse cells depending on how many scan points are contained in a cell. We calculate the average amount n_{avg} of scan points across populated cells and sample n_{avg} scan points from each of these cells. These sampled scan points are combined with all scan points in sparse cells to form the resulting scan point cloud. It makes sense to choose a relatively coarse grid so that populated cells contain a significant number of scan points. We will choose four by four meters for the cell size and consider cells containing at least 3 scan points populated.

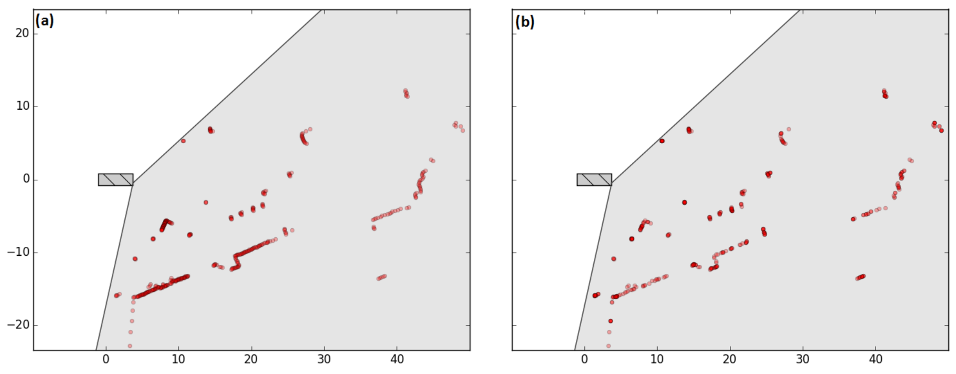

Figure 5.5: Effect of grid-based sampling. **(a)** before, **(b)** after.

Figure 5.5 shows the effect grid-based sampling has on a scan point cloud recorded at a typical inner-city intersection. In 5.5(a), one can see a high concentration of scan points on obstacles close to the car such as the two fragments of road boundary on the bottom. In 5.5(b), the density of scan points is approximately equal across the whole map except for some outliers.

5.6.5 Sampling based on Surface Smoothness

In [ZS14], they perform scan to scan matching. They investigate selecting edge scan points and planar scan points to use for scan to scan matching based on a subset of scan points. They propose a measure of smoothness, assigning high values of $c(pt)$ to edge points and low values of $c(pt)$ to planar points:

$$c(pt) = \frac{1}{|scan| \; ||pt||} \left|\left| \sum_{\substack{pt2 \, \in \, scan \\ pt \neq pt2}} (pt - pt2) \right|\right| \tag{5.5}$$

To understand this measure of smoothness, consider a set of eleven scan points arranged in a line with constant distance between two neighboring scan points. The sixth scan point, with equally many scan points on both sides, will be assigned 0 because the summands corresponding to the closest two scan points cancel each other out, as do those of the second-closest two scan points and so on. The eleventh scan point on the other hand will be assigned a much larger value because the summands all have the same sign in each of the components; the same thing happens for corner points.

Note that this measure of smoothness is local, it is designed to work on connected scan points. Thus, we will first cluster the scan points using DBSCAN (section 3.5) with parameters $\epsilon = 1.0$ and $minPts = 3$. Then we can weight and sample scan points individually per cluster. We will use edges and corners being less susceptible to erroneously being perceived at an offset along the wall than long, straight sections of it. Drawing some amount of samples from the set of scan points weighted using equation 5.5 will yield a scan point distribution that places significantly more scan points around corners and edges.

5.6.6 Sampling against Dynamic Obstacles

Dynamic obstacles - other vehicles and pedestrians - are not part of the static environment we want the OGM to contain. The OGM itself already does filter out a lot of dynamic object detections simply because they rarely stay in the same place, preventing the OGM from developing high occupancy probabilities for affected cells. However, this doesn't work in stopped traffic. So, generally, we'd like for dynamic obstacles to be removed before they have a chance of influencing the map. Various approaches exist, both with 2D [MON09; WPN15] and with 3D sensors [AA12; MDU11]).

Attempting to emulate more sophisticated approaches at detecting dynamic obstacles, we will use a simple filter that discards a portion of the scan points in a rhombus shape designed to cover most obstacles in front and behind the car where the road is as well as a bit to the side, where other vehicles may be blocking the view. Scan points within that area were given a 25% chance of being discarded, focusing the scan point cloud on objects more likely to be static.

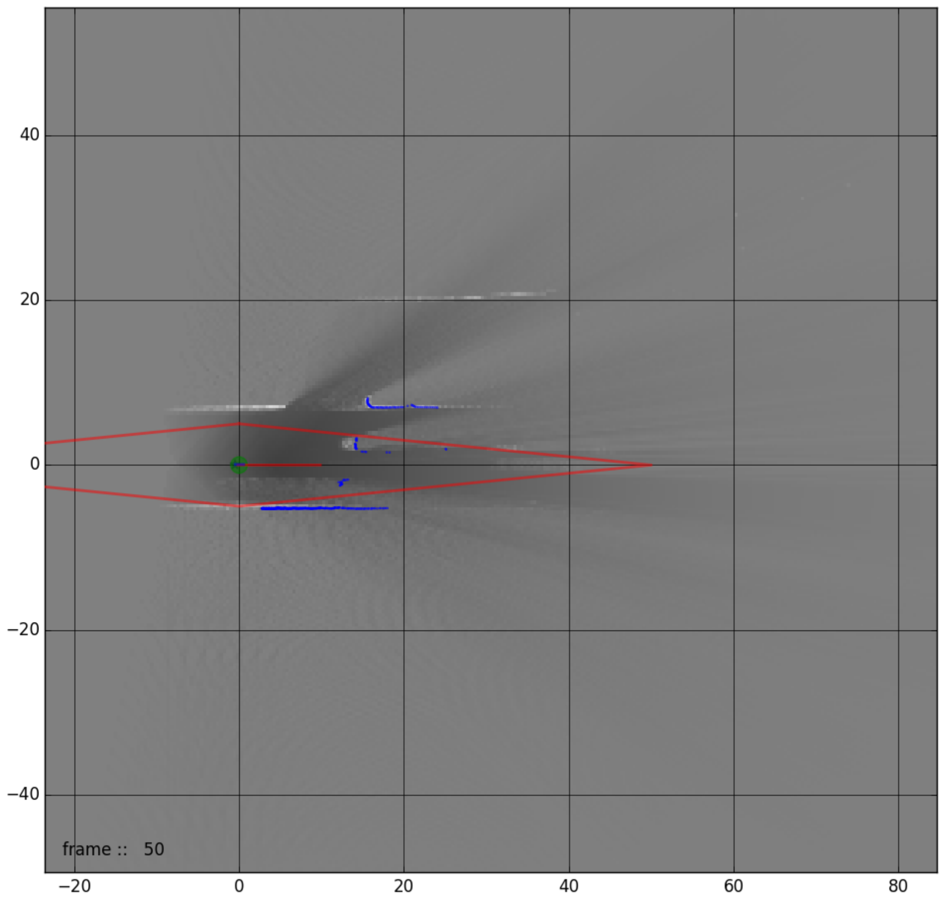

Figure 5.6: Outline of rhombus used for dynamic object pseudo-sampling overlaid on the OGM

Figure 5.6 shows the outline of this rhombus in red overlaid on the OGM displaying a typical situation in which we'd like to get rid of scan points representing other cars. Scan points on the road boundary remain unaffected while the scan points representing the two cars driving in front of the host will be thinned out heavily.

6 Evaluation and Results

In this chapter we show the results of the thesis. We present our method of evaluation and use it to examine and compare the various improvements over the standard particle filter presented in chapter 5. We look at traces recorded in different environments and find out how best to handle them. Before doing any comparisons, we configure the occupancy grid map presented in section 4.3, and, after investigating the MCL changes proposed in section 5.5, set up the the particle filter presented in section 5.2 before comparing the scan point sampling strategies presented in section 5.6.

6.1 Method of Evaluation

6.1.1 Obtaining Reference Data

For real world SLAM applications a common method of evaluation is to obtain ground truth data, which can be used to calculate approximation errors for any given time step. For this, we would use a differential GPS (*DGPS*) system which provides positional data within a very small margin of error, typically around 10 cm. As we have no access to such a system we will make do with the next best thing: *Google Maps* satellite imagery, which we will call *map* in this chapter. In combination with Google Maps we will use Bing Maps, Google StreetView and video recordings recorded along with the traces to better establish reference points that are difficult to make out from a top-down view.

Of course, obtaining ground truth data like this is not as simple as with DGPS. For each data point, consisting of a timestamp along with the vehicle's position and orientation in the OGM at that time step, we need to manually align it on the map. For every test scenario, we first stitch together the map and determine the scaling factor required to align world coordinates to it. Then we pick a couple of time steps, enough to have one data point for every 60 m to 200 m driven by the car. For each of those time steps we export the current scan point cloud into a file so it can be loaded without replaying the whole trace. The scan point cloud is saved in vehicle coordinates, thus finding the rotation and translation required to align the scan point cloud with the map equals finding the car's position and heading angle on the map. A good place to save scan points like this is directly after line 7 of algorithm 5.4. A short Python script then displays the scan points overlaid on the map given a position (x, y) and heading angle ϕ. We will refer to this visualization as the *overlay* from here on out.

Scan point clouds should not be selected at random because they are not all equally suitable for manual ground truth estimation. Whenever possible, one should select scan point clouds according to the following criteria:

- They should contain several different obstacles to best guarantee only one position on the map matches the scan point cloud
- They should contain straight surfaces to best aid in figuring out ϕ
- They should contain reliable scan points at large distances ($\geq$ 100 m) to narrow down guesses of ϕ (see also section 5.6.2)

Additionally, the first scan point cloud should not be chosen directly at the start of the trace. This is because the particle filter requires some time to stabilize while the OGM, which is initialized as entirely unknown, is constructed to an extent allowing stable tracking - typically, two seconds are plenty of warmup time.

Manually aligning the scan point cloud is simple but tedious. We give a work flow that proved to be effective assuming the position of the car is known to within a couple dozen meters: Put in a rough guess for ϕ, display the overlay and use the plot's display of cursor coordinates to get a good first estimate of (x, y). Put in (x, y) and look at the overlay again. By comparing straight lines in the scan point cloud and their corresponding obstacles in the map, ϕ can be improved significantly. Update ϕ and then repeatedly improve the parameters until changes are too small to make a difference on the overlay.

Estimates of paths driven by the vehicle contain the error rate integrated over time. Comparing such a path estimate against ground truth data reveals any biases in velocity in the shape of shortened or elongated path segments, and in yaw rate in the shape of paths continuously drifting to one side.

Figure 6.1 shows the result of aligning a scan point cloud to the map. The vehicle has been localized in the top left part of the image. Each red dot represents a scan point. This example demonstrates the three criteria listed earlier: the scan point cloud shows an emergency lay by which is clearly differentiable from the rest of the otherwise quite similar road. The guardrails provide multiple series of scan points arranged in straight lines. Far off in the distance, about 250 meters away from the car, we can see a few more scan points stemming from some small plastic dividers installed on the central road markings as well as two scan points stemming from the guardrails, and, most importantly, a couple of scan points stemming from a billboard or sign placed on the side of the road. Matching up this sign with Google StreetView data allowed us to obtain a very accurate value for ϕ.

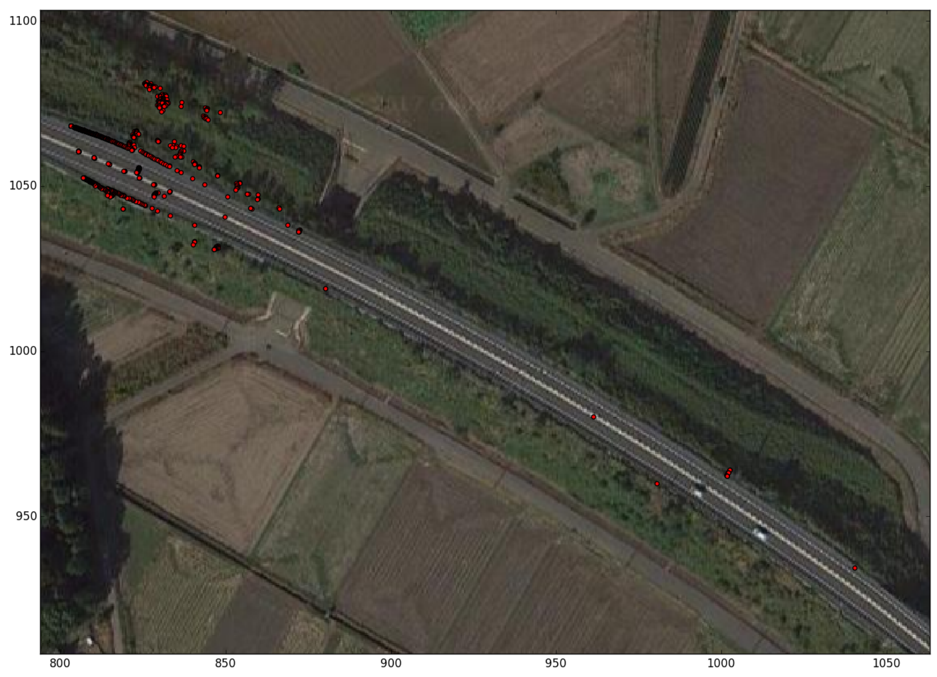

Figure 6.1: Scan point cloud aligned to satellite imagery. Map data: Google, ZENRIN.

6.1.2 Accuracy of Reference Data

We examine the accuracy of our evaluation for scan point clouds chosen according to the criteria listed in section 6.1.1. The map resolution is one pixel width per 19.5 cm. The scan point clouds we chose for manual localization all contained wall segments long enough to manually line up in parallel with the corresponding obstacles on the map despite its limited resolution. We also have several objects at different angles available to support localization. Using those, we can determine the host location up to the map and SCALA accuracy. To account for obstacle edges not always lining up with the pixel grid of the map we allow a two by two pixel area in which a scan point's true position is. Accounting for SCALA accuracy (section 3.4) we obtain the following upper bound for the error in position localization:

$$R(x) = R(y) = 2 * 19.5\ cm + 10\ cm = 49\ cm \tag{6.1}$$

Using this bound we can derive an upper error bound for the heading angle. Scan point clouds fulfilling our criteria contain reliable scan points at distances of at least 100 m. For the sake of this example the scan point cloud consists of exactly one scan point. We

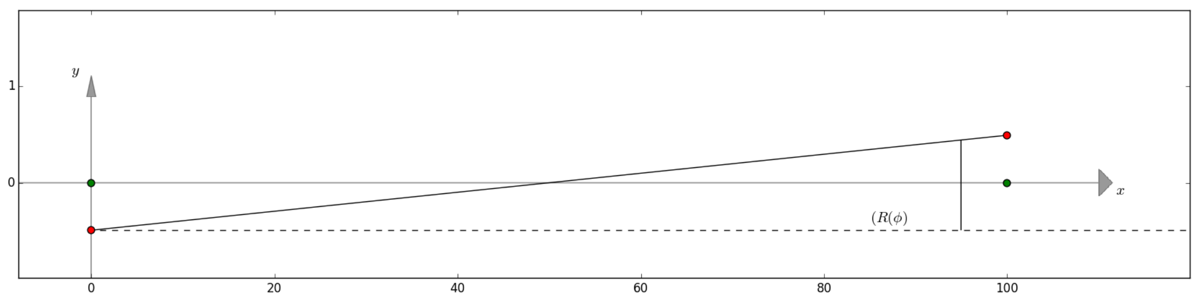

Figure 6.2: Derivation of the ground truth heading angle error. Note that the axis aspect ratio is not equal.

use the true host's coordinate system for the following calculations. The true location of the scan point is straight in front of the vehicle, at an angle of 0°. We construct the worst case scenario for ϕ localization and calculate the error introduced in that situation:

- The host position was localized as far to the right of the heading direction as possible
- The far off scan point was localized as far to the left of the heading direction as possible
- The far off scan point has a range reading of 100 m

The first two items here maximize the error in angle as both of them skew the perceived heading angle in counter-clockwise direction. The last item maximizes the impact of the first two items because lateral shifts of scan points have a larger impact on their angle the closer they are to the sensor. Figure 6.2 visualizes this scenario. The green dots represent the true host and scan point positions and the red dots represent their wrongly localized positions. The angle between the dashed line and the black line connecting the two wrongly localized positions is our upper error bound on manual heading angle localization. It is given by the following equation:

$$sin(R(\phi)) = \frac{100\ m}{2 * R(y)} \tag{6.2}$$

Solving for $R(\phi)$ yields $R(\phi) = 0.5615°$. Note that we do not need to account for errors in x-direction here because the distance between scan point and host is fixed.

6.1.3 Path Evaluation

To quickly review the data used to evaluate a path, this is what we have gathered for a trace:

- Ground truth data: A series of 3-12 data points (ts, x, y, ϕ)
- Vehicle path: A series of thousands of data points [1] (ts, x, y, ϕ), containing data points for the ground truth data's timestamps among them
- A scaled map, useful for visualization

6.1.3.1 Preparing path data for comparison

Before calculating any errors we need to line up the vehicle path with the ground truth data so they both start at the same place, except for some warmup time. Resetting the accumulated error by synchronizing the path at every ground truth data point allows us to separately consider each segment defined by two ground truth data points.

Algorithm 6.1 describes the path synchronization procedure. The path is synchronized at each data point in $ground\_truth$ so the error accumulated in each section becomes visible. The difference between the two conditional branches are minuscule: the first entry in $ground\_truth$ is used to align the whole path, including the warmup time; all other entries are used to align one section only, starting at the timestamp *after* the one listed for the current ground truth data point. This is so the ground truth data point can be compared with the corresponding data point in the driven path *before* the accumulated error is reset. As this is the only difference between the two conditional branches, we will only describe the first branch. Here, a series of points (the vehicle positions) is transformed to align with the first ground truth data point.

Because we will need to apply a rotation matrix in a few lines, we start by moving the series of points onto $(0, 0)$ from where they can be rotated without introducing an additional offset. We are synchronizing at the first ground truth data point entry, so in line 4 we choose the path's corresponding position as translation required for the shift onto $(0, 0)$. In the next line we calculate the angle by which we need to rotate so the heading angles match between ground truth and the vehicle path, this rotation is applied in lines 6 and 7. To finish off, the series of points, now rotated to match ground truth, is shifted onto ground truth at its first timestamp.

[1]The exact length is determined by the number of sensors and the duration over which the trace was recorded

Algorithm 6.1 Synchronize vehicle path with ground truth data

```
1:  procedure SYNC_PATHS($ground\_truth, path$)
2:      for $data \in ground\_truth$ do
3:          if $data$ is first entry then
4:              Subtract ($xy$ entry of $path$ where $ts = data[ts]$) from all $xy$ entries in $path$
5:              $\phi\_diff \leftarrow$ ($\phi$ entry of $path$ where $ts = data[ts]$) $- data[\phi]$
6:              Subtract $\phi\_diff$ from all $\phi$ entries in $path$
7:              Rotate all $xy$ entries in $path$ by $-\phi\_diff$
8:              Add $data[xy]$ to all $xy$ entries in $path$
9:          else
10:             Subtract ($xy$ entry of $path$ where $ts = data[ts]$) from all $xy$ entries in $path$
    after $ts$
11:             $\phi\_diff \leftarrow$ ($\phi$ entry of $path$ where $ts = data[ts]$) $- data[\phi]$
12:             Subtract $\phi\_diff$ from all $\phi$ entries in $path$ after $ts$
13:             Rotate all $xy$ entries in $path$ after $ts$ by $-\phi\_diff$
14:             Add $data[xy]$ to all $xy$ entries in $path$ after $ts$
15:         end if
16:     end for
17: end procedure
```

Figure 6.3 shows the effect of repeated path synchronization. In this scenario, the vehicle drove from the top left to the bottom right part of the road. Blue dots represent data points available in the ground truth data set, red lines represent vehicle path sections. In 6.3(b), only the first ground truth data point was used to localize the vehicle path on the map. In 6.3(a), every ground truth data point was used for synchronization. The error accumulated over each section is much easier to make out in the left half. Note that an artificial error was introduced to the vehicle's path to better visualize this effect.

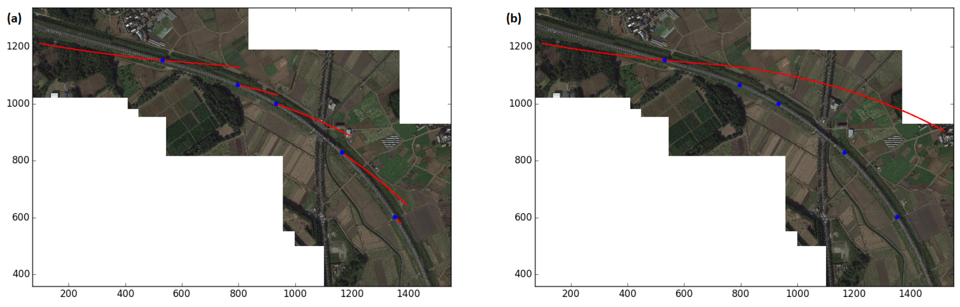

Figure 6.3: The effect of synchronizing paths at every ground truth data point. **(a)** synchronized at every ground truth data point, **(b)** synchronized once. Map data: Google, ZENRIN.

6.1.3.2 Comparing path data to ground truth data

We will use the *mean squared error* (MSE) to quantify the error in path approximations. The MSE is given by the sum of the squared approximation errors. In general, for a sequence of n ground truth data points g_i and n corresponding approximations h_i, it is calculated as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(g_i - h_i)^2 \tag{6.3}$$

Consider a trace and the path taken by the vehicle in it, such as the one in 6.3(a). We have manually established $n = 5$ ground truth data points for this trace. They divide the vehicle path up into six path sections: one section preceding each point plus one short section after the last point. [2] We look at the errors accumulated during each section to calculate the MSE. However, the last section cannot be used for this because it has no ground truth data point at its end allowing us to calculate the section's accumulated errors. The first section cannot be used either because we specifically aligned this section so that it matches the first ground truth data point. The remaining four sections are then used for error calculation.

There, we calculate the MSE as follows:

$$MSE = \frac{1}{n-1}\sum_{i=2}^{n}(g_i - h_{f(i)})^2 \tag{6.4}$$

Note that the summation index i here starts at 2 because we skip the first ground truth data data point. In practice, we do not have the same number of ground truth data points and estimated data points. Therefore we need to use different indices for the two sequences. In equation 6.4 we use a function f that maps ground truth indices to their corresponding indices in the estimated data sequence. In our implementation, corresponding indices are found by comparing the timestamps attached to each data point in both sequences.

Due to the vehicle rotating on the map, errors in the x and y components individually are dependent on the scenario. Instead we calculate the MSE in dxy, the 2-dimensional Euclidean norm. It directly relates to the error in position accumulated over a given time, be it caused by wrong velocity approximations or by wrong heading angle approximations. As will become apparent in the following sections, the accuracy of velocity data provided by the host is quite good already. So good, in fact, that measuring it using satellite imagery is no longer feasible: the error often lies on the same order of accuracy as our reference method. We also calculate the MSE in ϕ.

[2] Scan point clouds extracted for manual ground truth collection have to meet certain criteria outlined at the beginning of section 6.1.1. This is why we usually cannot choose the very last scan point cloud in a trace, resulting in an additional short path section after the last ground truth data point.

6.2 Configuring the OGM

We need to configure the occupancy grid map before we can start examining the various approaches developed in the last chapter. Specifically, we need to pick appropriate values for the following four parameters:

- o_d, the odds ratio multiplied onto grid cells in which scan points are detected. This parameter influences how quickly a cell changes towards occupied when a scan point is placed in it. Here we need to find a balance so dynamic objects do not stay on the OGM permanently and static objects become solid relatively quickly.
- o_{nd}, the odds ratio multiplied onto grid cells between a sensor and its nearest detections. This parameter, along with the inverse sensor model, influences how quickly a cell changes towards free when a scan point is detected somewhere behind it. See section 4.3.4 for the free space calculation integrated in the inverse sensor model.
- res, the OGM resolution, determines the side length of the square cells that make up the OGM. Here we need to choose a resolution fine enough to allow for accurate mapping and tracking, but also coarse enough to get a consistent OGM.
- $cells$, the amount of cells extending in both directions from the origin in both dimensions, is determined by the desired range and resolution of the map.

Considering the five 25MHz SCALAs have a combined average update frequency of 125Hz, o_d needs to be chosen relatively close to 1.0 so dynamic obstacles do not cause too high occupancy likelihoods. Some trial and error has shown $o_d = 1.041$ to be a good choice: obstacles appear quickly and reliably on the OGM while other vehicles mostly don't affect the OGM visually. We will choose o_{nd} similarly: $o_{nd} = 0.961$ works well in practice.

One could think choosing a resolution was simple given our knowledge of the sensors (section 3.4). However, their scanning multiple layers, inaccuracies caused by real world objects being detected from different angles, imperfectly calibrated sensor mounting positions and angles limit the resolution we are able to work with. Figure 6.4 shows one of the more obvious misalignments noticeable in our traces. Two scan point clouds, recorded in successive time steps, are displayed on the OGM. They just about line up within the stripe of cells they occupy. We will set the resolution to that used in the figure, $res = 0.4$. In practice, our particle filter produces worse results when finer resolutions are chosen.

We want the OGM to have a range of more than 100 meters. Not many scan points are returned much farther out, but a good portion of the scan points at this range are valid detections especially useful for angle estimation. We choose a range of 120 meters, so we set $cells = 300$ to yield the desired range at the chosen resolution $res = 0.4$.

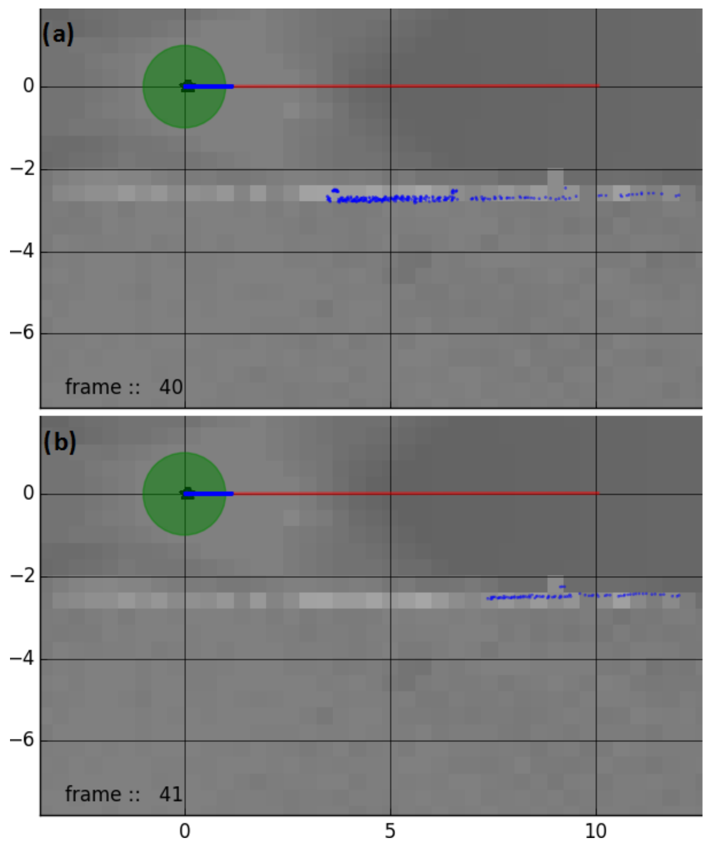

Figure 6.4: Scan point clouds not lining up perfectly between two time steps.

6.3 Test Scenarios

We have four traces available for testing, each set in a different environment.

Factory is set inside a private business estate. The surrounding buildings are warehouses and office buildings. Velocity is limited to a maximum of 20 km/h. Large static objects such as containers, storage shelfs and trucks delivering goods are the type of obstacles encountered in this trace. The route is 650 meters long.

Urban takes place on a curvy four lane urban street with inclines of up to 6%, intersections, traffic lights, pedestrians and other vehicles. Velocities range up to 50 km/h, the route is 1.5 km long.

Country takes place on a slightly curved elevated highway with one lane of traffic going each direction, guardrails on both sides and small dividers installed on the lane markings, separating the two directions of traffic. Few other obstacles are detected past the road boundaries. Here, the vehicle moves at 60 to 80 km/h for 70 seconds.

Highway takes place on a divided highway with four lanes going in the host's direction, starting on a curved onramp and following along for 7.5 km, gaining almost 100 meters of altitude on the way. Velocity is typical for highways in Germany, ranging from 80 km/h to 130 km/h.

Figure 6.5 roughly shows the paths driven in the four traces. 6.5(a) shows *factory*, driving around the central building in counter-clockwise direction starting from the bottom. 6.5(b) shows *urban*, driving towards the bottom of the image. 6.5(c) shows *country*, driving towards the bottom right. 6.5(d) shows *highway*, driving towards the bottom of the image.

6.4 Baseline

To evaluate the improvements we made in chapter 5 there needs to be a baseline to compare against. In this section we will set two baselines: one showing the accuracy of the default odometry data and one showing the accuracy of the particle filter without any of the improvements presented in sections 5.5 and 5.6. We will refer to this particle filter configuration as *base particle filter*. Every test that includes the particle filter will be performed with a set of $n = 500$ particles; more particles don't produce different results and significantly fewer particles make for unstable results.

Three of the four traces available for testing (section 6.3) - all but *country* - were recorded with badly configured sensors. Issues like those shown in figure 6.4 are so prevalent that running the particle filter with some of the sensors disabled actually produces better path approximations. Specifically, we are only using the two forward facing sensors for these traces. Nonetheless we achieve significant improvements over paths constructed purely from odometry signals.

Figure 6.5 shows the paths driven in the four traces reconstructed from odometry data. In 6.5(a), the vehicle drove from the bottom to the top, in the others it drove towards the bottom. 6.5(a,b,d) all show a significant negative yaw rate bias apparent from the path sections ending up to the right of the blue ground truth data points. 6.5(c) shows a significant drift towards the left. However, the path sections all appear to be the right length, indicating that the velocity odometry signal is quite good on its own.

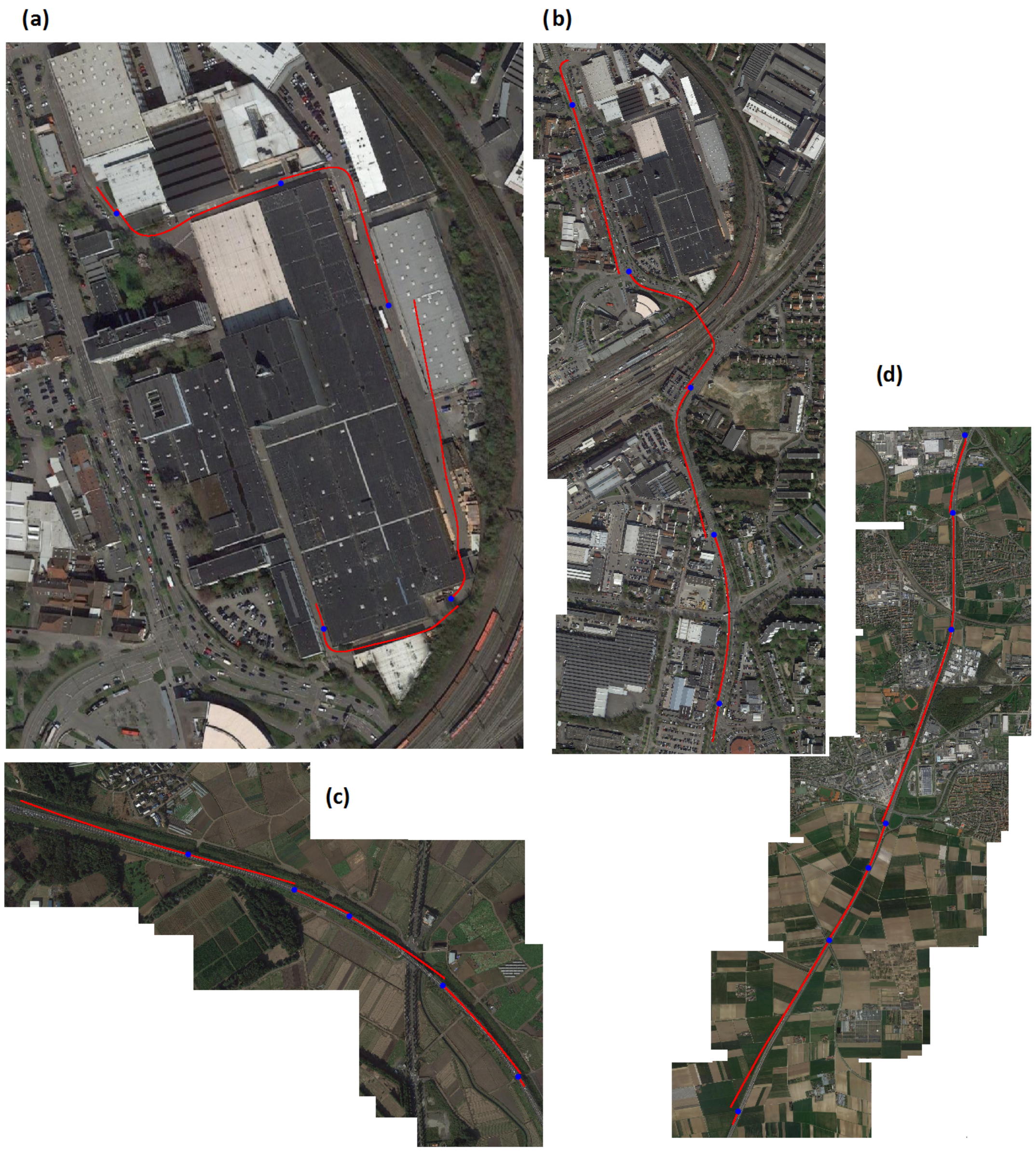

Figure 6.5: The traces available for testing. Map data: Google, ZENRIN, GeoBasis-DE/BKG.

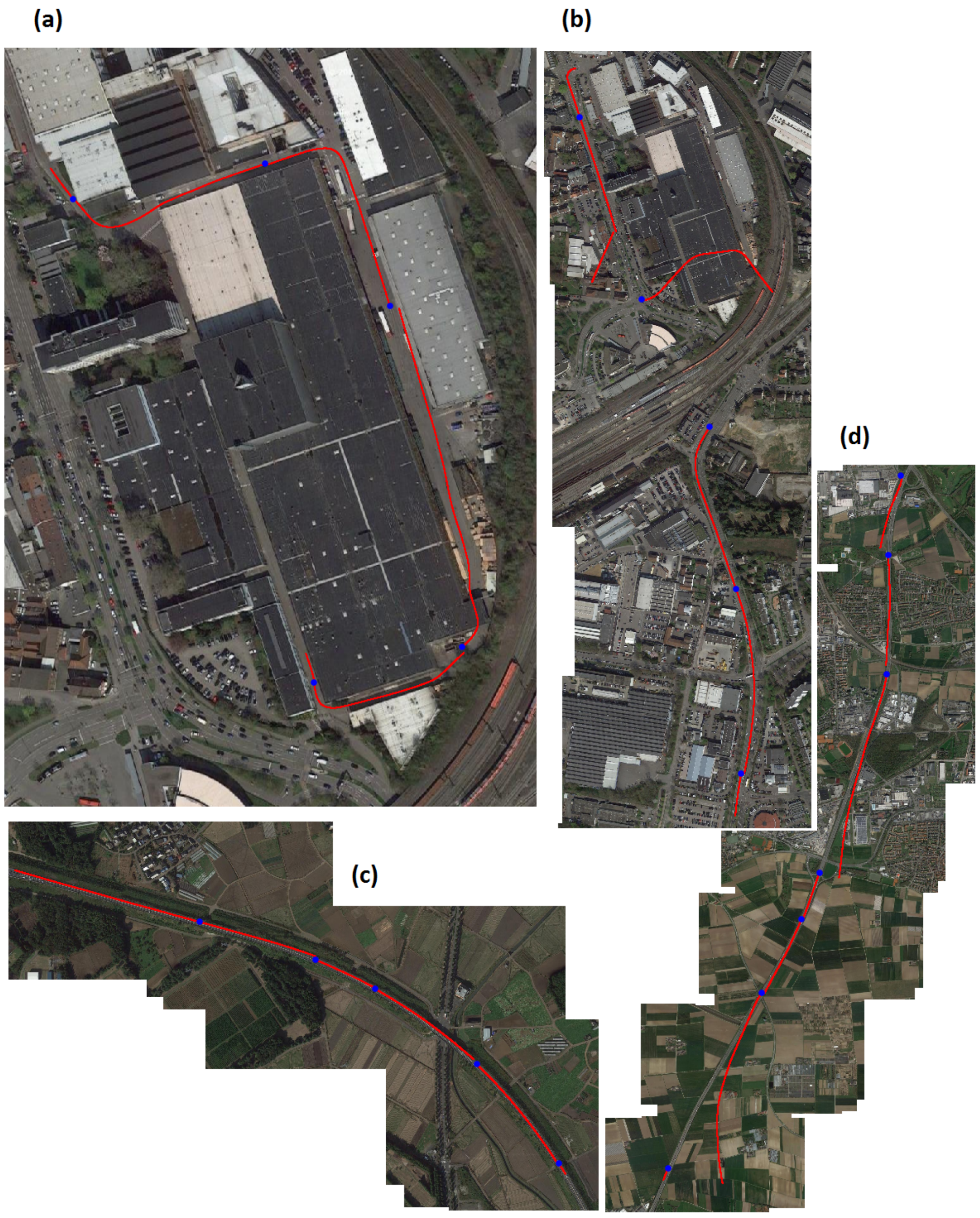

Figure 6.6: The four traces with paths estimated by the base particle filter. Map data: Google, ZENRIN, GeoBasis-DE/BKG.

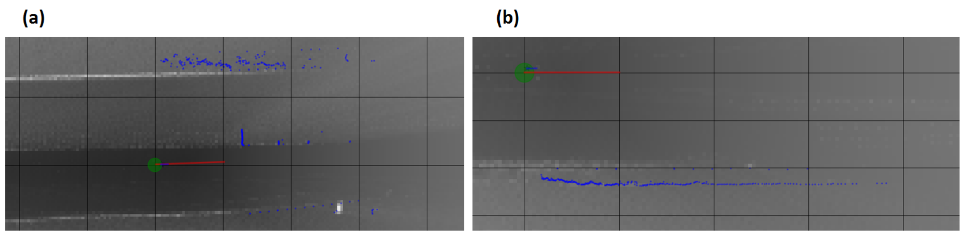

Figure 6.7: Difficulties in the *highway* trace

Figure 6.6 shows the paths estimated by the base particle filter. In *factory* and *country* (6.6(a,c)) the drift has mostly been corrected. In *urban* (6.6(b)) the particle filter got lost twice when stopping and starting for traffic lights. However, as apparent from the path sections that were aligned correctly, the drift has mostly been corrected.

Lastly, 6.6(d) shows *highway*. While the path estimate here does look better than that of *urban*, we have not been able to improve it significantly - not with the configuration presented in the next section and not with the various scan point sampling strategies tested in section 6.6. That is because of a combination of reasons:

- Other vehicles travel on both sides of the host
- Guardrails are not always present on the right-hand road boundary
- A significant part of the trace is spent in areas where the area to the right of the road is occupied with a man-made embankment covered in bushes and small trees. The ground structure there is very uneven compared to flat walls and guardrails. Also, the distance from detections to the sensor is highly subjective to changes in altitude as well as pitch and roll angles of the vehicle. This makes consecutive scan point clouds and even different layers of the same scan point cloud align very badly.

Figure 6.7 shows a truck blocking the view of the left guardrail (6.7(a)) and uneven surface detections on the right-hand side (6.7(b)). Therefore, *highway* results will be omitted from now on.

Table 6.1 shows the MSEs in ϕ [°] and dxy $[m]$ for the two traces that didn't diverge, *factory* and *country*. Listed are the path reconstructed from odometry data and the the path estimated by the base particle filter. For MCL data, the values were averaged over six runs for each test scenario[3]. Values in brackets show the standard deviation of the

[3]Last minute bug fixes prevented us from using previously prepared averages based on a higher number of runs.

values used for the average. We calculate the standard deviation σ of a series of n values v_i as follows:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \overline{v})^2} \tag{6.5}$$

Here $\overline{v}$ is the average of the v_i.

The entries for the path based on only odometry have been calculated deterministically and as such did not require averaging. Note that looking at only this table, comparisons between different traces are hardly possible: traces were recorded in very different environments under very different circumstances, and ground truth data points' distances differ.

While path approximations have been improved over the path reconstructed from odometry data, the standard deviations show that the particle filter doesn't produce particularly consistent path approximations at this stage.

	Factory		**Country**	
MSE	ϕ	dxy	ϕ	dxy
Odometry	30.37	109.28	26.28	169.95
Base MCL	8.16 (6.39)	17.68 (9.47)	3.14 (0.56)	85.06 (54.34)

Table 6.1: Comparison of MSEs between pure odometry data and base particle filter

6.5 Configuring the Particle Filter

In this section we test the improvements presented in section 5.5. Table 6.2 shows the results of the tests ran in this section: *Weights* for the alternative weighting function and

	Factory		**Country**		**Urban**	
MSE	ϕ	dxy	ϕ	dxy	ϕ	dxy
Odom.	30.37	109.28	26.28	169.95	27.10	205.11
Base MCL	8.16 (6.39)	17.68 (9.47)	3.14 (0.56)	85.06 (54.34)	–	–
Weights	6.13 (1.15)	11.72 (1.41)	2.19 (0.31)	132.65 (43.32)	9.42 (2.29)	62.91 (6.94)
Alpha	5.86 (3.29)	7.07 (1.54)	1.93 (0.10)	41.05 (1.16)	10.95 (1.46)	28.97 (9.70)

Table 6.2: Comparison of MSEs between baselines and particle filter configurations

Alpha for odometry alpha filters. The two baselines established in the previous section are labeled *Odom.* and *Base* for pure odometry and the base particle filter, respectively. Standard deviations are again given in brackets, where applicable.

We start by substituting in the alternative weighting function presented in section 5.5.1. This weighting function provides less aggressive weights so particles with lower weights survive for longer, allowing the particle filter to more freely explore hypotheses that start out appearing unlikely. For *factory* and *country*, this reduces the standard deviations of all MSEs, indicating more stable path estimates. The MSEs themselves also went down in all but one case: in *country*, the dxy MSE increased by more than half. This is because there are fewer obstacles in this particular trace which could help discredit bad hypotheses. Note that errors of this magnitude also occurred in the base particle filter's dxy MSEs. For *urban*, this particle filter configuration achieves stable path estimates, albeit the dxy MSE is still almost a third of the odometry baseline's.

Then we add in the alpha filters for v and ω as presented in section 5.5.2. While MSEs in ϕ haven't changed significantly for any of the traces, the dxy MSEs for *country* and *urban* are quartered and halved, respectively.

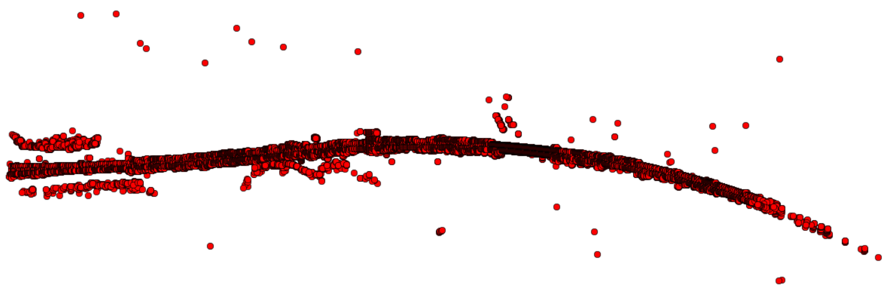

Figure 6.8: Collection of scan points recorded in *country* trace

The reason results for *country* are this much better with alpha filters is that this particular trace contains barely any obstacles apart from those that are part of the road. Figure 6.8 shows a representative subset of all scan points recorded during that trace: apart from some outliers, only a bridge and the forest near the start on the left side have generated scan points past the road boundaries. Before the alpha filter change, this allowed the odometry part of the particle weighting function, which is based on noisy odometry signals, to dominate the overall particle weights. The weights the scan matching part was assigning to slightly misaligned particles weren't small enough to offset the higher odometry weights attached to particles close to the (noisy) odometry signals.

Going forward we will use the alternative weighting function and alpha filters.

	Factory		**Country**		**Urban**	
MSE	ϕ	dxy	ϕ	dxy	ϕ	dxy
Odom.	30.37	109.28	26.28	169.95	27.10	205.11
Alpha	5.86 (3.29)	7.07 (1.54)	1.93 (0.10)	41.05 (1.16)	10.95 (1.46)	28.97 (9.70)
dist	9.70 (3.34)	8.56 (0.98)	1.69 (0.15)	35.23 (6.30)	4.72 (0.43)	22.43 (6.80)
far	5.71 (1.87)	6.45 (1.58)	1.70 (0.19)	40.34 (6.14)	7.29 (1.31)	30.07 (12.89)
angle	7.33 (1.87)	12.65 (4.57)	1.40 (0.24)	36.55 (4.05)	7.79 (6.18)	136.71 (31.45)
grid	3.64 (1.14)	5.66 (0.70)	16.91 (1.67)	119.19 (14.52)	3.98 (0.96)	15.80 (3.02)
smooth	16.19 (3.44)	17.14 (2.78)	4.65 (1.59)	79.50 (19.79)	14.70 (7.84)	177.99 (81.12)
dynamic	5.59 (1.19)	8.24 (2.37)	1.55 (0.12)	39.62 (5.72)	8.92 (2.35)	28.27 (3.44)

Table 6.3: Comparison of MSEs for scan point sampling strategies

6.6 Scan Point Sampling

In this section we evaluate the scan point sampling strategies presented in section 5.6. Following that, section 6.7 will combine some of these strategies to obtain a particle filter that works well on all three traces. Like before, we will give the results of each test series and discuss each strategy's impact on the three traces.

We test the following strategies:

dist	Sampling based on distance to the car (section 5.6.1)
far	Sampling for far detections (section 5.6.2)
angle	Sampling based on angle between road and obstacles (section 5.6.3)
grid	Grid-based sampling (section 5.6.4)
smooth	Sampling based on surface smoothness (section 5.6.5)
dynamic	Sampling against dynamic obstacles (section 5.6.6)

Table 6.3 shows the results of combining the particle filter configured in section 6.5 with each of the sampling strategies listed above. For comparison, the odometry-only path and the configured particle filter are also listed as *odom.* and *alpha*, respectively. The following paragraphs compare the *alpha* baseline to the performance of our sampling strategies.

Sampling based on distance to the car, which linearly downsamples scan points up to 20 meters away from the sensor, improved estimates of *urban* in both evaluated measures. There, many of the detections close to the sensors stem from other vehicles. Decreasing their impact improved the path approximation. The *factory* trace produced worse results. There, most of the close detections should be considered reliable as they belong to static obstacles such as palettes and containers. Removing part of these scan points negatively affects path approximations. Lastly, *Country* produced less stable dxy MSEs, however it did improve the ϕ MSEs. This is likely because most close detections were actually reliable, only some traffic dividers and a few vehicles going the opposite way would produce undesirable detections.

Sampling for far detections, which duplicates scan points past 80 meters, slightly improved the ϕ MSEs of *country* and *urban* at the cost of less stable dxy approximations. This is because more importance was assigned to lining up far detections which can be lined up quite well even with small alignment errors at smaller ranges. However, for *factory*, both measures have changed for the worse. In this environment, there rarely is 80 meters or more of visibility along the vehicle path, thus detections at these ranges are mostly clutter. Increasing their impact negatively affects approximation quality.

Sampling based on angle between road and obstacles, which favors scan points belonging to surfaces orthogonal to the road, was intended to improve velocity estimations. However, as explained in section 6.1.3.2, no significant improvements are possible for velocity estimation. Additionally, this sampling strategy actually significantly worsened path approximations by taking focus away from important scan points. Most MSEs and standard deviations have grown compared to those of *alpha*. Only ϕ MSEs for *country* and *urban* have improved. One possible explanation for *urban* is that road boundaries, of which detections typically only appear within forty meters of a sensor, have been discounted heavily; this would give more influence to obstacles at greater distances, which bear more importance for angle estimates, improving the ϕ MSE.

There is a small silver lining here: looking at the path approximations produced here we can see path sections ending up short of their successive ground truth data point. This shows that velocity estimates in other test scenarios were indeed good and that the particle filter possesses the freedom to deviate from the velocity signals provided to it through the alpha filters in section 5.5.2. It also shows that this sampling strategy did not work as intended, having entirely ignored the importance of other detections. Figure 6.9 shows path sections for (a) *factory*, (b) *country* and (c) *urban* that ended up short of their real world destinations.

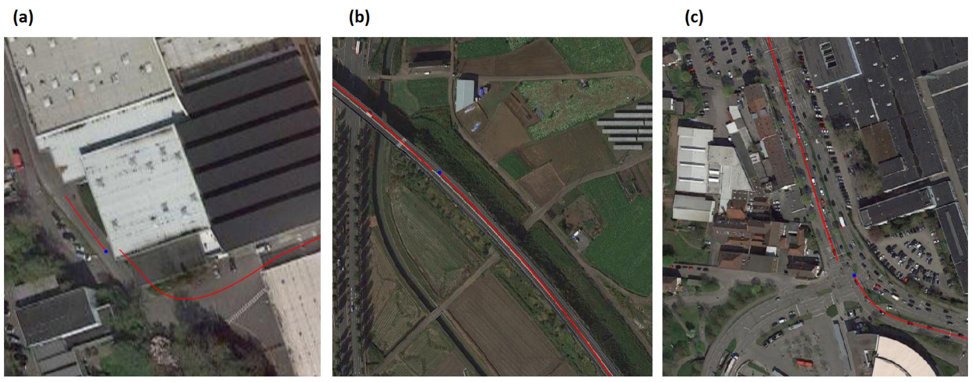

Figure 6.9: Path approximations ending up short of their destination. Map data: Google, ZENRIN, GeoBasis-DE/BKG.

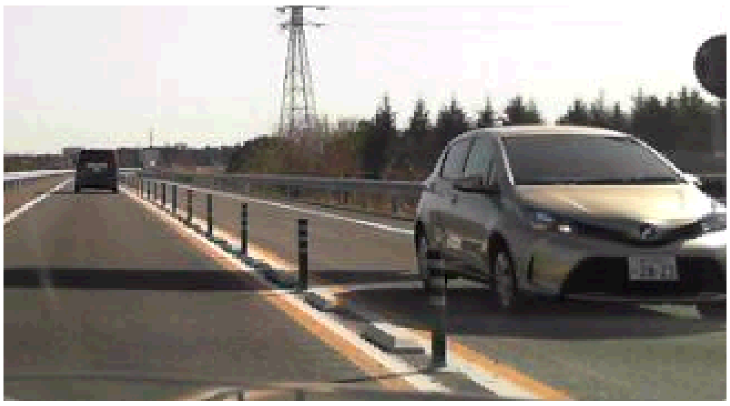

Figure 6.10: Poles and stone blocks separating opposite sides of traffic. Image: Valeo.

Grid-based sampling resamples the whole scan point cloud to achieve an approximately equally dense distribution across all detected obstacles. The resulting path approximations for *urban* and *factory* are decidedly better compared to *alpha*, both in average MSE and in standard deviation. The only exception is the *country* trace, for which the path estimates are now almost as bad as those based only on odometry signals. That is because on the road in that trace, opposite sides of traffic are separated by groups of single plastic poles followed by two small stone blocks every five meters. Figure 6.10 shows these dividers. Depending on vertical angle to the sensor, the stone bars can generate detections at different points along their long side. After resampling, these detections make up as much of the scan point cloud as each of the guardrails. Placing too much weight on these inconsistent detections lessens the approximation quality of path estimates.

Sampling based on surface smoothness places more weight on un-smooth scan points, such as those connecting two surfaces or those on the edge of a single line of scan points. This produced path approximations much worse than *alpha*, almost as bad as the paths reconstructed from only odometry data. That is because this strategy bears several problems. Surfaces that extend past a sensor's field of view still generate edge point detections because these sections look identical to sections with actual edge points. Surfaces that are partially hidden by a separate obstacle between the sensor and the surface will also generate edge points even though the partially hidden surface may extend past the detected edge point, as mentioned in [ZS14].

Sampling against dynamic obstacles is a simple strategy we implemented to lessen the impact of dynamic obstacles. An area shaped like a rhombus centered on the host is used to classify objects as dynamic (figure 5.6). Even though dynamic obstacles are most prevalent in *urban*, all three traces now produce slightly better ϕ MSEs. For *country*, where this is especially noticeable, a possible reason is that the obstacles shown in figure 6.10 now have less impact on the weighting function; this allows farther out scan points to influence ϕ estimations more, similar to what is done in section 5.6.2 (sampling for far detections). For *factory*, this strategy has a similar effect as sampling based on detection distance with the difference that walls are now only affected right next to the vehicle where the rhombus overlaps, leaving most of the walls within the 20 meter range untouched.

The MSEs for dxy, remaining largely the same on average, have become less stable for *country* and *factory*. That is likely because in those traces, only scan points belonging to static obstacles have been removed. Only for *urban* has dxy become more stable: *urban* is the only trace with frequent dynamic obstacles, exactly what this strategy was aiming at.

6.7 Combining Scan Point Sampling Strategies

The obvious next step is to combine our scan point sampling strategies. Table 6.4 shows the results discussed in this section. For comparison, the odometry-only path and the configured particle filter from section 6.5 are again listed as *odom.* and *alpha*, respectively. Additionally, the best result obtained for each trace so far is listed as *best*. Three configurations are compared; one based on the overall performance of scan point sampling strategies, and two that consider varying results for different traces.

We start with *combined1*. For this setup we included the four strategies that seemed most promising, omitting only the two strategies that produced worse results (*angle* and *smooth* in table 6.3). Apart from some exceptions addressed in the other setups discussed later on, these four strategies improved results for all traces. The sampling steps given by the four strategies were executed in the following order: First, grid-based sampling is applied. Applying it later would work against other strategies which specifically

	Factory		**Country**		**Urban**	
MSE	ϕ	dxy	ϕ	dxy	ϕ	dxy
Odom.	30.37	109.28	26.28	169.95	27.10	205.11
Alpha	5.86 (3.29)	7.07 (1.54)	1.93 (0.10)	41.05 (1.16)	10.95 (1.46)	28.97 (9.70)
Best	3.64 (1.14)	5.66 (0.70)	1.40 (0.24)	36.55 (4.05)	3.98 (0.96)	15.80 (3.02)
combine1	7.20 (3.92)	5.35 (0.72)	10.31 (0.85)	75.21 (6.28)	0.72 (0.20)	23.47 (1.40)
combine2	1.86 (0.26)	3.80 (0.54)	–	–	–	–
combine3	–	–	1.02 (0.10)	23.14 (1.96)	–	–

Table 6.4: Comparison of MSEs for combined scan point sampling strategies

affect the scan point cloud to be denser in some and sparser in other areas. The order within the other three approaches doesn't matter as they each look at every scan point separately.

Indeed, our scan point sampling approach further improved results for *urban* when combining multiple strategies. Most importantly, MSEs for ϕ have almost sunk to within the evaluation error margins; dxy MSEs have only gotten slightly worse. For *factory*, results are worse than the previous best results. That is because *combine1* includes distance-based sampling, which had previously produced results worse than that of *Alpha*. We address this in *combine2*, which removes distance-based sampling. After this change, *factory* results are now also better than its previous best results: ϕ MSEs have halved and dxy MSEs have gone down by a third.

To remedy *combine1*'s performance for *country*, we remove grid-based sampling from *combine1*, and replace it with angle-based sampling (section 5.6.3) as that strategy actually had a positive impact on this trace. The position of angle-based sampling in the order listed above is irrelevant because this strategy only considers surface orientation and that isn't affected by any of the other three strategies used in this setup. We execute angle-based sampling first because it typically reduces the scan point cloud size drastically, allowing for faster calculations in the following sampling steps. With this setup, listed as *combine3* in table 6.4, the errors for *country* decrease to below those of the best results obtained in the previous chapter: the dxy MSE has gone down by a third and the ϕ MSE is now almost down to 1.

Figure 6.11 shows paths provided by these final three setups compared to the respective paths reconstructed from odometry data.

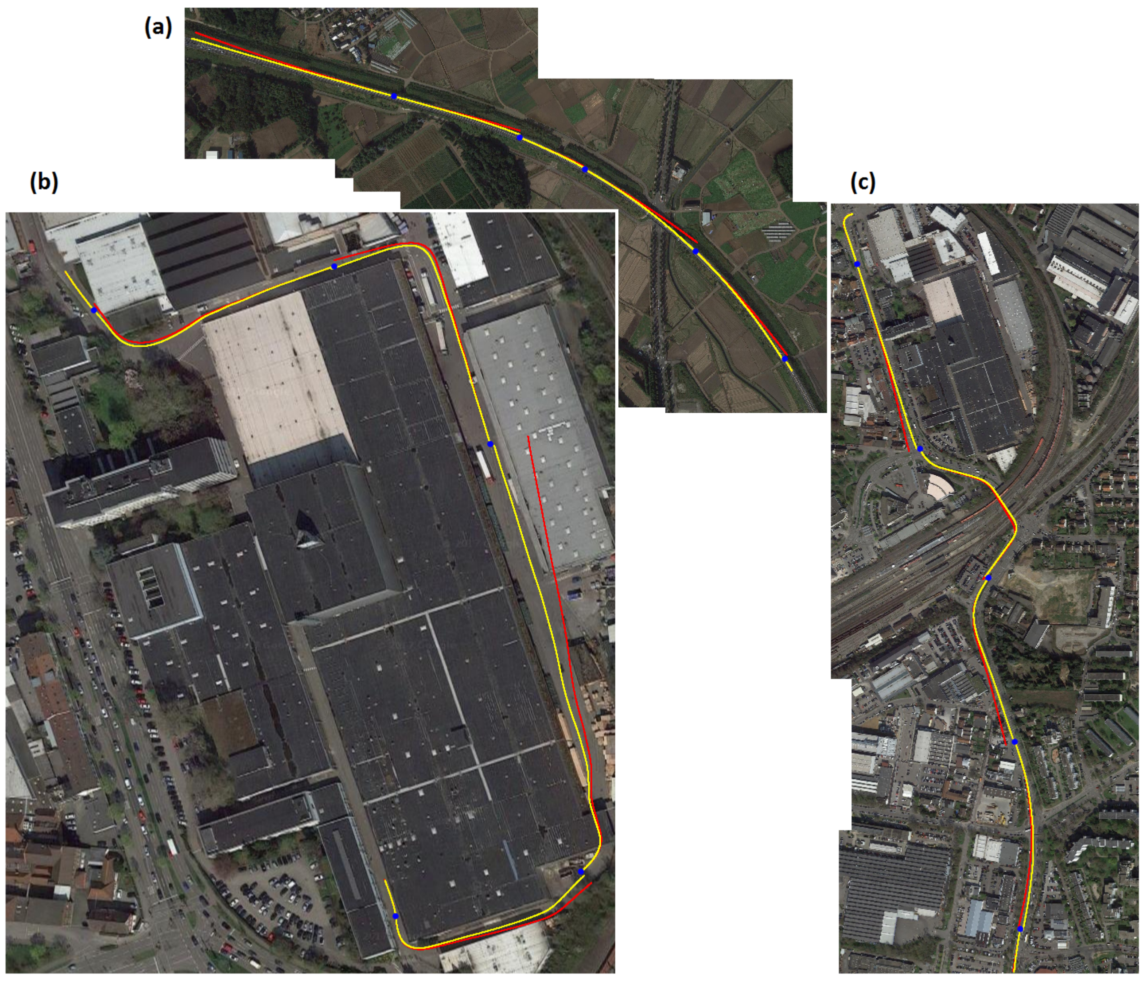

Figure 6.11: Path approximations provided by combined scan point sampling strategies (yellow) compared with odometry-only reconstruction (red): (a) *combine3*, (b) *combine2*, (c) *combine1*. Map data: Google, ZENRIN, GeoBasis-DE/BKG.

7 Conclusion

This work tackles automotive SLAM using a Monte Carlo particle filter handling scan-to-map matching on an existing occupancy grid map; alpha filters are used to handle odometry estimation. Two particle weighting functions are considered for the particle filter and a series of scan point sampling strategies that provide differently distributed scan point clouds for the calculation of particles' scan matching weights. Of particular interest were five sampling strategies that linearly downsampled detections at close range, supersampled very far detections, sampled based on weights determined by surface normals, resampled to approximate uniform scan point density and sampled against dynamic obstacles. Manually matching scan point clouds to satellite imagery yielded precise ground truth data enabling a comprehensive evaluation of path approximations.

Comparing path approximations against paths reconstructed from odometry signals showed that, accumulated over sections of 60 to 200 meters, the MSE in heading angle was reduced by 93-97% and the MSE in position was reduced by 86-96% through the use of different combinations of scan point sampling strategies. Evaluations in different scenarios demonstrated that no combination of scan point sampling strategies would work well in all scenarios. The particle filter was successfully tested in a factory setting, a complex urban road, and a two-way elevated highway. In a complex highway scenario our approach didn't succeed due to a number of difficulties; inconsistent scan point clouds generated by uneven embankment surfaces, vehicles blocking the sensors' views and an insufficient number of obstacle detections usable as a reference for localization posed difficulties to the particle filter.

7.1 Future Work

Several parts of this work naturally lead the way to some approaches that promise further approximation improvements.

Scan point sampling was investigated to find criteria for constructing scan point distributions yielding particle weights more representative of the particles' approximation quality; however runtime optimizations were not considered. SLAM typically is performed in real time so performance should be good enough to keep up with the stream of odometry and lidar measurements. Instead of changing the distribution of scan points directly, one could assign them weights to use as exponents in the calculation of particle scores. This avoids expensive duplicate lookups into the map and construction of a second scan point cloud.

Scan point classification could generally tie directly into the particle weighting function, allowing one to discard ground detections, dynamic obstacles, shrubbery and foliage, and other clutter detections. This would likely require knowledge beyond single scan point clouds and the current occupancy grid map state to properly assess scan point properties.

Scenario classification would allow one to exploit different characteristics of scan point sampling strategies in different scenarios. Dynamically using different parameters and weighting the impact of the individual scan point sampling strategies based on what type of scenario the vehicle is currently navigating would eliminate the problem of having to combine scan point sampling strategies into one single approach suitable for many different scenarios. Classifications could define scenario classes or they could provide information about scenario attributes such as amount of dynamic objects, road type, speed of traffic, and composition of obstacle detections.

Sensor mounting positions can become misaligned over the lifespan of a car. Using the particle filter to correct sensor mounting data would allow longer service intervals and could provide well-aligned scan point clouds even when sensors become misaligned.

List of Figures

Bibliography

[AA12] A. Azim, O. Aycard. “Detection, classification and tracking of moving objects in a 3D environment.” In: *Intelligent Vehicles Symposium (IV), 2012 IEEE*. IEEE. 2012, pp. 802–807 (cit. on p. 57).

[BD06] T. Bailey, H. Durrant-Whyte. “Simultaneous localization and mapping (SLAM): Part II.” In: *IEEE Robotics & Automation Magazine* 13.3 (2006), pp. 108–117 (cit. on p. 11).

[BF08] A. Barth, U. Franke. “Where will the oncoming vehicle be the next second?” In: *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE. 2008, pp. 1068–1073 (cit. on p. 31).

[Cha04] D. M. Chabukswar. “Modeling Simulation and Experimental Validation of ’ATRV-Jr.’” MA thesis. 2004, pp. 12–13. URL: http://diginole.lib.fsu.edu/islandora/object/fsu:182176/datastream/PDF/view (cit. on p. 31).

[CMC+17] A. Cosgun, L. Ma, J. Chiu, J. Huang, M. Demir, A. M. Anon, T. Lian, H. Tafish, S. Al-Stouhi. “Towards Full Automated Drive in Urban Environments: A Demonstration in GoMentum Station, California.” In: *arXiv preprint arXiv:1705.01187* (2017) (cit. on p. 11).

[CQB+13] Z. Chong, B. Qin, T. Bandyopadhyay, M. H. Ang, E. Frazzoli, D. Rus. “Synthetic 2d lidar for precise vehicle localization in 3d urban environment.” In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 1554–1559 (cit. on p. 11).

[DB06] H. Durrant-Whyte, T. Bailey. “Simultaneous localization and mapping: part I.” In: *IEEE robotics & automation magazine* 13.2 (2006), pp. 99–110. URL: https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/SLAMTutorial1.pdf (cit. on pp. 11, 14).

[DFBT99] F. Dellaert, D. Fox, W. Burgard, S. Thrun. “Monte carlo localization for mobile robots.” In: *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*. Vol. 2. IEEE. 1999, pp. 1322–1328 (cit. on pp. 11, 17).

[EKS+96] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231 (cit. on pp. 24, 25).

[Fox02] D. Fox. “KLD-sampling: Adaptive particle filters.” In: *Advances in neural information processing systems*. 2002, pp. 713–720 (cit. on p. 18).

[GG14] H. Gotzig, G. Geduld. “Automotive LIDAR.” In: *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort* (2014), pp. 1–20 (cit. on p. 23).

[HR16] M. Hammarsten, V. Runemalm. “3D Localization and Mapping using automotive radar.” MA thesis. Chalmers University of Technology, 2016. URL: http://publications.lib.chalmers.se/records/fulltext/238490/238490.pdf (cit. on pp. 20, 49).

[Hun07] J. D. Hunter. “Matplotlib: A 2D graphics environment.” In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95 (cit. on p. 39).

[JOP+01] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001. URL: http://www.scipy.org/ (cit. on p. 39).

[JZ98] R. J. Jesionowski, P. Zarchan. “Comparison of four filtering options for a radar tracking problem.” In: *Journal of guidance, control, and dynamics* 21.4 (1998) (cit. on p. 20).

[Kal+60] R. E. Kalman et al. “A new approach to linear filtering and prediction problems.” In: *Journal of basic Engineering* 82.1 (1960), pp. 35–45 (cit. on p. 20).

[LJ00] X. R. Li, V. P. Jilkov. “A survey of maneuvering target tracking: Dynamic models.” In: *Proceedings of SPIE Conference on signal and data processing of small targets*. Vol. 6. 4048. 2000, pp. 212–235 (cit. on p. 31).

[LMT07] J. Levinson, M. Montemerlo, S. Thrun. “Map-Based Precision Vehicle Localization in Urban Environments.” In: *Robotics: Science and Systems*. Vol. 4. 2007, p. 1 (cit. on p. 11).

[LR09] T. Laue, T. Röfer. “Pose Extraction from Sample Sets in Robot Self-Localization-A Comparison and a Novel Approach.” In: *ECMR*. 2009, pp. 283–288 (cit. on p. 46).

[LT10] J. Levinson, S. Thrun. “Robust vehicle localization in urban environments using probabilistic maps.” In: IEEE. 2010, pp. 4372–4378 (cit. on p. 11).

[LVL14] S. Lefèvre, D. Vasquez, C. Laugier. “A survey on motion prediction and risk assessment for intelligent vehicles.” In: *Robomech Journal* 1.1 (2014), p. 1 (cit. on p. 31).

[MDU11] P. Morton, B. Douillard, J. Underwood. “An evaluation of dynamic object tracking with 3d lidar.” In: *Proc. of the Australasian Conference on Robotics & Automation (ACRA)*. 2011 (cit. on p. 57).

[MON09] T. Miyasaka, Y. Ohama, Y. Ninomiya. “Ego-motion estimation and moving object tracking using multi-layer lidar.” In: *Intelligent Vehicles Symposium, 2009 IEEE*. IEEE. 2009, pp. 151–156 (cit. on p. 57).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. “Scikit-learn: Machine learning in Python.” In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830 (cit. on pp. 25, 39).

[RHG14] M. Roth, G. Hendeby, F. Gustafsson. “EKF/UKF maneuvering target tracking using coordinated turn models with polar/Cartesian velocity.” In: *Information Fusion (FUSION), 2014 17th International Conference on*. IEEE. 2014, pp. 1–8 (cit. on p. 41).

[RJMZ16] J. Rohde, I. Jatzkowski, H. Mielenz, J. M. Zöllner. “Vehicle pose estimation in cluttered urban environments using multilayer adaptive Monte Carlo localization.” In: *Information Fusion (FUSION), 2016 19th International conference on*. IEEE. 2016, pp. 1774–1779 (cit. on p. 11).

[SC86] R. C. Smith, P. Cheeseman. “On the representation and estimation of spatial uncertainty.” In: *The international journal of Robotic Research* 5.4 (1986), pp. 56–68 (cit. on p. 11).

[SRW08] R. Schubert, E. Richter, G. Wanielik. “Comparison and evaluation of advanced motion models for vehicle tracking.” In: *Information Fusion, 2008 11th International Conference on*. IEEE. 2008, pp. 1–6 (cit. on p. 31).

[TBB+96] S. Thrun, A. Buecken, W. Burgard, D. Fox, A. B. Wolfram, B. D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, et al. *Map learning and high-speed navigation in RHINO*. 1996 (cit. on p. 22).

[TBF05] S. Thrun, W. Burgard, D. Fox. *Probabilistic Robotics*. MIT Press, 2005 (cit. on pp. 13, 15, 17, 19, 22, 44).

[Thr03] S. Thrun. “Learning occupancy grid maps with forward sensor models.” In: *Autonomous robots* 15.2 (2003), pp. 111–127 (cit. on p. 23).

[UT13] C. Ulas, H. Temeltas. “A Fast and Robust Feature-Based Scan-Matching Method in 3D SLAM and the Effect of Sampling Strategies.” In: *International Journal of Advanced Robotic Systems* 10.11 (2013), p. 396. DOI: 10.5772/56964. eprint: http://dx.doi.org/10.5772/56964. URL: http://dx.doi.org/10.5772/56964 (cit. on pp. 50, 55).

[VAA07] T.-D. Vu, O. Aycard, N. Appenrodt. “Online localization and mapping with moving object tracking in dynamic outdoor environments.” In: *Intelligent Vehicles Symposium, 2007 IEEE*. IEEE. 2007, pp. 190–195 (cit. on p. 11).

[WCV11] S. v. d. Walt, S. C. Colbert, G. Varoquaux. “The NumPy array: a structure for efficient numerical computation.” In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30 (cit. on p. 39).

[WPN15] D. Z. Wang, I. Posner, P. Newman. “Model-free detection and tracking of dynamic objects with 2D lidar.” In: *The International Journal of Robotics Research* 34.7 (2015), pp. 1039–1063 (cit. on p. 57).

[Wu16] Y. Wu. “Image Based Camera Localization: an Overview.” In: *arXiv preprint arXiv:1610.03660* (2016) (cit. on p. 11).

[ZS14] J. Zhang, S. Singh. “LOAM: Lidar Odometry and Mapping in Real-time.” In: *Robotics: Science and Systems*. Vol. 2. 2014 (cit. on pp. 56, 77).

All links were last followed on July 31, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature