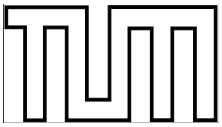
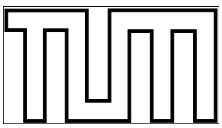


Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

SLAM-Based Navigation of An Autonomous Driving Car

Felix Feik



Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

SLAM-basierte Navigation eines autonom fahrenden Autos

SLAM-based navigation of an autonomous driving car

Author:	Felix Feik
Supervisor:	Prof. Dr. Ing-habil Alois Knoll
Advisor:	Biao Hu
Submission:	February 15, 2017

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

_____ _____

Place, Date Signature

Abstract

In this thesis, an autonomous driving model car is presented, using LiDAR sensor data and a 2D map to get the shortest path from a designated point to its destination.

First of all the whole area in which the car is to drive, is scanned with a LiDAR sensor. A 2D map with all obstacles is created, based on the sensor data. Afterwards the starting point of the car can be entered on this map and a destination where it should drive to can be set. The car will find the shortest path from the starting point to its destination, without crashing into any obstacles. On its way to the destination the LiDAR sensor is constantly scanning the local environment, so the car will also be able to detect uncharted obstacles and will replan the path, if needed.

The creation of the map and the route planning is done on a Laptop and the LiDAR sensor data is scanned by a RaspberryPi board.

Contents

List of Abbreviations

AMCL	Adaptive Monte Carlo Localization
CPU	Computer Processing Unit
D	Dimensional
EKF	Extended Kalman Filter
GUI	Graphical User Interface
LiDAR	Light Detection and Ranging
ROS	Robot Operating System
SCM	Software Configuration Management
SLAM	Simultaneous Localization and Mapping
TUM	Technical University Munich
XML	Extensible Markup Language

List of Figures

1. Introduction

1.1. Motivation

In the year 1970 the number of people who died in traffic accidents was the highest in Germany ever. Since that time the death rate strongly decreased, although the number of road users intensively increased. On the one hand that's because of the higher penalties for road delicts like a violation of the speed limit, but on the other hand, because of the constantly increasing safety of the automobiles.[1] Nowadays a car is a rolling computer, full of assistance systems to aid the driver and increase safety. For example a *Lane Keeping Assist* to hold the car's lane or an *Active Distance Control* to follow a car which is driving in front, with an optimal safety distance, and also an *Emergency Brake Assist* to rapidly stop the car in an emergency situation. A lot of these systems are only half automated systems, so they still need the driver to implement them and to do their jobs.

The next step is to develop a fully automated system, driving a car autonomously and replace the driver completely. This would of course greatly decrease the accident rate on the basis of human driving errors. The road safety is the most important criteria of autonomous driving, but there are also a few more. The traffic flow will also be optimized and time-efficient driving will become standard. Less traffic jams and thereby the reduction of fuel consumption and CO2 emissions are positive consequences in times of global warming problems. It is also an aid for disabled and old people, who aren't able to drive by themselves, but have to be mobile.[2]

1.2. Related Work

ROS "A flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms."[3] This operating system is the basis of the autonomous driving car. It runs on the Laptop as a full desktop version and on the raspberry pi as an embedded version.

urg_node The *urg_node* is the ROS driver library for the Hokuyo LiDAR sensor used on the car to scan the environment. This enables using the LiDAR as an ROS node together with the other tools needed for the project.[4]

hector_slam This is a package including the *hector_mapping* node to create a 2D map based on the "high update rate of modern LIDAR systems" developed for ROS.[5] It is used to create the 2D map of the environment in which the car

should drive and which is used to plan the shortest route from the designated point to the destination of the autonomous car.

hector_slam_example It is a bunch of launch files and configuration files to use the LiDAR sensor to create a SLAM map. Using these files makes it more easier and faster to create the map.[6]

navigation_stack The *navigation stack* is a library that includes all important functions to navigate a ROS running robot through a 2D map with the help of sensor data. "It needs to be configured for the shape and dynamics of a robot to perform at a high level". The library contains the *global_planner* to find the shortest route from a starting point to a destination, the *move_base* package is designated to give certain velocity commands for the robot's motor management based on the route and *amcl* localizes the car on the 2D map using the LiDAR sensor data.[7]

teb_local_planner "The *teb_local_planner* package implements a plugin to the *base_local_planner* of the 2D *navigation stack*."[8] It enables finding a local route based on the LiDAR sensor data to avoid obstacles and create velocity commands optimized for a car-like robot with Ackermann steering.[9] The *teb_local_planner* is responsible for replanning the original route of the car, should the LiDAR sensor detect some uncharted obstacles while the car is moving.

1.3. Problem Statement

The aim of this work is to demonstrate the development an autonomous driving car based on a SLAM map with the help of a LiDAR sensor. However, the goal contains some mechanical and software problems.

On the one hand all components have to be well placed in the car and they need to be connected correctly. All cables should run through the car without hanging on the ground or blocking the wheels and all the contacts need to be well insulated.

On the other hand the car requires some software to control the whole movement correctly. The first requirement is a software to create the 2D SLAM map based on the LiDAR sensor data. After that a software is needed that is able to find the shortest route on the global map and one to spontaneously avoid local obstacles. Furthermore the software has to connect a management unit in the car with a laptop via Wifi to exchange all important data. The LiDAR sensor has to be on top of the car, but the route finding algorithm and the map of the environment runs on a laptop because of the required CPU performance.

1.4. Structure of this thesis

The next chapter, chapter 2, includes some theoretical background information about the basic SLAM process and the hardware what is required to create a SLAM map.

In chapter 3 the implementation of the whole software and hardware used in this thesis is presented. It includes the different methods and its theory and functionality is illustrated.
In the last chapter, chapter 4, the results of the implemented method in this thesis are shown and evaluated.

2. Background

2.1. About SLAM

The term SLAM is as stated an acronym for Simultaneous Localization And Mapping. It was primarily developed by Hugh Durrant-Whyte and John J. Leonard[10] based on earlier work by Smith, Self and Cheeseman[11]. Durrant-Whyte and Leonard initially termed it SMAL but it was later changed to give a better impact. SLAM is concerned with the problem of building a map of an unknown area by a mobile robot while at the same time navigating through the environment using the map. So the robot starts anywhere in the environment and needs to know anytime where it currently is relative to the recognized obstacle.

SLAM can be implemented in lots of ways, with a huge amount of hardware that can be used. SLAM is more than just one algorithm, it is more like a whole concept to fix a few problem. It consists multiple parts like "Landmark extraction, data association, state estimation, state update and landmark update“[12] Every of these small parts of the concept can be solved by a lot’s of ways. It depends on the usage, like what robot will be used. In which area will the robot be driving? Is it a hall with only static objects or is it an urban crowded street with lots of movement during the building of the map? Do the robot need a 2D or a 3D vision? Questions like this needs to be answered before starting to implement the SLAM process. They all influence what hardware devices and algorithms used to implement a well working SLAM concept. In the case of this thesis, the autonomous driving car, only a 2D motion is considered. The area in which the car should drive is exclusively an indoor environment with static objects.

2.1.1. Required hardware

As important as the choice of good algorithms is the selection of the right hardware. To do SLAM a mobile robot and a range measurement device is required.

A robot using SLAM and 2D motion needs to be mobile and “should not have an error of more than 2 cm per meter moved and 2° per 45° degrees turned“. [12] In an indoor environment it’s normally a wheel-based robot like the car used for this thesis. The robot also needs a proper working motor management to control the car in the area correctly.

To scan the area different devices can be used. In these days normally a LiDAR sensor or a camera is used. The reason why most of the people doing SLAM using the LiDAR as measurement device is, that they are very accurate and the data output does not need a high computing power. A bad thing of the

laser sensors is, that they aren't able to measure underwater or don't recognize some surfaces like glass as an obstacle.[12]

(a) UTM-30LX-EW LiDAR sensor (b) RPLIDAR - 360 degree Laser Scanner

Figure 2.1: *Different LiDAR sensors.* (a) shows the LiDAR sensor used in the car presented in this thesis [13] (b) shows a 360 degree LiDAR sensor [14]

Another option is to use a camera as measurement device. It's also very accurate, but it needs lots of computing power to handle the big input data. Also the vision of a camera only works with a well lighted environment, so it is more prone to errors or certainly does not work anymore if it is used in a dark room or at night. The camera is better to use for 3D map, because of its 3D vision. A cheap choice to use is for example the Microsoft Kinect camera. The best result can be obtained by the combination of both sensor data.[12]

Figure 2.2: *Microsoft Kinect sensor.* The camera records RGB images and includes a depth sensor.[15]

2.1.2. The SLAM process

The aim of SLAM is to create a map of the area in which the robot is moving around. The basic SLAM process is done by a number of steps. All these steps can be done in different ways. The following graphic shows an example of such a SLAM process with use of the extended Kalman filter. The EKF is one of the first probabilistic SLAM algorithms. It is often picked to explained how the SLAM process is working, but in reality also some other algorithms can be used.

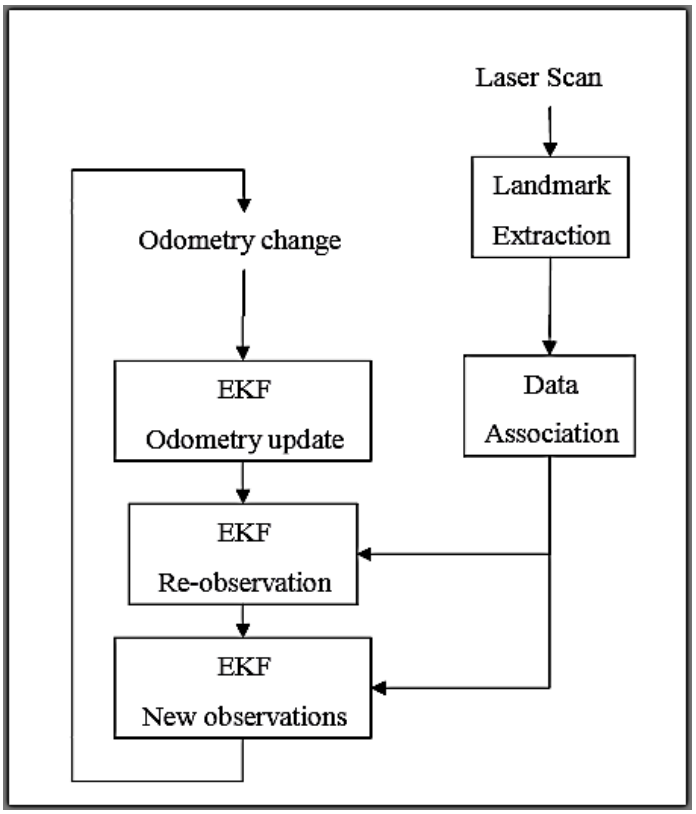


Figure 2.3: *Overview of a typical SLAM process.* The Graph shows the basic process of SLAM done with the EKF algorithm.[12]

The first step is to scan the environment of the initial pose with the measurement device and in this case gather the LiDAR sensor data. This data contains the distance how far a so called landmark is away and also the angle where it is located, based on the initial position of the robot. These landmarks are various points like obstacles in the local environment and are combined with the distances to the robot and their angles most the input of the EKF.

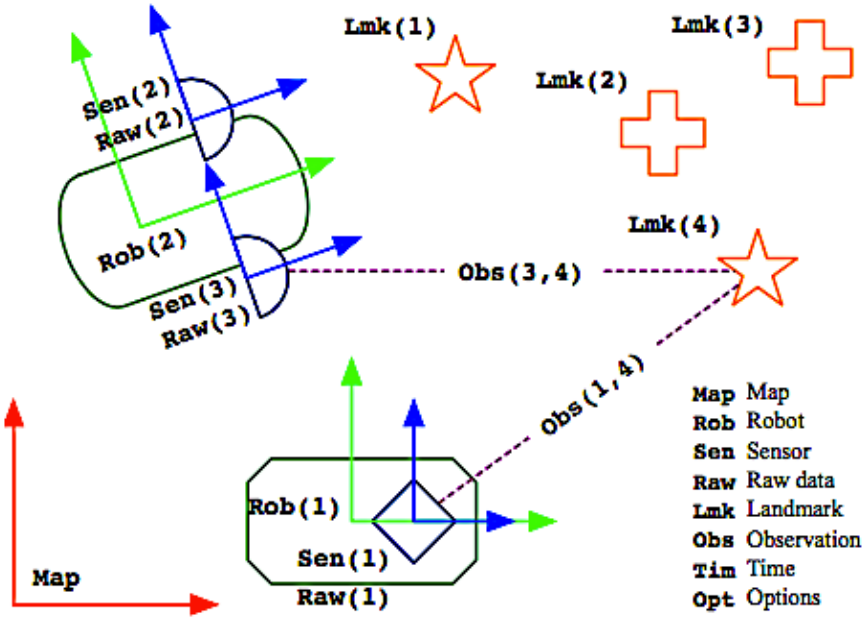


Figure 2.4: *Typical SLAM entities.*[16] The picture gives a good example how the sensor information is filled with an angle and a distance to an obstacle. Robot(1) detects landmark(4) at the initial pose, then it moves on to robot position (3) and it still detects the landmark (4) but with another distance and angle.

The EKF is an algorithm to estimate the robots position is at the moment on the map and update its odeometry if needed. The odeometry data of the robot contains the position of the robot on the map. This data needs to be very accurate to build the map correctly. The map is a large state vector stacking robot and landmark states, where **R** is the robot and **M** is the set of landmark states. The EKF algorithm models the map by a Gaussian variable using the mean and the covariances matrix of the state vector, denoted by **x** and **P**. The aim of SLAM and EKF is to keep this map up to date at all time. [16]

$$x = \begin{bmatrix} R \\ M \end{bmatrix} = \begin{bmatrix} R \\ L_1 \\ \vdots \\ L_n \end{bmatrix} \qquad P = \begin{bmatrix} P_{RR} & P_{RM} \\ P_{MR} & P_{MM} \end{bmatrix} = \begin{bmatrix} P_{RR} & P_{RL_1} & \dots & P_{RL_n} \\ P_{L_1R} & P_{L_1L_1} & \dots & P_{L_1L_n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{L_nR} & P_{L_nL_1} & \dots & P_{L_nL_n} \end{bmatrix} \tag{1}$$

At the beginning the map starts without any landmarks and the initial robot pose is set to the origin of the map, so **n** = 0 and x = R.

$$x = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2}$$

As soon as the robot starts moving around its odeometry will change. The robot motion is based on a generic time-update function. Based on the state vector **x**, the control vector **u** and the perturbation vector **n**.

$$x \leftarrow f(x, u, n) \tag{3}$$

During the movement and on every new position, the LiDAR collects new data of the environment. The significant landmarks get extracted again and they get associated to the landmarks the robot previously has seen. Based on the re-observated points the robot can update its new position in the EKF. This is possible because the robot always has a relative distance and angle from its current position.[12] The observation is based on the generic observation function where **y** is the noisy measurement, **x** is the full state, **h()** is the observation function and **v** is the measurement noise. [16]

$$y = h(x) + v \tag{4}$$

3. Implementation

This chapter is about the implementation of the autonomous driving car. It shows what hardware is used and what algorithms are used to produce a functional software to get the car working.

3.1. Used Hardware

3.1.1. The car

The basic of the whole project is of course the car. It is an electric driving car, constructed with metal and it needs an input voltage of 12 volts. The wheelbase of the car is 55 centimeters, the whole length is 70 centimeters and the width is 55 centimeters. Its maximum speed is up to 80 km/, but for this project a maximum velocity of 0,55 m/s is used. It is a good speed to operate safe in an indoor area, without damaging any obstacles or persons in case of testing failures.

(a) Front of the car (b) Back of the car

Figure 3.1: *Autonomous driving car*. The picture shows in (a) the front of the car used in this thesis. And (b) shows it from the back.

Like you can see in the pictures the car contains a lots of sensors and other electronically devices. As motor management the car has a small Arduino board in it to feed the engine with the required values. It is able to send a PWM signal in percentage to the engine as a speed value and a degree value to control the steering angle of the car. It is possible to do forward and backward driving. The input values are sent by a RaspberryPi board via serial USB.

The RaspberryPi board is also planted in the car. It is running Ubuntu Mate 16.04 as operating system and a ROS distribution is installed. Its job is to communicate via Wifi with the ROS Master, which is running on a Laptop. The RaspberryPi board receives the required data, like the steering angles and the

speed values, from the master. These data is calculated by the Laptop and the RaspberryPi board forwards them to the Arduino board.

3.1.2. The laptop

The control basis of the whole project is a customary Sony Vaio laptop. It is running Linux 14.04 Indigo as operating system and a ROS distribution is installed too. An Intel Pentium dual core processor is used to execute the route planning algorithm, present the map and compute the localization needed for the autonomous driving car. The laptop has all the required tools for the project installed and the ROS master is running on this machine. The data of all so called topics like the scan data of the LiDAR and the velocity commands are published to the ROS master. The whole visualization is done by it too. You can set the starting point of the car and you can set a goal on the created SLAM map.

3.1.3. The measurement device

The whole measurement is done by a single LiDAR sensor, the Hokuyo UTM-30LX-EW (Fig. 2.1(a)). It is placed in front of the car(Fig. 3.1(a)) and scans the environment with a radius of 270 degree. It has an accurate detection range from 0.1 to 30 meters with a scanning rate of 25 millisec/scan. The angular resolution is 0.25 degree.

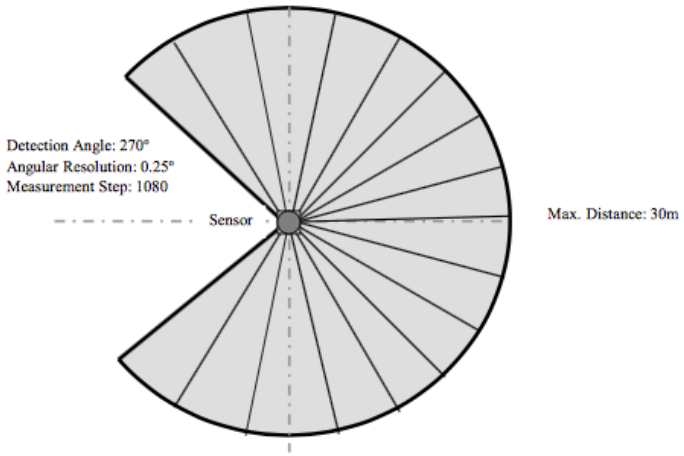


Figure 3.2: *Diagram of scanned area[17]*. It shows how the scanning area of the used LiDAR looks like.

The whole data is transferred via ethernet to the destination device. The output datatype is an array including the distance values detected from 0 to 270 degree around the LiDAR. So the LiDAR publishes every 25 milliseconds an array of 270 * 4 = 1080 double values. It is connected directly to the Laptop.[17]

3.2. Software

3.2.1. Operating system

ROS is the operating system of the autonomous driving car and it's the basis of the whole project. It is installed on the Laptop and on the RaspberryPi. The ROS system needs a master, where all devices are registered and where they can publish their data and receive new one from other nodes.

In this case this master is running on the laptop and both devices are connected via Wifi in an ad-hoc network to reach the master. The master can be started on every free port, in this case the port 44420 is used. The Laptop needs to publish the calculated velocity commands, so the RaspberryPi can forward them to the Arduino board. The LiDAR sensor publishes its scanned data. All algorithms and tools used in this project are based on the ROS system.

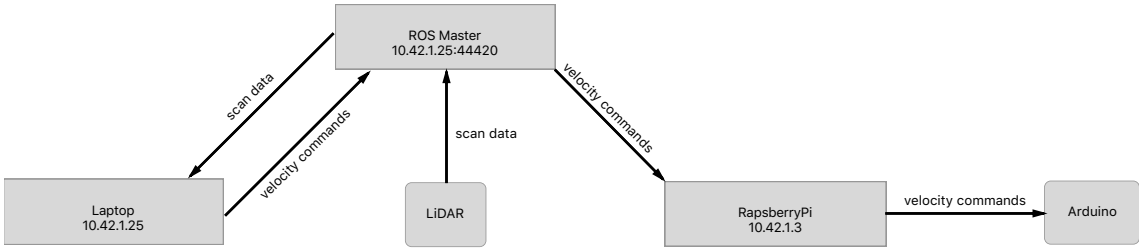


Figure 3.3: *Basic relation model of the hardware*. This graph shows the relation between the different hardware used for the car.

3.2.2. Mapping

A SLAM based navigation requires a SLAM map. It is the basis of the navigation part. It has to be a detailed map, so the localization algorithm will work properly. The maps used in this thesis are all created by the *hector_slam* library.

This algorithm uses the LiDAR sensor data to create a map of the whole area, in which the car should drive afterwards. The approach of this algorithm is to do the so called FastSLAM. The whole area is represented as an occupancy 2D grid map. Because of the high update rate of the LiDAR sensor it is possible to use only approximative data. The scanned endpoint data of the sensor is converted to a point cloud using the estimated platform orientation and joint values. As scan matching algorithm, only filtering based on the endpoint z coordinates is enough, so „only endpoints within a threshold of the intended scan plane are used in the scan matching process“[18].

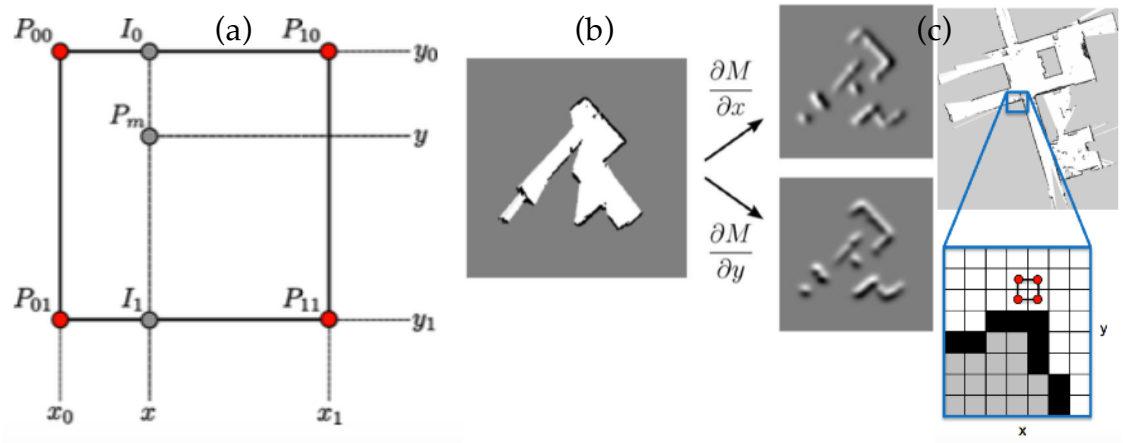


Figure 3.4: *Hector_SLAM filter method.* „(a) Bilinear filtering of the occupancy grid map. Point P_m is the point whose value shall be interpolated. (b) Occupancy grid map and spatial derivatives.“[18] (c) A small zoom of the whole grid map.

The pose estimation of the robot is done by a simple iteration. First project the endpoint onto map based on the current pose estimation. Next, estimate map occupancy probability gradients at the scan endpoint. And last but not least perform a Gauss-Newton iteration to refine the pose estimation. [18]

$$H = \left[\nabla M(S_i(\xi))\frac{\partial S_i(\xi)}{\partial \xi}\right]^T = \left[\nabla M(S_i(\xi))\frac{\partial S_i(\xi)}{\partial \xi}\right]^T \tag{5}$$

Based on the pose estimation of the robot(5) and the sensor data it is possible to build a well defined map with all obstacles contained in the area.

3.2.3. Navigation

The first step was to create the SLAM map as basis of the navigation part of this thesis. Next step is the navigation itself. The navigation isn't easy to do, because on the one hand the car has to find a valid global plan from its estimated starting point on the map to its destination. On the other hand the car has to react to its local environment. For example if an uncharted obstacle appears or the car has to drive through a small gap. That's why the navigation part contains two different planner. One is the so called local planner and one is the global planner.

The global route is calculated by the *global_planner* of the *navigation_stack* library in ROS. It is done by the Dijkstra algorithm. It's an algorithm to find the shortest path from a node A to B in a graph. The idea is basically really intuitive. At the beginning take a whole graph and set one node as starting point and set the distance to all others to ∞. Now the algorithm contains two queues, one

with all the visited nodes which is of course empty at the initialization and one with all other nodes expect the starting node. As long as the queue with the unvisited nodes isn't empty select the one with the minimum distance, mark it as visited and check if a new shortest path is found. If there is one available set it as the new value of the shortest path.

```
dist[s] ←o                          (distance to source vertex is zero)
for all v ∈ V–{s}
    do dist[v] ←∞                   (set all other distances to infinity)
S←∅                                 (S, the set of visited vertices is initially empty)
Q←V                                 (Q, the queue initially contains all vertices)
while Q ≠∅                          (while the queue is not empty)
do  u ← mindistance(Q,dist)         (select the element of Q with the min. distance)
   S←S∪{u}                          (add u to list of visited vertices)
    for all v ∈ neighbors[u]
        do if  dist[v] > dist[u] + w(u, v)      (if new shortest path found)
           then    d[v] ←d[u] + w(u, v)         (set new value of shortest path)
                                                (if desired, add traceback code)
return dist
```

Figure 3.5: *Pseudocode Dijkstra algorithm.* This picture shows the implementation of the Dijkstra algorithm in pseudocode. [19]

The Dijkstra algorithm can only be used if the whole graph doesn't contain any negative transitions. This is the case on the 2D grid map, created by the SLAM algorithm. The whole map can be seen as the graph, with every pixel as a node. The transitions between these pixels have always the same values, expect the one to obstacles. There aren't any transitions at all. The set starting point is seen as the starting node and the destination as the end node. In this way it is possible to get shortest route for the car over the whole map.

The next thing, which is needed to navigate the autonomous driving car correctly is the local planner to specify the local route. The global planner only plans the shortest route from A to B based on the recorded map, but if there will be any uncharted obstacles on this route the car would crash into. One more thing to care about is the Ackermann steering of the car, so it has a minimum turning radius. The car used in this thesis has something like two meters. The global planner doesn't care about and calculates like the robot is able to turn on spot and it also doesn't care about the cars' dimension. To fix this problem the *teb_local_planner*, a ROS library for the navigation stack especially for car like robots with Ackermann steering, is used. The *teb_local_planner* integrates the LiDAR sensor data in its algorithm to detect uncharted obstacles during the driving, so it spontaneously recalculates the local path if needed. The local planner also computes the velocity commands based on its path. This planner needs a lots of configuration, because it requires many different information as input, who all affect the result and the performance of the calculation. For

example the minimum steering angle, the speed of the car and also its dimension to know how far the route has to be away from obstacles. For performance improvements the size of the temporary map and its resolution can be defined too, but the less the resolution of the map is, the less obstacles can be detected. A full documentation of the teb_local_planner parameters is available on its section in the ROS wiki.[8] The configuration values used for this thesis can be found in the Appendix B chapter.

Figure 3.5 *Examples of some teb_local_planner arguments*. This picture shows a few of the required parameters for the *teb_local_planner*

3.2.4. Localization

To succeed a well performed navigation the autonomous driving car needs to get localized within the map correctly. From the very beginning and the whole way of the drive, the cars' odeometry is calculated based on the velocity command like the speed and the steering angle. But the car begins on an estimation point so its odeometry isn't exact at all. That's why a localization algorithm is required to get a more precise position of the car within map. This maximize the perfect building of a correct route by the local planner.

To achieve this, the adaptive Monte Carlo localization (amcl) is taken as approach. It's an algorithm to localize a robot in a map using a particle filter implemented for ROS. The algorithm requires a known map created by a laser sensor, like it's done with the *hector_slam* library and the task is to estimate the pose of the car within the map based on its motion and sensing. The algorithm starts with the initial beliefe of the robot's pose, what's in this case the estimated starting point of the car. The state of the robot needs to get estimated at every current time-step *k*. This problem "is an instance of the Bayesian filtering problem, where we are interested in constructing the posterior density $p(x_k \mid Z^k)$ of the current state conditioned on all measurements"[20] with x as state vector.

$$p(x_k|Z^k) \qquad x = \left[x, y, \theta\right]^T \tag{6}$$

In the specific case of using the Monte Carlo filter the density is represented by a set of N random particles[20]:

$$S_k = \{s_k^i; i = 1..N\} \tag{7}$$

For a proper localization it's required to recursively compute the density at each time step. This is done in two phases, the predication phase and the update phase.

In the prediction phase a motion model, is used to predict the current position of the robot. The state x of the time-step *k* is only decent on the previous time-step k-1 with a known control input u_{k-1}. The motion model is presented as a conditional density[20]:

$$p(x_k|x_{k-1}, u_{k-1}) \tag{8}$$

In the Bayesian filtering the predictive density over the state vector x_k is then calculated by integration:

$$p(x_k|Z^{k-1}) = \int p(x_k|x_{k-1}, u_{k-1}) p(x_{k-1}|Z^{k-1}) dx_{k-1} \tag{9}$$

The Monte Carlo localization starts with the set of particles S_{k-1}, computed in the previous time-step and apply the motion model to each particle s^i_{k-1} by sampling from the density $p(x_k \mid s^i_{k-1}, u_{k-1})$. So for each particle s^i_{k-1} a sample s'^i_k is drawn[20].

In the update phase a measurement model is used to integrate information from the sensors to obtain the density function described in (6). Each measurement z_k is conditionally independent of earlier measurements and the measurement model is given in terms of a likelihood. It means that the robot observers z_k at the given location x_k[20].

$$p = (z_k|x_k) \tag{10}$$

The posterior density over x_k is now obtained using the Bases theorem:

$$p = (x_k|Z^k) = \frac{p(z_k|x_k)p(x_k|Z^{k-1})}{p(z_k|Z^{k-1})} \tag{11}$$

The Monte Carlos localization algorithm takes into account the measurement z_k and weight each of the samples S'_k created in the first phase, by the weight m^i_k:

$$m_k^i = p(z_k|s_k'^i) \tag{12}$$

Then a sample s^j_k from $\{s'^i_k, m^i_k\}$ is drawn for each j = 1..N. The whole algorithm is computed recursively. To initialize the filter k = 0 with a random samples $S_0 = \{s_0^i\}$ from the prior $p(x_0)$.

4. Results

4.1. The map

I tested the map algorithm in different indoor environments with different methods to control the LiDAR sensor. The most important thing to care about is, that the laser sensor always have the position, or the pose estimation and the map building will fail. I tried to build a map by holding the sensor in my hands and run around in the area, but I wiggled too much to create an accurate map. The problem is that the sensor is just scans one layer, so if its position isn't straight the matching part does not work. With a well fixed position of the LiDAR sensor it works perfectly to create accurate SLAM maps of a whole indoor area. Another problem is that an obstacle like a glass door isn't detected as one. The size of the map in pixels needs to be defined before starting the algorithm.

Figure 4.1: *SLAM map of a main floor*. The map shows the main floor of my home. The stripes on the right side of the picture shows perfectly the problem of a glass door as obstacle.

All the obstacles are presented as black pixels and all free space, where the car is able to drive are the wide light grey areas. My home is just a small example of a SLAM map. I recorded it by fixing the LiDAR sensor and the laptop on top of a chair and then I moved it around. The stripes on the right of the picture show perfect what happened if the sensor should detect a glass door as an obstacle. It doesn't get detected, because the laser beams aren't reflected by this type of surface.

Next test was to record a map with the LiDAR fixed in the car. Because for the navigation part afterwards the map needs to be recorded on the correct height, so all the obstacles can be rematched in the amcl algorithm. To build the map I moved the car by a hand control.

(a) Robotics lab of the TUM chair VI

(b) Hall of the FMI building of the TUM

Figure 4.2: *SLAM map of TUM rooms.* These tow pictures present the slam map of the (a) robotic lab of the chair VI and of the (b) hall of the FMI building of the Technical University Munich.

The map of the lab is just one room and was only to test the attachment and the position of the LiDAR. The big map of the hall is the evidence, that the algorithm also works well in huge indoor areas. It shows all obstacles like the two famous slides and all meal benches for the students very precisely. This map is the basis of all the on going tests of the navigation and the localization part. It suits perfect as testing area, because of its huge size with lots of obstacle, but without any critical stuff like expensive things that can be broken in case of some test failures.

4.2. The route planning

The whole navigation is tested based on the map of the FMI hall (Fig.4.2(b)), because this is a huge area with lots of space. I tested the calculation of different global routes. I tried long routes, short ones, with lots of obstacles and much free space on the track.

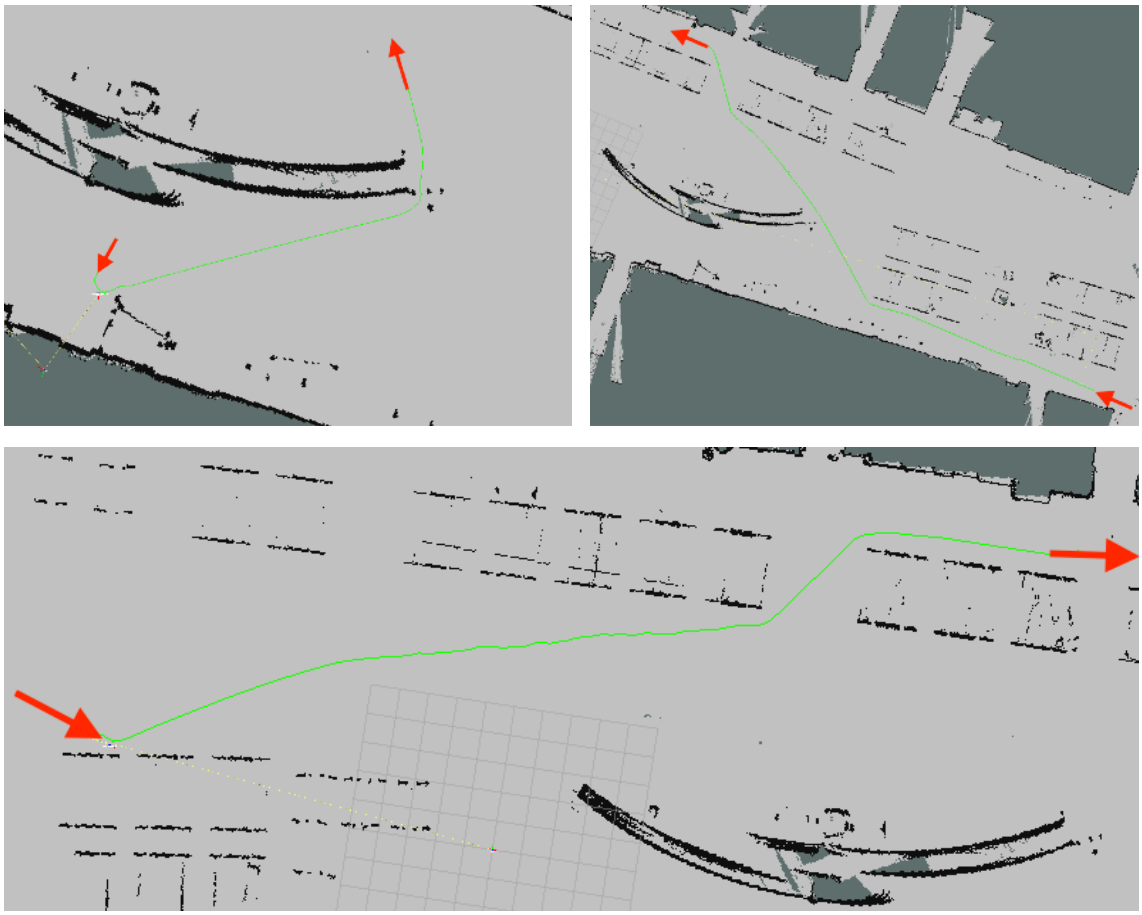


Figure 4.3: *Global routes.* Here are some global routes presented. The green lines are the calculated routes from the estimated starting point (arrows with peak to the lines) to the set goal (arrows with peak away from the lines). The arrow peaks also show the front of the car at the beginning and the end.

All the tested global routes are valid and the car is able to follow them. Compared to the whole map (Fig.4.2(b)) these routes don't contain any weird ways and are really good ones to drive. So the global planner does a very accurate job and is fully compatible to the topic of this thesis. With this accurate results the goal of the global route planning is achieved perfectly.

The more complicated planner was the local one. It needs lots of configuration to calculate accurate and usable results. I tested many different routes in many different constellations of charted and uncharted obstacles to adjust all the required parameters.

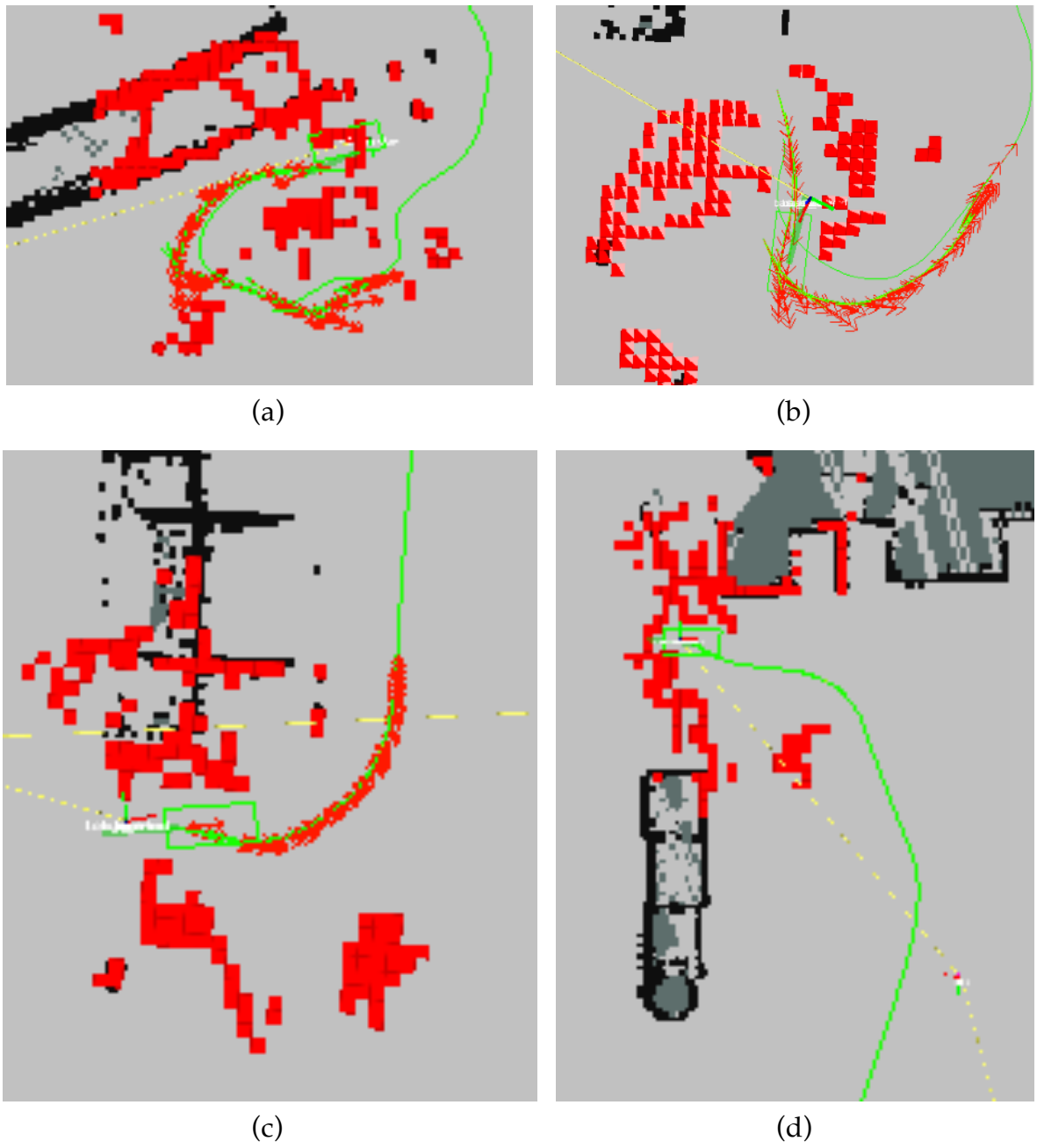


Figure 4.4: *Local routes*. The picture shows four different local routes. Picture (a) and (b) shows the difference between the local and the global route. The green line is the global route and the green line with the red arrows on it is the local route. In (c) the local route is nearly the global route. In (d) they are exactly the same. The red squares are detected obstacles.

All the tested are all valid but they are not the perfect one. Sometimes the planner calculates routes with many switches of forward and backward driving (Fig.4.4(a)(b)) instead of just driving a longer curve. The planner also sometimes starts a recalculation of the path although the path before was valid. These failures happen a small amount of times, but they are still there. To fix them it

still needs more specific adjustment of the planner. In the end the local planner is working and the car reaches its goal, although it sometimes maneuvering a lot to surround obstacles or to go through a gap. The planner isn't optimal to use it for a speed cup or in critical areas like in real road traffic, but in this thesis to control n autonomous driving model car it is sufficient.

4.3. Localization within the map

The Monte Carlo localization were tested on every try run of the car automatically, because the whole driving wouldn't work without it. The car needs to know where it all the time during its drive.

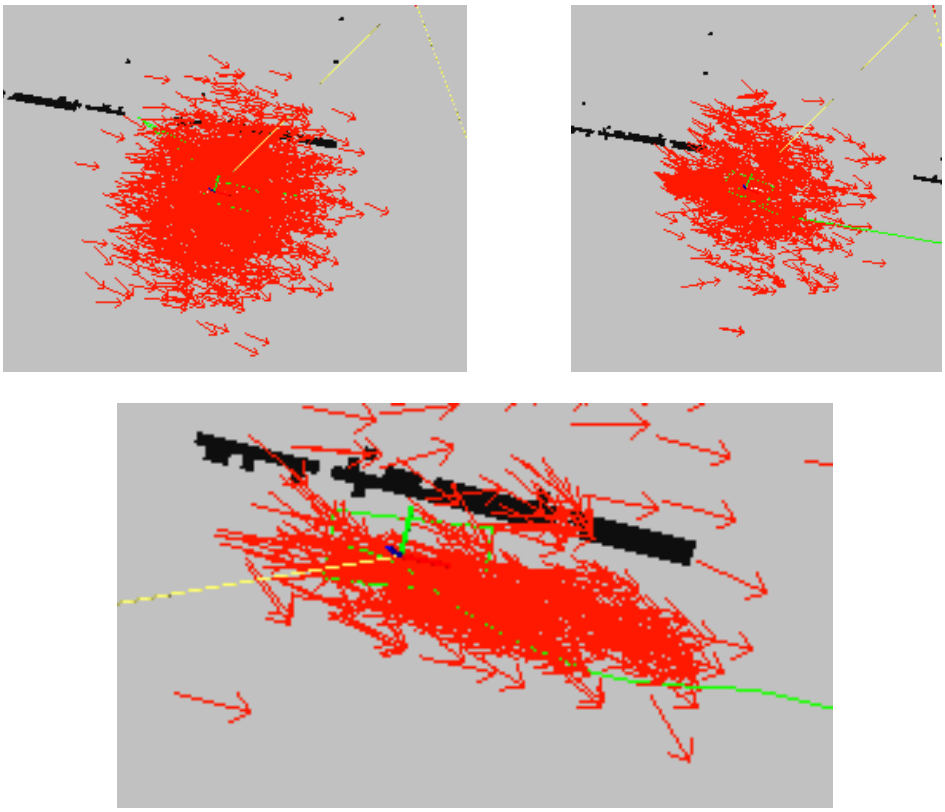


Figure 4.5: *Localization array*. The picture shows the step by step localization during the car's driving. At the beginning the estimation cloud is very big, but as soon as the car starts driving the estimation gets more precise.

The amcl algorithm works good to estimate the portion of the car within the map. But to do so, it is recommended to record the map on the same height as the LiDAR is used to navigate through it. So if the LiDAR is fixed 10 cm above the ground on the car, the map should also be build from this position too. Another important thing I recognized during the tests is, that the speed of the car needs to get published accurate, because the LiDAR measurement is based

on the motion model of the car, like it's described in the section 3.2.4. Expect these two things the localization algorithm works very precisely and is a very good choice. The cars pose within the map is estimated in just a few time steps, so it is also good to use in small environments.

5. Conclusion

In this thesis a fully autonomous driving model car is implemented with ROS as operating system. The whole navigation of the car is based on a recorded SLAM map. The car is able to navigate through an area with the help of a LiDAR sensor and the map. It can successfully find a valid route through the map and around all charted obstacle. Additionally it is able to spontaneously react to uncharted obstacle during its drive and replay the route if needed.

The map is a 2D grid based SLAM map created by the hector_slam library for ROS. It is based on the LiDAR sensor data and an approximative position of the robot.

The whole navigation part of the car is divided into two different planner. One is the global planner to calculate the shortest path from an estimated starting point to a set goal. The calculation is based on all charted obstacles within the recorded map and the Dijkstra algorithm. The second planner is the local planner to find the best route at the current incidents. It will replan the local route if the LiDAR sensor will detect uncharted obstacle. It will also replan the path if it isn't possible to follow the path without ranking. The local planner calculates the velocity commands too.

To localize the robot within the map and during the whole drive a Monte Carlo localization algorithm is used. It is scanning the environment and check if the measurements will match to the charted obstacles within the map. The matching is based on the motion model of the robot. Based on these data the algorithm is able estimate is current position and direction within the map.

5.1. Future work

The car is doing its job to drive autonomously from the starting point to its destination, but the LiDAR sensor is still connected to the laptop via cable and placed on the car. It would be better to implement a real time clock server into whole system, connect the LiDAR to the RaspberryPi board and publish the scanned data via Wifi to the ROS master. So it's not required to walk behind the car during it's driving.

I did lots of configuration on the local planner parameters, but it still can be optimized to find valid routes without ranking so much. With more optimization the car is also able to drive closer to obstacles and will faster reach its goal.

A third step that can be done is to test everything in and outdoor area, if the different algorithms will also work as good as in indoor areas.

Appendix A: Installation

This chapter deals with how to install from scratch all the required tools on the hardware, needed for the autonomous driving car. It deals with the Laptop and the RaspberryPi board. Each section describes what to do, so everyone is able to repeat it by itself after reading this chapter.

1. Laptop

The first thing and the basis of the autonomous driving car is the laptop. It is like a command centre of the whole robot. On the laptop the starting point and the destination of the car should be set on the map. The route of the car, its current position and all obstacles the LiDAR detects will be presented.

The laptop uses Linux Ubuntu as operating system. In this thesis I used the version 14.04, but it is also possible to use the latest current version available, Ubuntu 16.04. All tools introduced in this section are available for that version too.

1.1. ROS

Once Ubuntu is up and running on the laptop, the Robotic Operating System must be installed. It is possible to do this with the Linux package list, but the laptop has to accept software from packages.ros.org for this purpose. This can be done by setting up the *source.list* by typing the following command into the terminal:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
-sc) main" > /etc/apt/sources.list.d/ros-latest.list' [21]
```

The next step is to set up the connection to the ROS key server, as follows it is not possible to download the software:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-
key 421C365BD9FF1F717815A3895523BAEEB01FA116 [21]
```

Afterwards the *source.list* should be updated to the latest version of all packages by typing the basic Linux update command:

```
sudo apt-get update
```

After preparing the laptop as described in the last steps, it is possible to install ROS just via the source list. On the computer the full desktop version is needed, because there is a GUI included called *rviz*. Rviz is required to show the map of the environment, set the starting and the ending point of the car, to portrare the

route and to see all obstacles the LiDAR sensor will detect. To start the installation type the following command into the terminal:

```
sudo apt-get install ros-indigo-desktop [21]
```

The next step is initializing *rosdep*, before ROS can be used. "It enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS"[21]:

```
sudo rosdep init
rosdep update
```

It's recommended to install *rosinstall*, a command line tool, that enables the downloading of ROS packages directly with one command from SCMs (e.g. git):

```
sudo apt-get install python-rosinstall
```

Now the full desktop version of ROS is successfully installed and ready to create a catkin workspace for the project.

First of all create the directory, where the workspace should be saved. Replace the <path> with the favorite directory and <ws_name> with the name of the project:

```
mkdir -p <path>/<ws_name>/src
```

Afterwards change the working directory of the terminal to the *src* folder, created in the last step and initialize the workspace:

```
cd <path>/<ws_name>/src
catkin_init_workspace
```

Now go back to the <ws_name> folder and build the workspace with the *catkin* command, so one *build* and one *devel* folder appear .The last step is to source the bash file:

```
cd <path>/<ws_name>/
catkin_make
source devel/setup.bash
```

The workspace is ready to use and tools needed can be installed.

1.2. urg_node

The *urg_node* can be installed simply, by cloning the GitHub repository. All nodes in ROS will be saved in the *src* folder of the catkin workspace. Just open a

new terminal, change the directory to the *src* folder of the workspace created in section 2.1.1 and clone the *urg_node* repository:

```
cd <path>/<ws_name>/src
git clone https://github.com/ros-drivers/urg_node.git
```

After finishing the download, a *urg_node* folder will appear containing a *CMakeLists.txt*, that needs to be edited. Specify additional locations of header files by adding one command in the file. First of all go to your new folder and start the file with the editor:

```
cd urg_node
vim CMakeLists.txt
```

Now add the following line in before the *catkin_package(...)* command:

```
include_directories(include ${catkin_INCLUDE_DIRS})
```

The next step is to modify two lines in the source code, so the LiDAR will work correctly. Therefore move on to the next *src* folder inside the *urg_node* folder, open the *urg_node.cpp* with the editor and change the *true* in line 210 and the following, so it looks like this:

```
bool publish_intensity;
pnh.param<bool>("publish_intensity", publish_intensity, false);
bool publish_multiecho;
pnh.param<bool>("publish_multiecho", publish_multiecho, false);
```

Last but not least build the whole workspace and install all nodes located in the *src* folder of the workspace. Therefore change the working directory of the terminal back to the root of the workspace:

```
cd <path>/<ws_name>/
catkin_make install
```

1.3. hector_slam

The *hector_slam* package can easily be installed via the source list of the linux system by the *apt-get install* command. Open a new terminal and execute the command:

```
sudo apt-get install ros-indigo-hector-slam
```

1.4. hector_slam_example

The *hector_slam_example* can be installed like a ROS node into the *src* folder of the workspace. Just clone the git repository, add its path to the

ROS_PACKAGE_PATH environment variable and install it with the *rosdep* command:

```
cd <path>/<ws_name>/src
git clone https://github.com/DaikiMaekawa/hector_slam_example.git

ROS_PACKAGE_PATH=<path>/<ws_name>/src/hector_slam_example:
$ROS_PACKAGE_PATH

rosdep install hector_slam_example
```

1.5. navigation_stack

The *navigation_stack* is also rapidly installed via the source list:

```
sudo apt-get install ros-indigo-navigation
```

1.6. teb_local_planner

The *teb_local_planner* is also installed via the source list:

```
sudo apt-get install ros-indigo-teb-local-planner
```

2. RaspberryPi

The RaspberryPi is a small board working directly on the car to communicate with the laptop and forward the velocity commands to the motor management.

The RaspberryPi uses Linux as operating system too. I decided to use Ubuntu Mate 16.04, because it's easy to install and it has a user friendly interface. I haven't tested the tools on other operating systems, so I would advise users to also use this version.

2.1. ROS

Installing the Robot Operating System on the RaspberryPi is the same like on the laptop. First of all update the *source.list* and set up the connection to the key server, like it's done on the laptop. Create a workspace and build it. Detailed information how to it check the section 1.1 of the Installation chapter.

2.2. urg_node

The installation of the *urg_node* on the RaspberryPi is the same like on the Laptop, described in section 2.1.2 of the Installation chapter. Just clone the git repository into the workspace and modify the required files. After installing the

node on the RaspberryPi the LiDAR can be used directly on the car to scan the environment and publish its data.

Appendix B: Configuration

The following sections are about all the configurations needs to be done to use all the tools presented in chapter 2. The different parts describe the correct utilization of the so called launch files in ROS and show how to set the different parameters for a well navigating autonomous car. It also presents how to configure the wifi connection, so the laptop and the RaspberryPi can publish their data to each other.

1. Laptop

Most of the libraries and tools being used for the autonomous driving car are running on the laptop and all of them needs to get configured. This section is about how to setup all of them properly.

1.1. ROS

The most important thing in ROS are the nodes and tools getting started to use the robot. They get started by the so called launch files. These files contain a bunch of all the required tools, libraries and nodes. They are written in XML and have to be saved with the suffix *.launch.*

In the specific case of the autonomous driving car presented in this thesis, two different launch files are required. One starts the *urg_node* combined with the *hector_mapping* library, to create the 2D map of the environment. The other have to load the map, the *urg_node*, and the whole navigation and movement management. Both of them also have to start the user interface *rviz.* Before creating those launch files it's recommended to create a new package in the workspace to save all the required files. Let's do it step by step.

Replace <package_name> with the name the new folder should be called:

```
cd <path>/<ws_name>/src
catkin_create_pkg <package_name>
```

Until the command is finished a new folder, containing a *CMakeLists.txt* and a *package.xml* will appear. Now it's time to customize the package, so it will fit perfectly to the project. First step is to modify the *package.xml* file. It saves the name, version, maintainer, dependencies and license information as XML tags. The file is filled with lots of comments and a few standard tags. Open the file and check it:

```
cd <path>/<ws_name>/src/<package_name>
gedit package.xml
```

Let's check all tags step by step. The <name> tag contains the name of the package:

```
<name><package_name></name>
```

The <version> tag shows the version of the package. It's important to use a three dot convention:

```
<version>0.0.1</version>
```

The <description> tag can be edited at one's leisure and should just describe what is the package about. It's just a user information:

```
<description>Favorite description</description>
```

Next is the <maintainer> tag, an important and required tag in the XML file. It should show others who to contact, if there will be a problem with the package. So one tag is required all other are optional. It follows a specific convention with the email address as attribute[22]:

```
 <maintainer email=„email@provider.de“>Name</maintainer>
```

The <license> tag is also required and save the license used for the package:

```
<license>MIT</license>
```

Last step is to set the dependencies needed for the autonomous car. These are all we want to be available at build and run time. In this case a few ones will be set:

```
<buildtool_depend>catkin</buildtool_depend>
<run_depend>hector_mapping</run_depend>
<run_depend>hector_geotiff</run_depend>
<run_depend>hector_trajectory_server</run_depend>
<run_depend>hector_geotiff_plugins</run_depend>
<run_depend>hokuyo_node</run_depend>
<run_depend>urg_node</run_depend>
<run_depend>depthimage_to_laserscan</run_depend>
<run_depend>tf</run_depend>
<run_depend>rviz</run_depend>
```

Save and close the file. Now the *package.xml* file is finished. The <export> tag isn't required for this project, so it can stay empty. The file whole I used for the project can be checked in the Appendix chapter of the thesis.

Furthermore a few lines needs to be added in the *CMakeLists.txt*. The file is located in the same folder like the *package.xml* file edited before. Open the *CMakeLists.txt* place the following lines among the *find_package(catkin Required)* command:

```
catkin_package()
install (DIRECTORY launch
      DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
      USE_SOURCE_PERMISSIONS)
```

After editing these two file it's finally time to go on with the launch files. At the beginning the map of the area in which the car will drive needs to be created, so let's start with the required launch file.
First of all create a directory in the package folder, where all the files will be saved. Then open the new folder, create a file inside and call it *mapping.launch.* Let's check step by step what to add to the file. All parameters and nodes added to the file are between *<launch>* tags. At the beginning implement the following four parameter:

```
<launch>
<param name="pub_map_odom_transform" value="true"/>
<param name="map_frame" value="map"/>
<param name="base_frame" value="base_frame"/>
<param name="odom_frame" value="odom"/>
```

To scan the environment the LiDAR sensor data is required, so it's needed to include the *urg_node* package. The sensor is connected via ethernet, so the IP address of the sensor has to be set as node parameter. The address of the sensor used in this thesis is set to 192.168.0.10 by the developer:

```
<node pkg="urg_node" type="urg_node" name="urg_node">
   <param name="ip_address" type="string" value="192.168.0.10"/>
</node>
```

To create the whole map a few transform information are required. They present the distances from the center of the car to the LiDAR sensor. So it basically shows where the sensor is placed in the car. These information are required to work with the mapping library:

```
<node pkg="tf" type="static_transform_publisher" name="map_2_odom"
args="0 0 0 0 0 0 /map /odom 100"/>

<node pkg="tf" type="static_transform_publisher"
name="odom_2_base_footprint" args="0 0 0 0 0 0 /odom /base_footpirnt
100"/>

<node pkg="tf" type="static_transform_publisher"
name="base_footprint_2_base_link" args="0 0 0 0 0 0 /base_footprint /
base_link 100"/>

<node pkg="tf" type="static_transform_publisher"
name="base_link_2_base_stabilized_link" args="0 0 0 0 0 0 /base_link /
base_stabilized 100"/>
```

```
<node pkg="tf" type="static_transform_publisher"
name="base_frame_2_laser_link" args="0 0 0 0 0 0 /base_frame /laser
100"/>

<node pkg="tf" type="static_transform_publisher"
name="base_2_nav_link" args="0 0 0 0 0 0 /base_frame /nav 100"/> [6]
```

To display the map and see the driven trajectory the interface *rviz* is also required. As config for the *rviz* tool, it is possible to use the one from the *hector_slam_example* package:

```
<node pkg="rviz" name="rviz" args="-d $(find hector_slam_example)/
launch/rviz_cfg.rviz"/>
```

To finish this launch file just two more files need to be included and the <*launch*> tag have to get closed:

```
<include file="$(find hector_slam_example)/launch/
default_mapping.launch"/>
<include file="$(find hector_geotiff)/launch/geotiff_mapper.launch"/>
</launch>
```

The second launch file is to start the autonomous driving car and to load the map and all the navigation, movement and localization packages. Create a new file and name it *autonomous_drive.launch*. My launch file starts with a map server. It is important because the server is loading the 2D map of the environment in which the car is driving. To start open the still empty *autonomous_drive.launch* and start editing:

```
<launch>
<node name="map_server" pkg="map_server" args="$(arg map_file)"/>
```

After loading the map we need to set the parameter the same values as in the *mapping.launch*:

```
<param name="pub_map_odom_transform" value="true"/>
<param name="map_frame" value="map"/>
<param name="base_frame" value="base_frame"/>
<param name="odom_frame" value="odom"/>
```

Also the transform nodes are the same as in the *mapping.launch* and can be added by copy and paste :

```
<node pkg="tf" type="static_transform_publisher"
name="base_link_2_base_stabilized_link" args="0 0 0 0 0 0 /base_link /
base_stabilized 100"/>

<node pkg="tf" type="static_transform_publisher"
name="base_frame_2_laser_link" args="0 0 0 0 0 0 /base_frame /laser
100"/>
```

```
<node pkg="tf" type="static_transform_publisher"
name="base_2_nav_link" args="0 0 0 0 0 0 /base_frame /nav 100"/>[6]
```

The rest of the launch file needs a configuration of all the packages itself, so it's explained in each subsection what to add. This gives a better understanding why everything is added.

1.2. navigation_stack

The navigation stack includes the two route planners and offer the possibility to drive autonomously. First of all the different cost maps needs to be added to the workspace. Create a new folder called *costmaps* in the package folder and change into the directory:

```
mkdir <path>/<ws_name>/src/<package_name>/costmaps
cd <path>/<ws_name>/src/<package_name>/costmaps
```

Create the following 4 .yaml files in the folder:

```
costmap_common_params.yaml
local_costmap_params.yaml
global_costmap_params.yaml
teb_local_planner_params.yaml
```

These are the files containing all the parameters for the different manners and the navigation stack. Let's start with the configuration of the *costmap_common_params.yaml* , open the file and add the filling lines:

```
footprint: [ [-0.1,-0.125], [0.5,-0.125], [0.5,0.125], [-0.1,0.125] ]
transform_tolerance: 0.2
map_type: costmap
obstacle_layer:
 enabled: true
 obstacle_range: 3.0
 raytrace_range: 3.5
 inflation_radius: 0.2
 track_unknown_space: false
 combination_method: 1
 observation_sources: laser_scan_sensor
 laser_scan_sensor: {sensor_frame: base_link, data_type: LaserScan,
topic: scan, marking: true, clearing: true}
inflation_layer:
  enabled:              true
  cost_scaling_factor:  10.0  # exponential rate at which the obstacle
cost drops off (default: 10)
  inflation_radius:     0.5  # max. distance from an obstacle at which
costs are incurred for planning paths.
static_layer:
  enabled:              true
  map_topic:            "/map"
```

Then open the *local_costmap_params.yaml* and fill in the following:

```
local_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 3.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 4
  height: 4
  resolution: 0.1
  transform_tolerance: 10

  plugins:
   - {name: static_layer,        type: "costmap_2d::StaticLayer"}
   - {name: obstacle_layer,      type: "costmap_2d::ObstacleLayer"}
```

Go on with the *global_costmap_params.yaml* and add the following:

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 3.0
  static_map: true

  transform_tolerance: 30
```

And last but not least open the *teb_local_planner_params.yaml* and add the following:

```
TebLocalPlannerROS:
 odom_topic: /odom
 map_frame: /map
 # Trajectory
 teb_autosize: True
 dt_ref: 0.4
 dt_hysteresis: 0.1
 global_plan_overwrite_orientation: True
 max_global_plan_lookahead_dist: 3.0
 feasibility_check_no_poses: 2
 allow_init_backward_motion: false
 # Robot
 max_vel_x: 0.4
 max_vel_x_backwards: 0.2
 max_vel_theta: 0.3 # the angular velocity is also bounded by
min_turning_radius in case of a carlike robot (r = v / omega)
 acc_lim_x: 0.5
 acc_lim_theta: 0.5
 # ****************** Carlike robot parameters ********************
 min_turning_radius: 2.7       # Min turning radius of the carlike
robot (compute value using a model or adjust with rqt_reconfigure
manually)
 wheelbase: 0.55                 # Wheelbase of our robot
```

```
cmd_angle_instead_rotvel: True # stage simulator takes the angle
instead of the rotvel as input (twist message)
 footprint_model: # types: "point", "circular", "two_circles", "line",
"polygon"
   type: "line"
   #radius: 0.2 # for type "circular"
   line_start: [-0.55, 0.0] # for type "line"
   line_end: [0.0, 0.0] # for type "line"
   #front_offset: 0.32 # for type "two_circles"
   #front_radius: 0.27 # for type "two_circles"
   #rear_offset: -0.25 # for type "two_circles"
   #rear_radius: 0.27 # for type "two_circles"
   #vertices: [ [0.25, -0.05], [0.18, -0.05], [0.18, -0.18], [-0.19,
-0.18], [-0.25, 0], [-0.19, 0.18], [0.18, 0.18], [0.18, 0.05],
[0.25,0.2] # for type "polygon"
 # GoalTolerance
 xy_goal_tolerance: 0.5
 yaw_goal_tolerance: 0.3
 free_goal_vel: False
 # Obstacles
 min_obstacle_dist: 0.25 # This value must also include our robot's
expansion, since footprint_model is set to "line".
 include_costmap_obstacles: True
 costmap_obstacles_behind_robot_dist: 1.5
 obstacle_poses_affected: 30
 costmap_converter_plugin: ""
 costmap_converter_spin_thread: True
 costmap_converter_rate: 5
 # Optimization
 no_inner_iterations: 3
 no_outer_iterations: 3
 optimization_activate: True
 optimization_verbose: False
 penalty_epsilon: 0.1
 weight_max_vel_x: 2
 weight_max_vel_theta: 1
 weight_acc_lim_x: 1
 weight_acc_lim_theta: 1
 weight_kinematics_nh: 1000
 weight_kinematics_forward_drive: 100
 weight_kinematics_turning_radius: 1
 weight_optimaltime: 1
 weight_obstacle: 50
 weight_dynamic_obstacle: 10 # not in use yet
 # Homotopy Class Planner
 enable_homotopy_class_planning: True
 enable_multithreading: True
 simple_exploration: False
 max_number_classes: 4
 selection_cost_hysteresis: 1.0
 selection_obst_cost_scale: 1.0
 selection_alternative_time_cost: False
 roadmap_graph_no_samples: 15
 roadmap_graph_area_width: 5
 h_signature_prescaler: 0.5
 h_signature_threshold: 0.1
 obstacle_keypoint_offset: 0.1
 obstacle_heading_threshold: 0.45
 visualize_hc_graph: False
```

These are all the parameters, who affect the result and the performance of the local planner described in the thesis. After creating all these parameter files, they need to get added to the *.launch* of the robot:

```
<node pkg="move_base" type="move_base" respawn="false"
name="move_base" output="screen">

<rosparam file="$(find <package_name>)/costmaps/
costmap_common_params.yaml“ command="load" ns="global_costmap" />

<rosparam file="$(find <package_name>)/costmaps/
costmap_common_params.yaml“ command="load" ns="local_costmap" />

<rosparam file="$(find <package_name>)/costmaps/
local_costmap_params.yaml“ command="load" />

 <rosparam file="$(find find <package_name>)/costmaps/
global_costmap_params.yaml“ command="load" />

<rosparam file="$(find find <package_name>)/costmaps/
teb_local_planner_params.yaml“ command="load" />

<param name="base_local_planner" value="teb_local_planner/
TebLocalPlannerROS" />
        <param name="controller_frequency" value="5.0" />
        <param name="controller_patience" value="10.0" />

<param name="clearing_rotation_allowed" value="false" />
</node>
```

Now the navigation_stack is fully implemented.

1.3. amcl

For the localization part an odeometry publisher is needed. So first of all create a source folder in your package folder:

```
mkdir <path>/<ws_name>/src/<package_name>/src
```

First create a file called *odom_publish.cpp* and fill it with the following lines of code. This code work with the car used in this thesis. It need to be edited specific for the car used:

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>
float velocity;
float angle;
void vel_sub (const geometry_msgs::Twist::ConstPtr& vel_msg) {
     velocity = vel_msg->linear.x;
    angle = vel_msg->angular.z;
}
double dt;
```

```
double delta_x;
double delta_y;
double delta_th;
int main(int argc, char** argv){
  ros::init(argc, argv, "odometry_publisher");
  ros::NodeHandle n;
  ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom",
50);
  tf::TransformBroadcaster odom_broadcaster;
    geometry_msgs::Twist vel_cmd;
    ros::Subscriber cmd_vel=n.subscribe("cmd_vel" , 10 , vel_sub);
  double x = 0.0;
  double y = 0.0;
  double th = 0.0;
  double vx = velocity;
  double vy = -0.0;
  double vth = angle;
  ros::Time current_time, last_time;
  current_time = ros::Time::now();
  last_time = ros::Time::now();
  ros::Rate r(50);
  while(n.ok()){
    ros::spinOnce();               // check for incoming messages
    current_time = ros::Time::now();
    //compute odometry in a typical way given the velocities of the
robot
    if(velocity > 0){
     dt = (current_time - last_time).toSec();
     delta_x = (0.45 * cos(th) - vy * sin(th)) * dt;
     delta_y = (0.45 * sin(th) + vy * cos(th)) * dt;
     delta_th = angle * dt;
}
    else if(velocity < 0){
     dt = (current_time - last_time).toSec();
     delta_x = (-0.55 * cos(th) - vy * sin(th)) * dt;
     delta_y = (-0.55 * sin(th) + vy * cos(th)) * dt;
     delta_th = angle * dt;
}
    else {
         dt = (current_time - last_time).toSec();
         delta_x = (velocity * cos(th) - vy * sin(th)) * dt;
     delta_y = (velocity * sin(th) + vy * cos(th)) * dt;
     delta_th = angle * dt;
}
    x += delta_x;
    y += delta_y;
    th += delta_th;
//since all odometry is 6DOF we'll need a quaternion created from yaw
    geometry_msgs::Quaternion odom_quat =
tf::createQuaternionMsgFromYaw(th);
    //first, we'll publish the transform over tf
    geometry_msgs::TransformStamped odom_trans;
    odom_trans.header.stamp = current_time;
    odom_trans.header.frame_id = "odom";
    odom_trans.child_frame_id = "base_link";
    odom_trans.transform.translation.x = x;
    odom_trans.transform.translation.y = y;
    odom_trans.transform.translation.z = 0.0;
    odom_trans.transform.rotation = odom_quat;
```

```
    //send the transform
    odom_broadcaster.sendTransform(odom_trans);
    //next, we'll publish the odometry message over ROS
    nav_msgs::Odometry odom;
    odom.header.stamp = current_time;
    odom.header.frame_id = "odom";
    //set the position
    odom.pose.pose.position.x = x;
    odom.pose.pose.position.y = y;
    odom.pose.pose.position.z = 0.0;
    odom.pose.pose.orientation = odom_quat;
    //set the velocity
    odom.child_frame_id = "base_link";
    odom.twist.twist.linear.x = vx;
    odom.twist.twist.linear.y = vy;
    odom.twist.twist.angular.z = vth;
    //publish the message
    odom_pub.publish(odom);
    last_time = current_time;
    r.sleep();
  }
}
```

Now it's required to add the following two lines at the end of the *CMakeLists.txt* of the Package, so the *odom_publish.cpp* will be build:

```
add_executable(odom_publish src/odom_publish.cpp)
target_link_libraries(odom_publish ${catkin_LIBRARIES})
```

Now we need to add *amcl* and the *odom_publish.cpp* to the launch file:

```
<include file="$(find amcl)/examples/amcl_omni.launch" />
<node pkg="<package_name>" type="odom_publish" name="odom_publish"/>
```

At the end build the workspace

2. RaspberryPi

On the RaspberryPi an interface to forward the velocity commands to the Arduino board is required. Create a file in a catkin_package and subscribe the *cmd_vel* topic and send the data to the motor management. My code is a python script, but on another car of course the interface needs to be adjusted:

```
import rospy
import serial
from geometry_msgs.msg import Twist
port = '/dev/ttyUSB0'
ard = serial.Serial(port, 19200)
buffer = "||"
def data_to_car(data):
    global buffer
    speed =int(data.linear.x*10)
        angle =(round(data.angular.z, 2)*60)*-1
```

```
        if speed==0.0:
                toAdr = "||||b;" + str(angle) + ";"
        elif speed > 0.0:
                toAdr = "||||" + "8" + ";" + str(angle) + ";"
    else:
        toAdr = "||||" + "-25" + ";" + str(angle) + ";"
    if buffer != toAdr:
        rospy.loginfo("Try to put: " + toAdr);
        ard.write(toAdr)
            ard.flush()
        buffer = toAdr
def listener():
    rospy.init_node('car_interface', anonymous=True)
    rospy.Subscriber("cmd_vel", Twist, data_to_car)
    #ard.close()
    rospy.spin()
if __name__ == '__main__':
    listener()
```

Bibliografie

[1] Statistisches Bundesamt, Unfallbilanz 2015: Mehr Unfälle und mehr Verkehrstote. July 2016.

[2] Esser, M., Automated driving. June 2015.

[3] Open Source Robotics Foundation. *About ROS*. Available from: http://www.ros.org/about-ros/.

[4] Rockey, C. *Package Summary*. Available from: http://wiki.ros.org/urg_node.

[5] Kohlbrecher, S. *Package Summary*. Available from: http://wiki.ros.org/hector_mapping.

[6] Maekawa, D. *hector_slam_example*. Available from: https://github.com/DaikiMaekawa/hector_slam_example.

[7] Marder-Eppstein, E. *Package Summary*. Available from: http://wiki.ros.org/navigation.

[8] Rösmann, C. *Package Summary*. Available from: http://wiki.ros.org/teb_local_planner.

[9] Rösmann, C. *Planning for car-like robots*. Available from: http://wiki.ros.org/teb_local_planner/Tutorials/Planning%20for%20car-like%20robots.

[10] Hugh F. Durrant-Whyte, J.J.L., Mobile Robot Localization by Tracking Geometric Beacons.

[11] Randall Smith, M.S., Peter Cheeseman, Estimating Uncertain Spatial Relationships in Robotics.

[12] Søren Riisgaard, M.R.B., *SLAM for Dummies*.

[13] Hokuyo Automatic Co. LTD. *UTM-30LX-EW*. Available from: http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/utm-30lx-ew/.

[14] Seeed studio. *RPLIDAR - 360 degree Laser Scanner Development Kit*. Available from: https://www.seeedstudio.com/RPLIDAR-360-degree-Laser-Scanner-Development-Kit-p-1823.html.

[15] Microsoft Corporation. *Kinect Sensor*. 2012; Available from: https://msdn.microsoft.com/en-us/library/hh438998.aspx.

[16] Solà, J., Simulataneous localization and mapping with the extended Kalman filter. 2014.

[17] LTD, H.A.C. Scanning Laser Range Finder UTM-30LX-EW Specification. 2012.

[18] Stefan Kohlbrecher, O.v.S., Johannes Meyer, Uwe Klingauf, A Flexible and Scalable SLAM System with Full 3D Motion Estimation. Technische Universität Darmstadt.

[19] Yan, M. *Dijkstra Algorithm*. Available from: http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf.

[20] Frank Dellaert, D.F., Wolfram Burgard, Sebastian Thrun,, *Monte Carlo Localization for Mobile Robots*, in *Computer Science Department*. Carnegie Mellon University.
[21] Open Source Robotics Foundation. *Ubuntu install of ROS Indigo*. Available from: http://wiki.ros.org/indigo/Installation/Ubuntu.
[22] Open Source Robotics Foundation. *CreatingPackage*. Available from: http://wiki.ros.org/ROS/Tutorials/catkin/CreatingPackage.