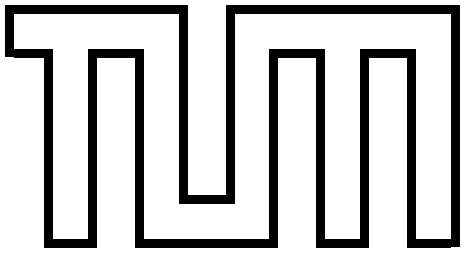


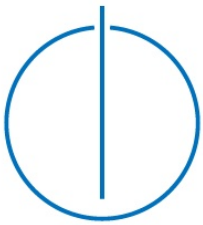
Technische Universität München

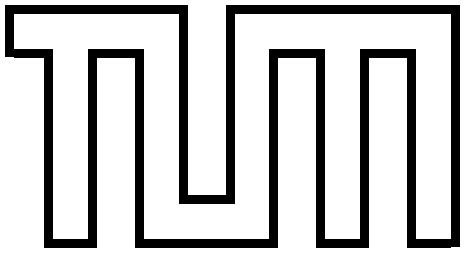
Fakultät für Informatik

Bachelor's Thesis in Informatik

Autonomous Driving navigated by combining the data of a Lidar Sensor and Stereo Camera

Heiko Lengenfelder





Technische Universität München

Fakultät für Informatik

Bachelor's Thesis in Informatik

Autonomous Driving navigated by combining the data of a Lidar Sensor and Stereo Camera

Autonomes Fahren navigiert durch das Vereinigen der Informationen von einer Lidar und Stereo Kamera

Author: Heiko Lengenfelder

Supervisor: Prof. Dr.-Ing habil. Alois Christian Knoll

Advisor: Biao Hu

Submission: February 15, 2017

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

München, February 15, 2017

(Heiko Lengenfelder)

Abstract

The goal of this bachelor's thesis was it to use both a Lidar sensor and a stereo camera to retrieve data and with that navigate a small-scale autonomous car. The computations were to be performed on a lightweight device (Raspberry Pi). In this work there is a focus on the software setup used, but there is also some insight of the hardware side. The navigation algorithm, witch is embedded into the ROS environment, gets illustrated and some optimization work is also part of this thesis. It was possible to implement a simple collision avoidance algorithm for the car, that worked well with the Lidar system. Though the incorporation of the stereo data turned out unsuccessful, because of performance issues.

Inhaltsangabe

Das Ziel dieser Bachelor Arbeit war es mit den Daten eines Lidar Sensors und einer Stereo Kamera die Navigation eines kleinen selbst fahrenden Autos zu steuern. Die nötigen Berechnungen sollten auf einem leicht gewichtigem Gerät (Raspberry Pi) durchgeführt werden. Der Fokus dieser Arbeit liegt hierbei auf der genutzten und entwickelten Software, wobei trotzdem ein Augenmerk auf die Hardware gelegt wird. In der Arbeit wird die Funktionsweise des Navigationsalgorithmuses, welcher in eine ROS Umgebung gebettet ist, genauer beleuchtet. Außerdem wird auf dessen Performanz Optimierung eingegangen. Das Ziel dieser Arbeit wurde in so fern erreicht, dass es möglich war einen simplen Algorithmus zur Kollisionsvermeidung zu implementieren, welcher vor allem mit dem Lidar Sensor gut funktioniert. Bei dem Einbinden der Stereo Daten kam es jedoch zu Schwierigkeiten, welche auf einen zu hohen Rechenaufwand zurückzuführen sind.

Acknowledgments

I would like to especially thank Biao Hu to advise me on such an interesting topic like autonomous driving. I am thankful for his patience when things did not work out and/or took long to finish. I am also thankful to Prof. Alois Christian Knoll for letting me do this work at his chair. More thank goes to Felix Feik, who also worked on a bachelor's thesis on this car and has been a great support in the early stages of the project. I also want to thank two students that worked on the car on behalf of their attendance of the "Hardware/Software Co-Design with a LEGO car" course. The first is Quentin Schlitz for his collaboration to get the car running. Special thanks goes to the second of those students. Without Justin Iszatt the car probably would have never worked properly. Thank you!

Contents

1. Introduction

1.1 Motivation

In recent times the topic of autonomous driving has got a rising amount of attention. Even enough to end up as a topic of public debate. This is caused by a general change in the autonomous industry where companies try to promote their products by advanced functionality. There already exist car on the public marked that provide simple autonomous tasks like parking or holding a driving lane. This results in a lot of research taking place in the sector of autonomous driving. Witch directly leads to a variety of feasible tools being established in this region of research. One of them is the Lidar sensor, witch originated from different subjects like land surveying, but turns out to be a effective data gathering device for autonomous driving.
The project covered by this thesis is built upon using such a Lidar sensor as the main tool of information collection. The Lidar sensor is used in combination with a more traditional stereo camera to detect the cars surrounding. From that the goal is it to develop a navigation algorithm for simple collision avoidance. For that a small scale car will be use. Therefore the two sensors are mounted to that car and a smal computation unit will be used to run the navigation processing. This in the end should lead to the car driving autonomously.

1.2 Structure

To get a good sense about the specific setup in witch the navigation algorithm should operate at the beginning of this thesis all the hardware used will get presented. This will cover the data about the car, the characteristics of the sensors as well as the properties of the data processing tools used. Leading that will be a section that will describe the software used that should lead to the car driving. This includes the internal functionality of navigation algorithm used and the framework in witch it is embedded in. Because a light weight computation device is used on witch the software is running, some performance optimization were needed to get the algorithm running at a reasonable speed. This will be covered by the following section. In the end the car should be able to drive autonomously. So there were tests made to test this functionality. From that the end results of the thesis were concluded. This will be the topic of the last section of this thesis.

2. Hardware

All hardware involved reaches from a very low level, for example the motor and steering control of the car, to rather high level as for example the raspberry pi. This section tries to cover all the used hardware and the setup of those. Also it will give a first insight on the way data is moved between all the components. The hardware will be structured into three parts. On the one hand there is the lidar sensor, which is the core tool the whole project is built around. Then there is the raspberry pi on which most of the computation takes place. When talking more precisely about the software and its performance, the pis hardware data will get important. And last obviously there is the actual car that in the end should drive by itself.

2.1 Car

Since this is a small scale project, it is also done on a small scale car. Its properties are very important, since the navigation algorithm needs to be fit to the specific parameters of the car to work properly. The specific output of the algorithm heavily relies on the movement speed, maximum steering angle, size of the car and also on the more detailed behavior of some of the components. These parameters will be analyzed in this section.

Figure 2.1: The car in front view.

2.1.1 Car Properties

Most dimensions of the car are not necessary to know for this project. Although the width of the car ($0.55m$) is important, since this value is essential for collision avoidance. Also the wheelbase (also $0.55m$) is relevant, because by using the Ackermann Steering principle it could be used to compute the needed steering angle to follow a certain path. During testing it turned out that this method did not lead to the car driving the aimed for driving lane, so a different approach was taken (cf. Testing). Though to safe a safety distance the car should hold this value still is of importance. The height of the car is $40cm$. This property comes into play in a case of moving below obstacles like tables.

2.1.2 Arduino Nano

The micro controller used to control the steering servo and the electric motor is a "Arduino Nano w/ ATMEGA328". It is able to produce multiple PWM signals on hardware level. Two of those are used to control the car. The Arduino is powered via a mini USB port, which in is connected to the Raspberry Pi. This connection is also used for serial communication.

2.1.3 Steering and Motor

For controlling the Steering angle there are two "CYS S8203 Servos" put on the car. They are controlled by PWM signals produced by the Arduino Nano. The maximum steering angle the servos are able to perform in the given setup is supposed to be 15°. This angle is capped by a maximum duty cycle the micro controller will produce, to prevent the servos of trying to set up a state the physical setup is incapable of doing.
To accelerate the car there is a "EZRUN-150A-PRO brushless speed controller" controlling a brushless DC motor. In theory this motor is able to get the car to a maximum speed of $100\frac{km}{h}$. Equivalent to the steering control, the maximum duty cycle of the PWM signal produced is capped by the Arduino Nano. This leads to a maximum velocity of approximately $7\frac{km}{h}$.

2.1.4 Braking

There are no usable brakes attached to the car. Yet the car still needs a way to react to possible collisions. The current approach is it to let the motor accelerate in the backwards direction for a short period of time and then let at low speed the friction of the gear and motor stop the car completely. This leads to a rather long braking distance, but with a maximum speed of $7\frac{km}{h}$ this is acceptable. This method also wears on the gear of the car and should probably not be used at higher velocities.

2.1.5 Ultrasonic Sensor

The speed control and also the braking of the car is mostly managed by the navigation algorithm running on the Raspberry Pi. This could lead to situations in which the communication between the car and the Pi breaks of and because of the lack of new control commands the car stays in its current steering and motor state. To still avoid collisions in this case, there is a ultrasonic sensor attached to the front of the car. The exact model used is the "KS103 ultrasonic sensor". It constantly measures the distance to objects ahead and is directly connected to the Arduino Nano.

2.2 Lidar sensor

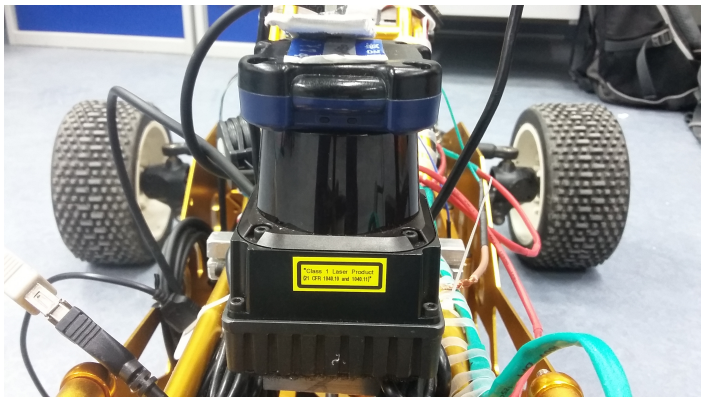


Figure 2.2: The Hokuyo Lidar sensor

The word "lidar" is originally a combination of the words light and radar. Nowadays it sometimes is also referred as short for "light detection and ranging". That itself already explains the core functionality of the sensor. It uses a laser to detect the distance to objects.

2.2.1 History

The lidar sensor has been around for quit some time. It was invented in the 1960s as a result of discovering the laser. Right from the beginning the lidar was used by NASA. A lidar system has been put on the Apollo missions 15 to 17[1]. It was used to get data about the surface of the moon. Today it is still used in the field of land surveying, because of the lidars high accuracy. Recently the lidar system gained attention in autonomous driving, since it is capable of not only detecting objects in a $360^{/circ}$ angle, but also can determine the speed their respective speed.

2.2.2 Functionality

As already mentioned a lidar uses a laser to detect the distance of surrounding objects. It sends out one short beam in certain direction and then maps the intensity of the returning light over

time. The distance of the hit object can be easily deduced from the time it took the light to return. But it is also possible to get more information about the object from the shape of the light intensity curve. Lidar systems are for example used to gain data about tree vegetation, because they are able to detect the first impact onto leaves put at the same time detect the distance to the surface bellow. Though in this project we will only focus on the ability of measuring distances.
Most of the times and also in the case of this project, lidar systems only use one light source for one level of measuring. A mirror is used to redirect the beam into different angles.

2.2.3 Hokuyo Lidar

The specific lidar used for this project is the "UTM-30LX-EW" produced by the Japanese company "HOKUYO AUTOMATIC CO., LTD.". The laser it uses has a wavelength of $\lambda = 905nm$ so its in the space of infrared light. Its circular scanning range is 270° and it can measure distances up to $30m$ with an accuracy of $\pm 50mm$. It has a scan rate of $40hz$, meaning each scan takes $25ms$ to do. In that time it performs 1080 individual measurements each 0.25° apart from each other. All of them are taken on the same height level so the data lies in a two dimensional space. Therefore each data point can be described as a 2-tuple (θ, d) of angle θ and its corresponding measured distance d.
The Hokuyo lidar needs to be powered by a direct currency of $12V$. For communication of the scan data the sensor uses an Ethernet cable. The exact protocol used will not be covered by this work, because a driver will be used to make the scan data available.

2.3 Stereo camera

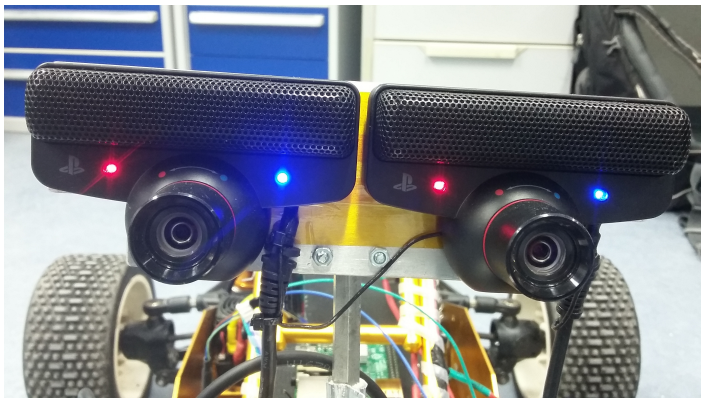


Figure 2.3: The two PS3 Eye cameras

As another device to generate data of the cars surroundings there is also a stereo camera attached to it. It is able to gain 3-dimensional image data of the front area of the car. In this project it will be used to cope with some weaknesses the used Lidar system has. Though because it lacks in accuracy and efficiency, the stereo camera will only be used as a extension to the navigation solution provided.

2.3.1 Functionality

The stereo camera functions in a way much like the human eyes. By simultaneously generating two images data about the distance to objects can get retrieved. On the hardware side the distance does not get directly delivered. It needs to be computed from the images retrieved of the cameras. To do so points of the same object get located on both images. These points should have a slightly different location on each of the images, since the frames were taken from different angles. From the spacing between those image points and the fixed distance of the two lenses the distance of these points can get estimated. Hereby it is important to ensure the two images are taken at the same time.

2.3.2 PS3 Eye Cameras

The specific setup used are two "PS3 Eye" cameras. They are supposed to be used with the PlayStation 3 to add some capabilities to the gaming device (e.g. motion captioning). They were installed on the car for a different project, because they were low-cost at a decent quality. Also they can be synchronized on hardware level to ensure frames being taken at the same time. They feature a resolution of 640×480 with a frame rate of 30fps. They each a use USB to transfer the video data.
In the current setup the two cameras are placed directly next to each other on top of a pole. This is so the ground takes up a smaller portion of the frame. They are the highest part of the car.

2.4 Raspberry Pi

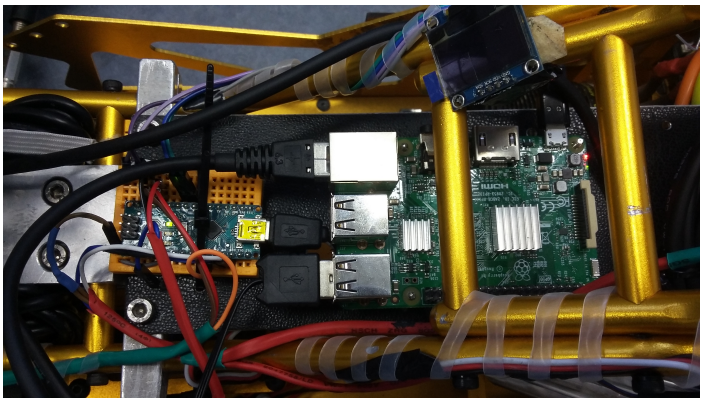


Figure 2.4: Arduino Nano (left) and Raspberry Pi (right) mounted to the car.

The majority of the computation needed is performed on a raspberry pi. That is a small scale computer able to run a operating system, which enables it to run the ROS environment, and it also provides a variety of peripherals needed to communicate with the car and sensors. The raspberry pi also has a moderate amount of computational power which is enough to run the needed algorithms.

2.4.1 Technical data

More specifically all the hardware of the raspberry pi is fit on a $8.5cm \times 6.5cm$ board. The model used for this project is the "Raspberry Pi 3 Model B". It provides a 64-bit ARMv8 processor with $1.2GHz$ on each of the four cores. Furthermore it has $1GB$ RAM and in this work is used with a $32GB$ micro SD card for physical memory, though that amount is not needed. There is also a Ethernet port, which is needed for communication with the lidar sensor, as well as four USB ports mounted on it. There are also GPIO pins on the raspberry, that could be used for serial communication to the Arduino board, though in this case one of the USB ports is used for this task.

2.5 Power supply

The car, that includes the engine and the steering servos, and the Lidar sensor both can be supplied by a $12V$ direct current. So most of the times one battery was used to supply both of them. The battery has an amperage of 5200mAh.
The power for the Arduino board and the the two cameras of the stereo setup can be provided via their respective USB connections. So they are indirectly supplied by the Raspberry Pi. The Raspberry Pi is powered by $5.1V$ obtained via a micro USB port, which is connected to a rechargeable battery.

2.6 Hardware testing

A lot of the hardware components turned out to behave in unexpected manners. So it was necessary to do some hardware testing to get some of the specific parameters needed.
Even though in theory the duty cycles of the PWM signals that control the steering and motor speed should be possible to be computed verily easily, because of the linear mapping of the PWM signal to the resulting angle and velocity, the behavior of the car was not as expected in the practical tests.

2.6.1 Motor

When trying through different duty cycles for the motor controller to largely cover the set of possible ones the wheels did not gradually increase their speed. They rather turned out to run at precisely three different speeds (excluding no movement). At low duty cycles some of the gear started to wiggle, which made it seem like the motor was not able to overcome the fraction of it. At higher power that phenomenon is unclear to me.
Nonetheless this is not a big issue, since the navigation algorithm would use discrete predetermined velocities anyway and the different speeds are spread equally.
There is no working built in sensor which could measure the speed of the car. So to determine these velocities a different approach was needed. Since it is not necessary to determine these values very precisely and the Lidar sensor is able to produce accurate distance measurements it

can be used to attain these data. When driving the car towards a static target, the Lidar sensor puts out decreasing distances. With the difference between their values and difference between the data time stamps it is possible to get an approximation of the velocity of the car. The results of this test were $2\frac{km}{h}$, $4.5\frac{km}{h}$ and $7\frac{km}{h}$.

2.6.2 Steering

Unlike as for the motor control, the steering servos seem to be able to put the wheels into continuous angles. This was also tested by gradually increasing the input value and observing the effect on the wheels. Here the assumption that the input duty cycles maps linearly to the actual wheel angle was made.
As described earlier it was originally planed to use the principal of Ackermann steering to deduce the needed steering angle to drive a certain path. This method turned out to deliver bad results. So to get information about the actual behavior of the car in relation to its input values the smallest circle it can drive was measured. At a input steering angle of $15°$ the car drove a circle with a diameter of $2.77m$. This value will later be used to compute the steering angle.

3. Software

In this section all the needed knowledge about the environment the navigation algorithm is running in will be provided. This reaches from the operating system and setup of the Raspberry Pi to the routine executed by the Arduino nano and the from that resulting protocol used to handle the communication between those two. Also the ROS (Robot Operating System) environment, the algorithm is embedded in and which provides several useful libraries, will be introduced.

3.1 Arduino nano

To start things at the lowest level the routine of the micro controller "Arduino nano" will be analyzed. In the hardware section it was already established that it acts as the interface between the Raspberry Pi and the car.
Originally it was planed that this communication interface would be provided by the student Quentin Schlitz, who worked on it in the lab course "Hardware/Software Co-Design with a LEGO car". Upon several errors occurring during testing Quentin Schlitz, Justin Iszatt (another student attending this lab course) and I ended up spending quit some time setting up a working version. In the end a slightly different version of code was used for this project than what was generally provided for other projects on the car.

3.1.1 Arduino IDE

The company "Arduino LLC" marketing the Arduino hardware also provides a free integrated development environment (IDE) to comfortably write and upload code to their products. For this project the "Arduino 2:1.0.5+dfsg2-4" version was used. It is important to note that this IDE was used on a laptop to upload the code to the Arduino nano and only later the micro controller was connected to the Pi to then run its routine.

3.1.2 Routine

Arduino also provides a library that establishes a serial interface on the board. This interface can be used to read data out of a input buffer byte per byte. This buffer has a size of 64 bytes and works according to the round robin principle. Because the routine is executed slowly, this causes the buffer to override old input data if commands arrive at an high rate. That leads to

uncontrollable behavior of the car. So the input buffer needs to be flushed every time a command got read out.

3.1.3 Protocol

Since the input buffer of the serial communication is read out bytewise, a command understood by the Arduino is a chain of characters. The formatting of a accepted command is: "||||*speed*;*angle*;". The placeholders *speed* and *angle* should be substituted by the actual values the car should perform. The feasible values of *angle* hereby lie at any value between -15 (corresponding to maximum left steering) and 15. Note that the values should rounded to a moderate amount of decimal places to prevent the input buffer from being fluted. *speed* however should only be substitute by natural numbers. Since the car will only drive at discrete speed values anyway (cf. Hardware testing/Motor) the reasonable assignments are:

speed	car behavior
0	$0\frac{km}{h}$
10	$2\frac{km}{h}$
14	$4.5\frac{km}{h}$
18	$7\frac{km}{h}$
b	braking

3.2 Raspberry Pi

The software setup of the Raspberry Pi needs to be suited to run the navigation algorithm on it. To run it ROS needs to be fitted onto it as well as a Linux based operating system. Also some more configuration is needed to deal with communicating to the Lidar, the Arduino and external equipment like a laptop.

3.2.1 Ubuntu Mate

The bare Raspberry Pi does not have a specific operating system running on it. So it can be used with any OS compatible with the hardware provided by Pi. Since the supported platforms of ROS are all Linux based that should also hold for the platform running on the Pi. The generic OS to use on the Raspberry is called "Rasbian" and is based on "Debian Jessie" witch is also supported from ROS. Though it turns out that to get ROS running on Rasbian a installation from the source code is needed. To have a more convenient way of setting up the Pi the Ubuntu based operating system "Ubuntu Mate 16.04" was chosen. Using Ubuntu Mate the Advanced Packaging Tool (APT) can be used to easily install the up-to-date version of ROS.

3.2.2 General setup

The Pi provides Wi-Fi as well as a Ethernet port. The Ethernet port is needed to communicate and read data of the Lidar sensor, so the best option to communicate to the Pi from an external

device is by using a wireless network and tunneling onto the Raspberry by ssh. This is essential to set up the Pi and get data when testing, since the Pi is not used in an environment where it can be attached to peripherals like a keyboard, mouse and monitor. To conveniently use the wireless communication the Pi was given a static IP address in a passwordless WiFi network.
The IP address of the Ethernet port is set to always be "192.168.0.12", because the used Hokuyo Lidar sensor is adjusted to only communicate with that IP. The Lidars IP address itself is statically set to be "192.168.0.10".

3.3 ROS

The Robotic Operating System (ROS) is a framework created to more comfortably write robot applications. It also provides several commonly needed libraries like for example image processing tools, hardware drivers and path planing tools. Furthermore there exist plenty of debugging tools, like data visualization and logging tools, to make complex system more accessible. The decision to used it for this project was obligatory, because a convenient driver library exists for the used Lidar sensor and thus providing a way read out already interpreted Lidar data.
In this section the basic functionality of ROS as well as the used libraries will be introduced. Also the core structure of a ROS software will be explained. This will be the cornerstone of understanding the implementation of the navigation algorithm, since it uses these functions during build- as well as during run time.

3.3.1 ROS software structure

The basic structure of a ROS software is that there are simultaneously running individual programs. Each of witch represents an independent functionality. These programs are called "nodes". They can be combined as needed in different projects. Their size can range from very small programs handling just one specific task, to larger ones with a variety of abilities. For instance the whole navigation algorithm of this project is represented by only one node whereas the communication of Pi to the micro controller of the car is handle by one short python script also considered an individual node.
So during run time a ROS projects consists of multiple nodes. There needs to exist some way of managing all of those and also the communication between them. The system used by ROS is called "roscore". It handles creation and destruction of nodes as well as parameter configuration and the logging of general and debug information.

3.3.2 Node communication

Since these nodes are basically independent programs, there needs to exist some way of communication to join their functions together. ROS provides several alternative approaches of doing so. These can also be combined to free extend in a single project. The primary way chosen for this project is using the so called "topics". They can be compared to objects in object oriented programming. Even so they do not embody any functions, they are containers that hold data. Much like structs in c++. There exist basic data types provided by ROS that can be freely

combined to form a certain type of topic. ROS also already provides a big variety commonly used topic types, but they can also be created by the user to fit a more specific problem.
To communicate internally or between multiple ones, a node can publish such a topic. This publication is then handled by the roscore. It notifies all nodes interested in the topic and passes its data on for them to work with. An advantage of this system is that a node witch releases a topic does not need to know about witch other node will use this data. For example in the case of debugging a visualization node could be set to subscribe to a topic that normally is just used to pass on raw data and by that giving insight the way the data evolves during its processing. Then later at a point of performance optimization this node can be turned off again.

3.3.3 Launch files

ROS nodes can be reused for any project they are needed. So a specific ROS project defines itself exclusively by the kind of nodes used and their configuration. To have a convenient method of starting up such a project, launch files are used. These are files the ROS system is able to interpret and witch tell the ROS launching tool what nodes to start with what kind configuration. Most nodes provide a range of parameters that can be used to adjust them for a particular project.

3.3.4 Lidar driver

To read data from the Hokuyo Lidar in this project a ROS node of the type "urg_node" is used. It is a open source software and can be downloaded from a Github repository. It is specifically fit to work with Hokuyo laser systems. More precisely it acts as a wrapper for the URG library used for products of the Hokuyo Automatic Co.. When configured with the IP address of the used Lidar system, assuming the Ethernet port is in the same network, the node publishes the scan data as a ROS topic at the rate of scanning. This topic can be used by other nodes to work with the data. The information gets delivered in an array of distances where each of the entries corresponds to a specific angle the measurement was taken.

3.3.5 Stereo camera

As mentioned earlier the stereo camera attached to the car can be used as a secondary data gathering device. Unfortunately the computation of the depth interpretation very recourse intensive. This leads to it being not performable on the Raspberry Pi at a decent update rate. Though when run on a laptop the it adds to the navigation algorithm.
To get the frame data of the two cameras the "uvc_camera" package is used. It also wraps that data together with calibration data into topics used by ROS to transport images. It provides a node that retrieves and synchronizes images from both cameras.
From this image data stereo images can get computed. This is done by the "stereo_image_proc" package specified for this task. It is of the "image_pipeline" stack of ROS that is covers a variety of image processing tools. This package takes the left and right images of the cameras and also their calibration data as input and from that derives a set of points in 3-dimensional space. These points correspond to the image points for witch it was possible to determine a position. This of course is a subset of all image points. To fulfill this task the "stereo_image_proc" node

uses the OpenCV library. This is widely spread library providing functions for image processing tasks. Note that to run this node a functioning version of OpenCV has to be installed on the system.

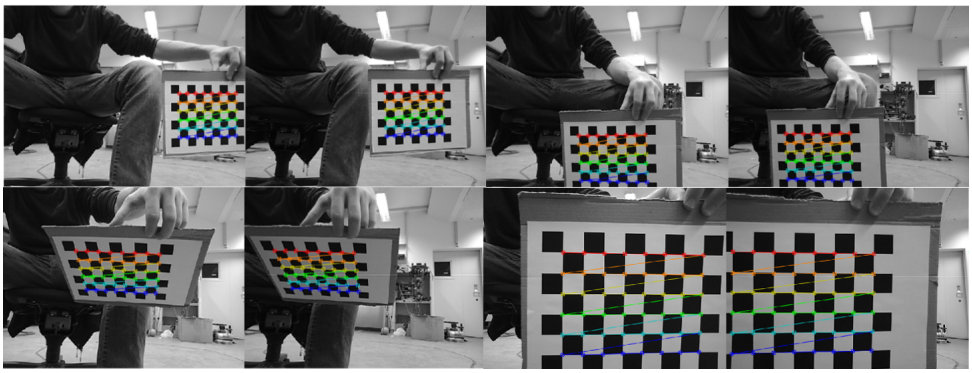


Figure 3.1: Four sets of images used to generate calibration data of the cameras.

For reliable stereo image data first a calibration of the cameras needs to take place. The calibration data used by this project was also produced by a ROS package called "camera_calibration" witch is also part of the "image_pipeline" stack. The calibration data gets generated by moving a chessboard in front of the cameras. This data later gets used to remove lens distortion and to deduce the relation between the cameras.

3.3.6 Car interface

There needs to some way of handling the communication between the Arduino board and the system running in the ROS environment. A simple python script that acts as a node in the ROS system is used to cope with the oddities of the cars control inputs. It takes a target speed in $\frac{m}{s}$ and a steering angle in degree out of a ROS topic. From that it generates the message needed to communicate those values to the micro controller. This script also handles the serial communication.
The time it takes for to run one cycle of the entire algorithm needs to be kept below a certain threshold. This is to ensure the car will not crash even though data is provided that would prevent this. The threshold used is 2hz. This means if no new steering control has been published the car is obligated to use its brakes. This is also handled by this node.

4. Tentacle Algorithm

The Tentacle algorithm was originally used in the C-Elrob (European Land Robot Trial) 2007 and the 2007 DARPA Urban Challenge. It was created by the Department of Engineering of the University of Federal Armed Forces Munich. In combination with a GPS path planning tool it navigated a SUV to along a street and through ruff terrain. Hereby the tentacle algorithm took the role of real time obstacle avoidance were it uses the general path obtained by the GPS tool to prioritizes directions for the car to take. This algorithm is the main navigation tool for this project. Its functionality and implementation will be cover in this section.

4.1 Core idea

Assuming complete knowledge of the nearby surroundings the algorithm evaluates a variety of possible paths to take. Each of them correspond to a fixed steering angle. This leads to each path having circular shape were the diameter directly dictated by the steering angle it corresponds to. Of course most of the time driving a full circle is not reasonable so just a sort section of it is considered. If visualized this leads to the paths looking like tentacles laid out in front of the car. These tentacles are then evaluated by the means of different parameters. Depending on properties of a given situation these can vary and be adjusted. After a tentacle got chosen the car will be induced to follow this path until a new set of tentacles on the now slightly different position (and orientation) will be evaluated. Note that a path will only be followed for a short portion rather than to its end.
It is important to emphasize that the tentacle algorithm does not track the cars position in space over time. At any point it detects the surrounding with the provided sensors and then follows the best path according to only that information.

4.2 Initial point

Since there are several differences between the original project, the tentacle algorithm was used in, and this one, it was necessary to have some adjustments to it. This disparities reach from less sensors used as input over the driving environment of the car.
Similar to this project originally a Lidar system was used to obtain an occupancy grid of the surrounding area. Though the Lidar system used was much more potent, because it had a total of 64 beams scanning the area for a 3-dimensional detection.

4.3 Occupancy grid

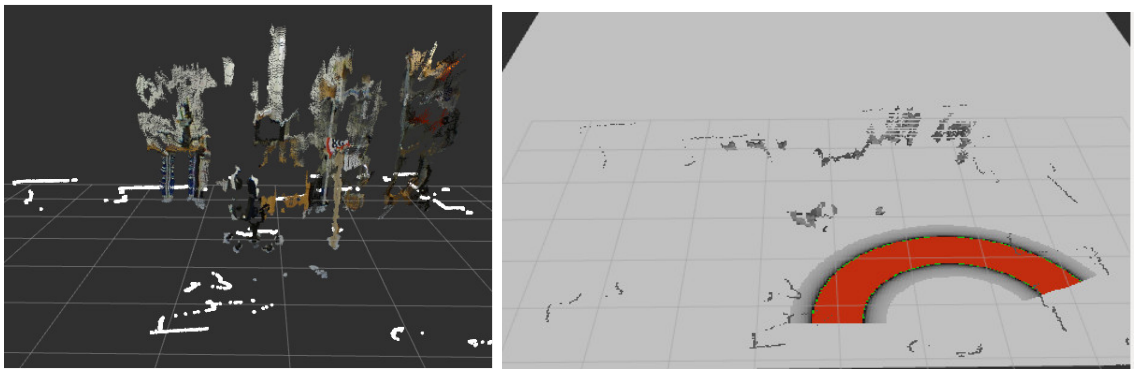


Figure 4.1: Left: Stereo point cloud and laser scan (white) data as input; Right: Occupancy grid with obstacles (grey) and preferred driving lane (red).

The occupancy grid is a simplified description of the surrounding of the car. It cramps down all the possible information to just the ones needed for obstacle avoidance. This is done by a 2-dimensional grid of same sized cells. Each of witch describes a certain part of space. The grid is interpreted to be top-down view, so a single cell can be seen as a square one the ground. If any part of an object that would disturb driving lies in this square, the whole cell is considered undrivable.
Adjusting the cell size leads to more precise driving in case of smaller ones. Though this comes at the cost of higher computational power needed. The side length of the squares chosen is for this project is approximately $2.286cm$. This results in 43.75 cells for one meter length. The whole occupancy grid also has the shape of a square with 525 pixels in length and width. Thus the resulting side length of the grid is $12m$.
Internally the occupancy grid gets represented by an array of bytes. So it would be possible to have different ratings of grid cells. In the original version of the tentacle algorithm this is used to describe the roughness of the terrain of a cell. Since it is not possible to gain useful information about properties of the ground here the algorithm only distinguishes between drivable and undrivable grid cell.
Since all sensor measurements are made from the car, the grid describes position of obstacle in relation to the car. This results in the car always having the same position and rotation on the occupancy grid. Here it will be considered to drive along the x-axis. Because the car obviously drives a forward motion information about obstacles in front of it are more important. Thus the car is placed on the left side of the grid. This position also serves as the origin of the tentacles.

4.3.1 Interpret Lidar data

The data of the Lidar sensor can be easily transformed into a occupancy grid. The data is delivered by an array of distances where each data point corresponds to a specific angle the data was taken at. The angle off set between two consecutive data points is 0.25°. This leads to 1080 point at an angle range of 270°. For each of them the Cartesian coordinates get computed

and the corresponding position is marked on the occupancy grid. This is a very easy and fast method, but there are possible faults that can appear.
Because in this implementation there is no interpolation between the data point happening, there could be gaps on the occupancy grid appearing in a detected continuous object. If the distance between two scan points falls bellow $2.286m$(grid cell side length), this can no longer happen. This is the case at a distance smaller than 5 meters, which gives enough time to react. Also the Lidar sensor only retrieves data from one height level there are still objects existing that will remain undetected but block the driving lane. This can either be the case of a flat object lying below the scan (e.g. office chairs) or obstacles being above the scan. These cases basically can not be avoided by solely using the Lidar system as data input. For this another device like a stereo camera would be needed.

4.3.2 Interpret stereo data

If stereo data is obtained it can be used to refine the occupancy grid. This is assuming that there also exists data of the Lidar sensor in the setup. Since the stereo and the laser data get published by independent threads, the data first needs to be synchronized. This can be done by "message_filters" provided by ROS. The raw laser data normally should get published at higher rate than the stereo data witch first takes some processing. In this implementation only if new laser and stereo data is available a new occupancy grid gets computed.
The laser data first gets transferred to the grid. This is done by the same method as already presented (cf. Interpret Lidar data). In contrast to the laser data not every data point of the stereo data is of interest. There are some data points representing the ground the car is driving on as well as some points of objects that lie above the car. Because the location in 3-dimensional space of every point is known, these points can be filtered out by knowing the position of the camera in relation to the ground and also the height of the car. The remaining points can be directly transferred to the occupancy grid.

4.4 Tentacle computation

A set of tentacles are used the evaluate possible driving paths of the car. They will be called "speed sets", because each of them corresponds to a certain velocity of the car. A speed set for a higher movement speed of the car will hold less curved driving lanes/tentacles. This is done to prevent the car from driving to sharp corners at high speed. Each tentacle is represented by a set of points on the occupancy grid. For better performance of the algorithm multiple speed sets are computed beforehand. This section will cover the computation of the tentacles and by that also who to derive their geometry.

4.4.1 General

There is a fixed number of tentacle sets. Each of these sets corresponds to a fixed velocity of the car. The tentacles in a set corresponding to a faster velocity are less curved. In each of those

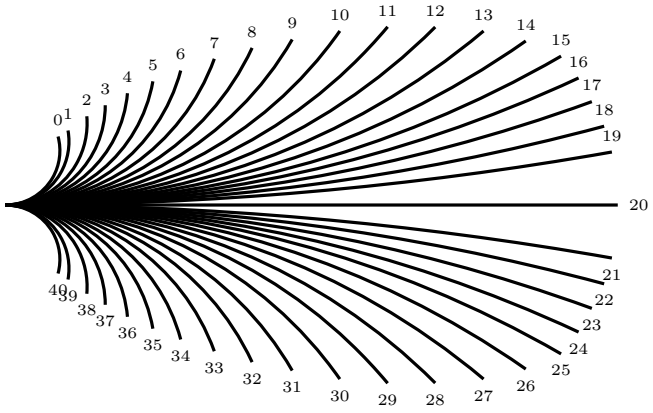


Figure 4.2: Possible shape of a speed set. Highlights internal enumeration of driving lanes/tentacles.

sets there is fixed number of tentacles. Every tentacle consists of a various amount of points depending on its size and curvature.
Here there are 41 tentacles used in three different speed sets. Those numbers are variable and can be adjusted to fit different problems. Since during the hardware testing it turned out that the motor controller is only able to drive at three speed levels, it made sense to also calculate one set of paths for each of those. The number of 41 tentacles was picked, because the steering of the car is not as precis that it could drive at a higher number of different angles. Having a lower number of tentacles allows for better performance when picking the best one.
For notation each set will get enumerated. Set number $i \in \{0, 1, 2\}$.
Each tentacle in will get a number $k \in \{0, 1, ..., 40\}$ 4.2.

4.4.2 Shape

Each tentacle is a circular arc. So it can be defined only by its radius and its length. Note that a smaller radius leads to a higher steering angle and length is a factor of who much of the lane will be considered when evaluating. In the following the process of computing the properties of a single tentacle used in the presented algorithm will be illustrated. The formulas used to compute these two values are based on the paper referred to earlier. Though some of them got tweaked to fit this project.
Given a tentacle with number k of speed set i. First a base length l_i gets computed from the set number i. For tentacle sets corresponding to a higher velocity the tentacle should be longer. This results in the algorithm checking for collision in a bigger distance.

$$l_i = 3m + 3m \cdot \frac{i}{3}$$

This value serves as the smallest length of a tentacle in a given speed set. A less curved tentacle should have a smaller length. To get the exact length $L_{k,i}$ of a tentacle k the following formula is used.

$$L_{k,i} = \begin{cases} l_i + 5m \cdot \sqrt{\frac{k}{20}} & 0 \leq k \leq 20 \\ l_i + 5m \cdot \sqrt{\frac{40-k}{20}} & 21 \leq k \leq 40 \end{cases}$$

Because of the square root, this formula results in strongly curved tentacles having substantially smaller length and straight tentacles being similar in length. Driving along a path with a high steering angle for a longer time would results in the car going circles. This is not constructive for reaching a goal, but should rather be done for a short wile to avoid obstacles. So narrow curves are not evaluated for a long distance.
Similar as for the length of tentacles, to compute the radius also a base radius r_i gets computed for a whole speed set. This radius corresponds to the smallest radius of the set and is dependent of the sets base length l_i and speed set number i.

$$r_i = \frac{l_i}{0.375 \cdot 2\pi \cdot (1 - \frac{i}{3})}$$

This formula may not seem intuitive. Though when rearranged to show the relation of length l_i and perimeter u of the corresponding circle its origin gets clear.

$$0.375(1 - \frac{i}{3}) \cdot u = l_i$$

The most curved tentacle of the slowest speed set is a 0.375th of a full circle. This fraction decreases for higher speed set numbers. The radius $R_{k,i}$ of a tentacle with number k is obtained by:

$$R_{k,i} = \begin{cases} r_i \cdot 1.2^k & 0 \leq k \leq 19 \\ \infty & k = 20 \\ r_i \cdot 1.2^{40-k} & 21 \leq k \leq 40 \end{cases}$$

From these given formulas the length and radius of every tentacle can be computed. By that the basic shape is defined.

4.4.3 Area

Each tentacle represents a possible driving lane of the car. So just the general shape of one not sufficient to draw it. It also needs to have a certain width to ensure the car can actually drive along it. The area needed to fit the car will be called "classification area". If an obstacle lies in this area, the car is assumed to hit when following the tentacle to its full extend. The width of the classification area is dependent to the width of the car. The width of the area here will be $60cm$. This the width of the car ($55cm$) in addition to some safety space ($5cm$).
In addition to the classification area also a second bigger area is defined for each tentacle. It will be called "support area". It is used to have an additional factor during the tentacle evaluation. This is to avoid driving close by objects even though more space is available. Its width can be adjusted to have the car circumnavigate objects more widely or closely if needed. Here it will be $120cm$.

4.4.4 Points

The area of a tentacle is described by a set of points which lie in the occupancy grid. Internally the algorithm describes each point in a set by the 4-tuple (o, w, d, f). The offset o of a point

describes its exact position on the grid with:

$$o = y \cdot 525 + x$$

Each points can either lie in the support area or the classification area of a tentacle. The flag f describes this distinction. The distance d of a point is also of interest. Hereby d is not the direct distance to the car. It rather describes the distance to the car following the path of the tentacle (cf. Figure 4.3). Each point is also given a weight w which describes its importance

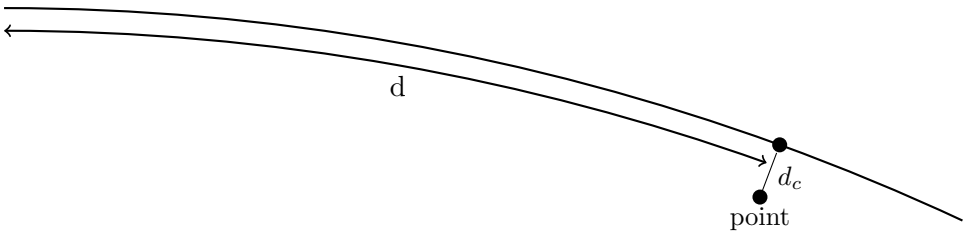


Figure 4.3: General distance value d and distance to center d_c of point on tentacle area, but not on tentacle center line.

if an obstacle lies on it. It values lie between zero and ten where ten are points of the most significance. The value is obtained from the distance d_c of the point to the center line of the tentacle (cf. Figure 4.3). More precisely it is described by the formula:

$$w(d_c) = \begin{cases} 10 & d_c \leq 0.3m \\ \frac{10}{1+30\cdot(d_c-0.3m)} & d_c > 0.3m \end{cases}$$

All points of the classification area have maximum wight. A value for d_c bigger than $0.3m$ implies the point not being part of the classification area. A hyperbola is used to describe the weight values of points on the support area.
For generating the point set of a tentacle first its length and radius values get computed according to the previous section. From these values the set of points can be generated. This is done by running through all grid points and checking for each point if its position lies inside the classification or support area. If this is the case, all properties (offset, weight, distance, flag) of the point get computed and it gets added to the set of points.

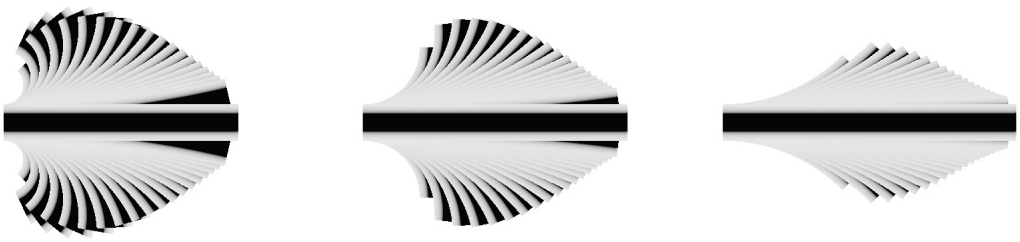


Figure 4.4: The exact shape of tentacles used in this project. Darker color corresponds to bigger point weight.

4.4.5 Evaluation

To get the best driving path the algorithm compares all tentacles of a speed set corresponding to its current velocity. This is done by running through a tentacles set of points and checking the occupancy grid according to the offset of a point. Upon finding obstacles a classification value gets computed for the tentacle that is dependent on the location of these objects. The range of this value lies between one and zero. Hereby a low value gets preferred. If no obstacles have been found inside the area of the tentacle, it gets assigned to a classification value of zero. The classification value has multiple components to it witch define direct collision and driving through narrow space.

4.4.5.1 Distance value

The first component is called the "distance value" or v_{dis}. It gives insight on the distance the car can follow on a tentacle before colliding with an obstacle. Consequently for its calculation only the classification area gets regarded. First the the closest point that is blocked by an object gets located. The closest point is defined by the distance value defined for each point. The smallest value corresponds to the closest point. The clearance value then gets derived of the distance d of this point. This is done by first calculating the constant c_{dis}.

$$c_{dis} = \frac{\ln(3)}{d_{0.5}}$$

The parameter $d_{0.5}$ represents the value at witch distance the distance value will be 0.5. In this project this distance is $d_{0.5} = 5m$. So if the closest obstacle on the classification area lies in a distance of 5 meter, the distance value will be $v_{dis} = 0.5$. The distance value d of a point in combination with this constant then gets used to derive the actual clearance value v_{dis}.

$$v_{dis} = 2 - \frac{2}{1 + e^{-d \cdot c_{dis}}}$$

4.4.5.2 Clearance value

To use free space more efficiently the classification value also has a second component. It will be called "clearance value". Like the distance value it ranges from zero to one where a lower value will be preferred. To compute the weighted average of all grid points will be constructed. Since the distance to obstacles should also have an influence on the clearance value, first the distance value of a point containing an object will get computed. Let the set $\mathcal{D} = \{v_1, v_2, ..., v_m\}$ be the set of distance values of all points on the tentacle with a grid value greater than zero and let $\mathcal{W} = \{w_1, w_2, ..., w_m\}$ be the weight of the corresponding point. Note that the values in $\mathcal{D}$ are the distance values derived by the method above and thus also range from one to zero. From that the average can be computed.

$$avg_{clear} = \frac{\sum_{i=1}^{m} v_i \cdot w_i}{\sum_{i=1}^{m} w_i}$$

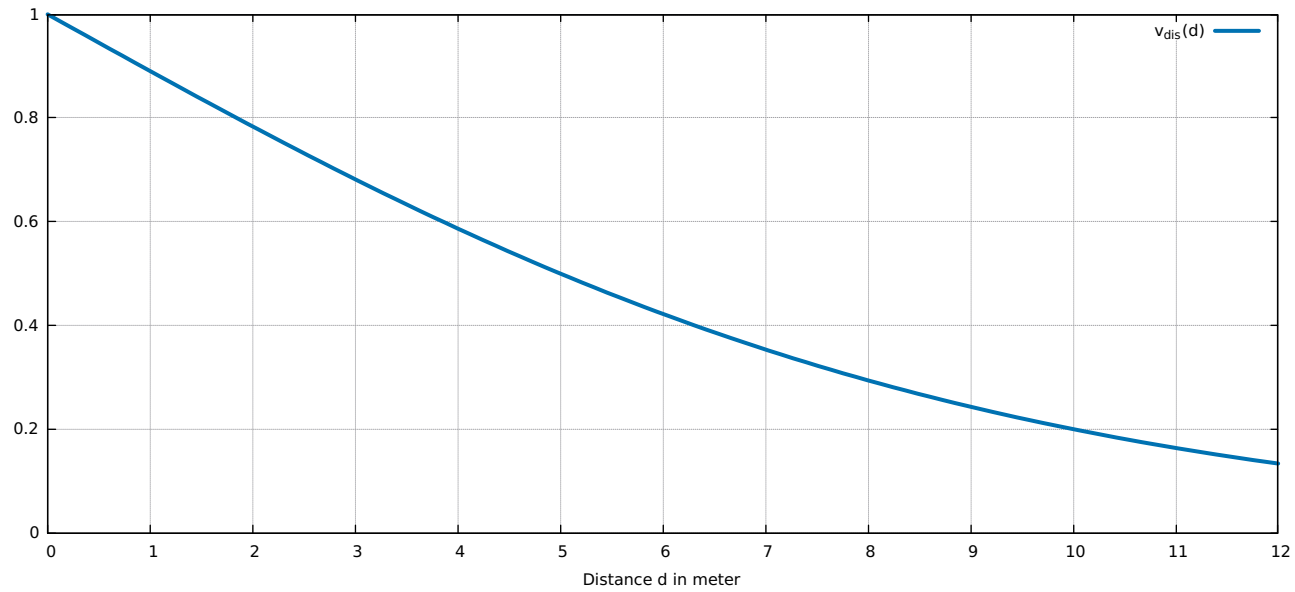


Figure 4.5: Distance value

To now deduce a value in the range of one to zero from this average a similar method as for the distance value will be used. First a constant c_clear will be evaluated.

$$c_{clear} = \frac{\ln(3)}{c_{0.5}}$$

The parameter $c_{0.5}$ represents the input average at witch the clearance value v_{clear} become 0.5. In this implementation $c_{0.5} = 0.8$ was picked. From the clearance constant c_{clear} and the weighted average distance of all points a_{clear} the actual clearance value will get computed.

$$v_{clear} = \frac{2}{1 + e^{-a_{clear} \cdot c_{clear}}} - 1$$

4.4.5.3 Classification value

Now the distance value v_{dis} and the clearance value v_{clear} can be combined by linear combination to attain the classification value v_{class} of the evaluated tentacle.

$$v_{class} = a_{dis} \cdot v_{dis} + a_{clear} \cdot v_{clear}$$

Here these two values can be weighted against each other by the the values of a_{dis} and a_{clear}. Though in this implementation both are weighted the same by $a_{dis} = a_{clear} = 0.5$.
In the original paper it is proposed that a third value can be computed to also rate the direction of a tentacle in relation to a path attained from a GPS module or a different method. Since that was not available the third value was cut for this project.

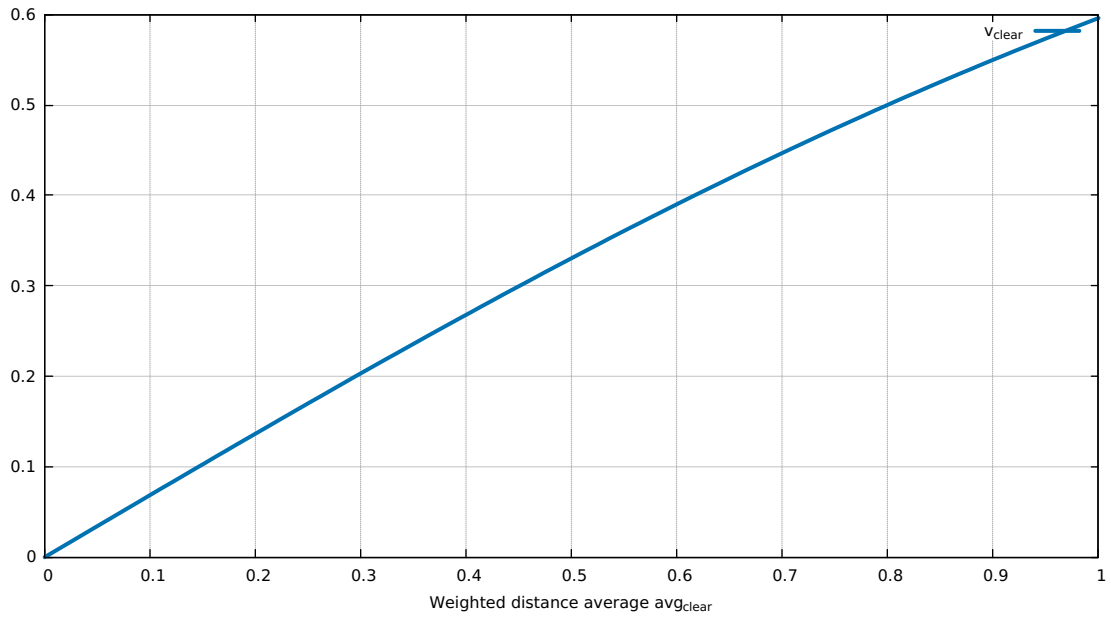


Figure 4.6: Clearance value

4.4.6 Crash distance

If an object gets detected close to the car on a certain tentacle the car should be obligated to brake if taking that direction. To handle this case the crash distance l_{crash} is introduced. This is the distance the car needs for slowing down and still remaining in a safe distance to the object. Since the different tentacle set are related to a different velocity of the car, the crash distance varies between them. It is composed by a safety distance l_{safe} and the distance it takes the car to stop l_{stop}. The stop distance l_{stop} can be computed from the braking deceleration a_{brake} and the current velocity v of the car.

$$l_{crash} = l_{safe} + l_{stop} = l_{safe} + \frac{v^2}{2 \cdot a_{brake}}$$

The explicit values used here are a braking deceleration of $a_{brake} = 1\frac{m}{s^2}$ and a safety distance of $l_{safe} = 0.8m$. The braking speed has a very low value, because the car does not have functional brakes attached. Also note that the security distance l_{safe} should be higher than the distance the car can travel during one computational iteration of the algorithm. So these values need to be not only adjusted to the car and the environment its driving in, but also to the hardware performance.
When during evaluation an obstacle is detected with a distance smaller than the crash distance the tentacle will be considered a braking tentacle. These tentacles will be handled differently.

4.5 Choosing a tentacle

After all tentacles of a set have been evaluated the preferred one can be picked. Generally the tentacle with the lowest classification value gets chosen. But there are also different parameters that are taken into account.
First of all a tentacle that is considered a braking tentacle will get avoided at all costs. So unless every tentacle of a speed set is a braking tentacle, a non braking one gets picked. Though if that is the case the tentacle with the lowest distance value gets picked. That results in the car taking the direction with the longest distance to an obstacle. Needless to say in that case a signal to perform braking is send to the car.
Sometimes there exist tentacles with very similar or even identical classification values. If this difference goes below a certain threshold for two tentacle they are considered to be equal in preference. For this project a rather high threshold of 0.1 got chosen. Of two equal tentacles the one closer to the current steering angle of the car gets picked. This is to avoid a frequent change of direction by the car. The high threshold value is motivated by the fact that the direction of the car is taking does not have an influence on the classification of tentacles.

4.6 Deduce steering angle

To deduce the steering angle necessary to follow a tentacle, a simplified version of the car is taken in account (cf. Figure 4.7). This model is deduced from the Ackermann steering principle. A simple model is sufficient, because the steering servo is unable to adopt an exact angle anyways.
By geometrical considerations it can be easily established that the angle α of the triangle

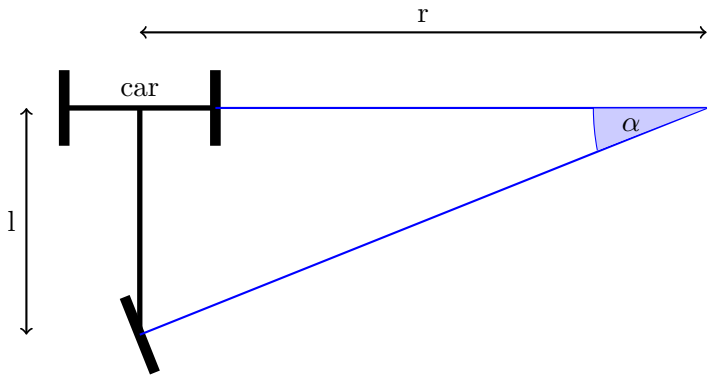


Figure 4.7: Simple model of the car steering. Angle α corresponds to the steering angle. The car drives on a circle with radius r and has a axis distance of l.

corresponds to the steering angle. The radius r is defined by the tentacle and the distance between axis of the car l is a general value for a certain setup. From this the steering angle can get computed.

$$\alpha = \tan^{-1}(\frac{l}{r})$$

During testing it turned out that the actual axis distance ($0.55m$) of the car would not lead to the car following a tentacle correctly. So the actual value picked here is $l = 0.375m$. So this approach rather delivers the dependency between steering angle and radius, than the actual physically correct model. Though with adjusted parameters it serves it purpose.

4.7 Car control

Suppose the car is driving at some speed. From the that and the sensor data an optimal steering angle can get derived. If it is not necessary to use braking, it is unclear at what driving speed the car should continue. In this project an intuitive approach has been chosen. When the car drives at a flat angle into a free space it should accelerate.
These values can be derived of the tentacle that serves as a driving lane. Its classification value represents the space available and its radius the steering angle. If at some point the classification value of the picked tentacle is zero, this means there are no obstacles close to the driving lane. In combination with a big radius, witch corresponds to a low steering angle, this is a case of speeding up the car. The target speed will be increased to match the next speed set.
For slowing the car down the inverse case is used. A high classification value or a narrow turning angle imply that there are objects to be avoided. So before a case of braking is inevitable the car is slowed down by one speed set.

5. Performance

There is a lot of computation needed to get from the raw data to the output of the car controls. All of it needs to be performed by the Raspberry Pi, witch does not have a powerful processor. In this section a overview over the whole system is given with some insight on the performance benchmarks and optimization.
Though the Pi has a low frequent ARM processor it features four cores. This results in a great gain of performance if the program is using multiple threads. In the optimal case a critical region can get is processed by four threads. The goal will be to find such a region and try to split its processing up.

5.1 General structure

First it is important to get an overview of the environment the tentacle algorithm is embedded in. Since it is embedded in the ROS system, it can be illustrated by showing all nodes processing data and the communication between these nodes Figure 5.1. The edges in the graphics represent the topics nodes are communicating with. Note that each of these nodes is executed by a different thread. For example the tentacle algorithm can be still evaluating tentacles when there are already new images captured from the cameras. In the following this will get neglected, since we are interested in the time it takes to process one data set. So the only small improvement gained from the generic multithreading of ROS is the simultaneous fetching of the raw data by the scan node and the camera node.

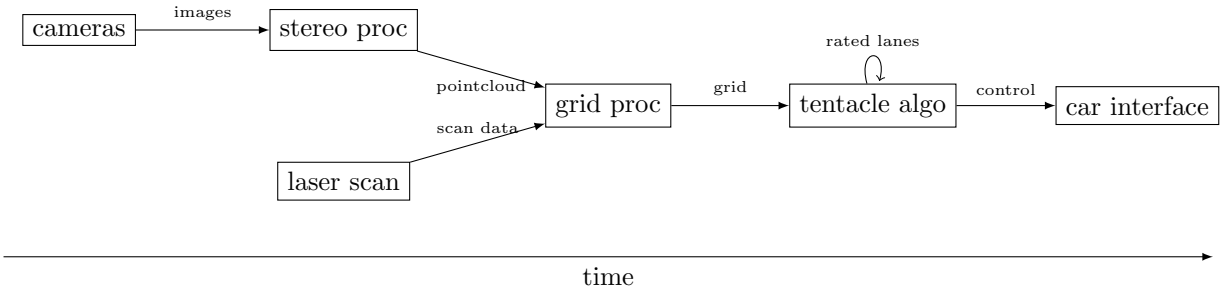


Figure 5.1: Data flow between all ROS nodes. Raw data get collected by camera and scan data node. The car interface node communicated with the car.1

The nodes taking a long time processing the data are the stereo image processing node and the tentacle algorithm node. Since the stereo processing is a external library it is not recommended to try and improve its performance. So the only bottle neck that can be optimized is the tentacle algorithm.

5.2 Tentacle algorithm structure

To improve the the tentacle algorithm first its structure will get reviewed.

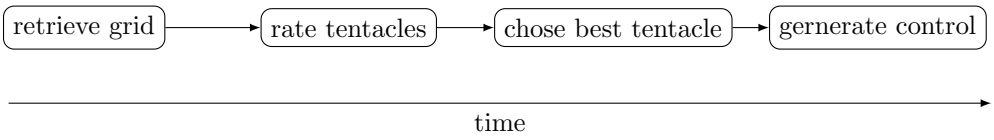


Figure 5.2: Single thread version of the tentacle algorithm.

Retrieving the grid is a rather formal measure to make it available for algorithm to work with. This is achieved with very fast. After that all the tentacles of the appropriate speed set get evaluated by the method presented earlier. To do so the algorithm runs through every point of all tentacle and checks its position on the occupancy grid. This is computational heavy since a single tentacle can feature up to thousand points. After all tentacles have been evaluated the best needs to be chosen and from it the car control needs to get retrieved. These are tasks that are fast and need to be handled in the same thread.

5.3 Optimization

So to improve performance its advised to focus on the rating of the speed set, since tentacles can be focused on independently and during this process there are only look ups on the grid happening. So this part of the algorithm can get parallelized.

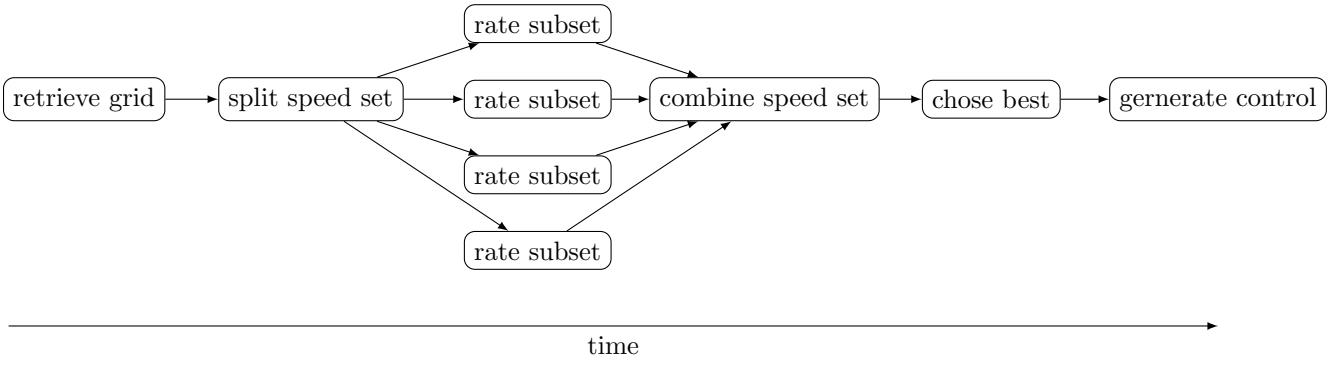


Figure 5.3: Multithread version of the tentacle algorithm. Here four threads are used.

The whole speed set gets split up into a number of exclusive subsets that corresponds to the number of threads that should be used. The tentacles of each of those subsets then get evaluated by one thread each. The source code was written in way so the actual count of threads used could vary. For the Raspberry the count intuitively was set to four, though different arrangements were tested.
Internally choosing the optimal driving lane out off a already rated speed set is done by a separate thread as well. This does not speed up the treatment of one data set, but in a sequence of data sets arriving this might reduce idling of cores.

5.4 Stereo processing

The stereo processing turned out to demand the most amount of computational power. So to actually get it to a level were it can be handled by the Raspberry Pi the resolution of the retrieved images was lowered. This is possible, because the PS3 Eye cameras provide this feature. The resolution of 640×480 was lowered to 320×240. This results in the number of pixel are only a fourth in comparison to the original resolution. This comes at the loss of quality of the stereo image. So the location of objects is not precis when using low resolution pictures.

5.5 Performance testing

The performance of the software was tested in different setups to get an better insight on software parts. Here some of the data retrieved and their interpretation will get listed. To rate the performance the publishing rate of topics is regarded. This represents the number of data sets that get finished in one second. So a high publishing rate means the data is computed faster. Here the rate raw data gets produced is high, since there is a low amount of computation needed. It make sense to restrict the rate data gets produced if not all of it can be used. It happens that the input data of a node is having a higher rate than its output data. In this case data is lost and unnecessary computations have been made generating the data.

5.5.1 Tentacle algorithm

The tentacle algorithm part of the software was benchmarked in a setup were only the Lidar scanner and grid generator was used. This was done, because retrieving the Lidar data and generating the occupancy grid from it does not take a lot of computation. So variation of the performance is mostly dependent on the configuration of the tentacle algorithm itself. The rate of grids generated was capped at $20Hz$. The scan data gets delivered by $40Hz$ from the Lidar sensor.

Threads	1	2	3	4	5
Scan data	40Hz	40Hz	40Hz	40Hz	40Hz
Grid	20Hz	20Hz	20Hz	20Hz	20Hz
Control	4.7Hz	6.9Hz	9.5Hz	12.8Hz	12.6Hz

The result of this is as expected. It shows that the algorithm runs best on the Pi if four threads are used internally. The generation of the occupancy grid is fast enough so it can get delivered at its maximum allowed rate.

5.5.2 Stereo processing

To test the performance of the stereo processing on the Pi it was separated from the rest of the system. Meaning only the camera driver and the stereo processing node were running. The tests aimed to compare the performance of low resolution video capturing to high resolution. Thus knowing the gain in performance at the prize of accuracy.

Resolution	640×480	320×240
Raw Images	6Hz	19Hz
Point cloud	1.7Hz	6.3Hz
Disparity image	2.2Hz	8.5Hz

The disparity image is a alternative way to retrieve the stereo information. So the version used in this project is not optimal and a performance gain is to be expected if the grid generation algorithm was modified to use disparity images as input.Also the test showed that running the high resolution capturing was not a option on the Raspberry, since the rate of data release is to low to work sufficiently with a navigation algorithm. This test already indicates that there could be improvements made to the occupancy grid generation method.

5.5.3 Combined test

So the actual goal is it to achieve a setup that uses the stereo and the laser scan data to control the navigation of the car. So as a final performance test all nodes needed to accomplish this task were started. This is also the setup presented in the "General structure" section. Here three tests were made. One running the tentacle algorithm at an optimal configuration and one at an slightly sub optimal setup with only three threads used and one at an optimal one reducing the occupancy grid size drastically to use a 252×252 cell count. The image resolution was always kept at 320×280.

Threads	4	3	4
Grid res	525×525	525×525	252×252
Scan	40Hz	40Hz	40Hz
Images	12.5Hz	12.5Hz	13Hz
Point cloud	5Hz	4.8Hz	4.8Hz
Grid	1.3Hz	0.8Hz	1.6Hz
Control	1.3Hz	0.8Hz	1.6Hz

From this data there is a lot of information to gain. First of all the raw data retrievers (Scan, Images) are not really influenced by a lot of computational power reserved by other processes, because the seem to have a stagnant release rate. That also seems to be the case for the stereo image processing. Though the release rate of the point cloud is low, it still lies above a critical point and could still be used for navigation purposes. The tentacle algorithm releases

control data at the same rate as it does receive it. This shows that its processing speed of the data is fast enough.
The critical area in the entire setup of the current version seems to the grid generation. Here the output release rate is very low even though enough data is delivered. When reducing the optimizing the tentacle algorithm it also lead to a increase of performance of this task, even though it is only indirectly affected by it. Contrary to intuition this is also the case for reducing the grid size what actually should only be a great improvement for the tentacle algorithm.

5.5.4 Result

The general result of this test is that the integration of the stereo data is not performed well by the grid generation node. This can be caused by problems based on the synchronization of the Lidar and stereo data, or more likely be caused by the amount of data points of the point cloud. A possible solution is to also parallelize the grid generation or by choosing a completely different approach using disparity images to attain data.
Also the setup when using Lidar and stereo data is currently to slow. At a release rate of 1.3Hz of control information, a safe navigation can not be assured.

6. Results

6.1 Navigation testing

To test if the tentacle algorithm actually navigates the car in a proper way a setup with only the Lidar sensor was used. This was done because the sensor delivered reliable data at a good performance rate. To rate the algorithm there were some goals defined that in theory the algorithm should be able to achieve. The car should be able to ...

- ... have no collisions.
- ... drive through narrow path ways.
- ... take sharp corners.
- ... follow a road like setup.

These are easy goals, but they are sufficient for short term collision avoidance. One should realize that there is no path planing and long term planning implemented by the tentacle algorithm. To test these goals individually test tracks were created that would showcase each of these tasks. The car was used with an identical setup in each of those tests.

6.1.1 Collisions

This is the most general of the goals. Its testing generally takes place when driving on any of the presented tracks. It basically means that the car should always take a free driving lane and if that is not possible it should stop before colliding with an obstacle. Since the car does not have working brakes the term of "not colliding" actually was interpreted as "stopping the motor timely and at most softly touch an object". A collision with an object not only can occur frontally. It is also possible to collide with the back wheels of the car when taking a sharp corner. The track that was used was a situation at witch there were objects randomly scattered around the room in a way so that the car would have to stop at some point. These object need to have a height so that the Lidar sensor can actually sense them.

Figure 6.1: Collision test track.

There were multiple tests done on this setup. The car was engaging the obstacles from different angles. In most cases the car avoided all of the obstructions by driving around them. In situations if needed stopped timely. So this test was successful.

6.1.2 Narrow path way

The car should be able to go through a narrow path way. This means going between two objects that are place in a way that the car in addition to some safety pace just fits through them. Though this should not be a situation in witch the car would fit perfectly and needs to be placed precisely in the right position to go through the passage, because that would not reflect a realistic situation. Meant by this is a setup in witch the car gets guided into a narrow passage still with some space for error. The car can hereby slow down at much as it wants if it goes through in the end. This test is used to show the quality of the tentacle algorithm to navigate a car precisely.

Figure 6.2: Left: Car in test setup. Right: Internal representation of setup.

In figure 6.3 it is showcased that the setup was first tested to ensure the car would follow through if set up perfectly. This was done to get an insight of the security space implemented. If a more narrow path was chosen the car would only drive through is the safety space was decreased.
The car was set at an slight angle to the path to make necessary to steer before going through. In the test the car managed to drive through the narrow path. Though it slowed down significantly. If the car would drive through narrow space for a longer distance, it is possible that the car does not manage to rearrange itself and stop at some point.

6.1.3 Sharp corners

For this test it is the goal to let the car drive around a corner for witch a full or almost full steering angle is needed. Here for a situation needs to be created in witch a sharp turn is the only option for the car to take. Of course there would need to be some space of error so the car does not need to arrive at the perfect position to take the corner, since the algorithm does not plan taking the corner before hand. Same as for the narrow space the car can go through at any velocity if it continues driving afterwards.

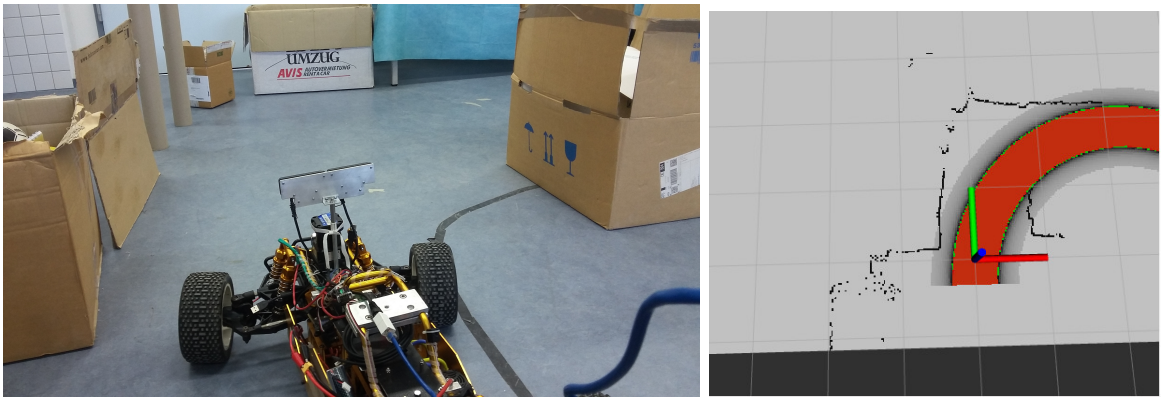


Figure 6.3: Left: Car in test setup. Right: Internal representation of setup.

Here same as fore the narrow path test first the setup was tested. So the internal representation was looked at the obstacles were adjusted to lead to car taking its sharpest turn. A sharper setup would be possible if the security space of the car would be reduced. Though this is not reasonable.
This was the first test that turned out to be unsuccessful. The car took the corner, but ended up taking it to steep so the back wheel ended up being blocked. The algorithm actually notice this fault and stopped the engine! When testing the same setup with a slightly less curved corner the car managed to drive it. The result of this test indicates that it would be necessary to make some adjustments to strong steering of the car. A possible fault could be a wrong translation of the steering angles.

6.1.4 Follow a road

This test should show that in combination with a path planning tool the algorithm should be able to adjust the cars navigation in a way so it can follow a longer path. The interpretation of a road like setup here will be a track were there is enough space to vary the driving some in combination with curves in both directions. In this case the car should avoid driving to close to the borders and also drive at continuous velocity if possible. So actually in this test there are more goal than just driving through the obstructions. It also should continue driving afterwards.

Figure 6.4: Section of road like setup. Designated route marked black.

Because of the lack of recourse the "road" borders were obstructed by objects arranged in a way so the car would not be able to fit between them. Figure 6.4 shows a section of the track with a long right turn. Before and after this turn there were two left turns. The car managed to follow the path in both direction and was using the free space efficiently.During one test the car actually managed to slip through a tight gap in the border. However the car had some issues when driving from a turn into opposite turn.

6.1.5 Note on testing speed adjustments

In the later stages of the project there were issues arising with the mechanics of the car. This lead to the car only being able to drive at a very low speed and sometimes even stopping when not obligated to. The problems were not resolved before this thesis was finished. This made it impossible to test the speed adjustments of the car control and high speed steering of the algorithm.

6.2 Conclusion

The Lidar sensor represents a powerful tool for autonomous driving, because of its accurateness and its efficient data retrieval. It can be used with a low processing device and still remain effective. Its data can be combined with those of a stereo camera to correct some of its weaknesses. Though in this thesis only a easy method was used to combine these data and there is still a lot of room to improve the gain of information by combining these sensors. Using stereo vision as a reliable data source turned out to be computational intensive. So to handle this data quickly a much more powerful device than for the Lidar data is needed.
The tentacle algorithm is an autonomous navigation approach that works exceptionally well with the Lidar data. It can serve as a light weight, but still reliable collision avoidance method. With a rather low demand on computational resources it delivered satisfying results.

6.3 Future work

There are several ways the current version of the project can be improved. The tentacle algorithm at its current state already provides several. On the one hand it is not perfectly optimized. A socked for the Lidar system mounted to the car would be of quite some value. If a fixed and stable space relation between car and Lidar would be known, the algorithm could get calibrated for a more precise navigation of the car. Also in general the algorithm would benefit from more work put into its configuration.
Another big improvement would be to redo the occupancy grid generation when interpreting stereo data. In the performance region it has been shown that the whole system would benefit if it would use disparity images as input. Also there is the possibility to generate the map using multiple threads and thus optimizing it for the Raspberry Pi system. It should definitely be possible to speed this setup up so it can work as a navigation approach.
The most interesting and promising future work would be to incorporate a GPS path planning tool. How to handle this project has been described by the paper of the tentacle algorithm. Also there have been projects on the car implementing a GPS device. Since the algorithm works well in direct collision avoidance and the GPS tool could get used to transform it into a path finding algorithm.

Bibliography

[1] James B. Abshire. Nasa's space lidar measurements of the earth and planets. In *Presentation to: IEEE Photonics Society Meeting University of Maryland*, 2011.

[2] Roger Rösch. Development and construction of an autonomous car with low cost components. Master's thesis, Technische Universität München, 2016.

[3] Felix von Hundelshausen, Michael Himmelsbach, Falk Hecker, Andre Mueller, and Hans-Joachim Wuensche. Driving with tentacles: Integral structures for sensing and motion. *Journal of Field Robotics*, 25, 2008.