Universiteit Leiden
The Netherlands

Opleiding Informatica

Robots with Vectors

Taco Smits

Supervisors:
Todor Stefanov & Erik van der Kouwe

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

31/05/17

Abstract

In this thesis, we implement and evaluate the robustness of the Vector Field Histogram. The Vector Field Histogram is a real time motion planning method that uses sensors to look around a robot for obstacles and steers around these obstacles while continuing to move towards a target. To evaluate the robustness of the method we test where the method fails and how quickly it is able to reach a target.

The method is based on the Virtual Force Field method (VFF), as some problems were found with that method and were solved with VFH. VFH itself is also a base for the VFH+ method and the VFH* method, which improve on problems found with VFH method, but they do not describe the found problems.

In order to test the method we will use ROS and Gazebo. ROS, or Robot Operating System, is a collection of tools, libraries, and conventions that programmers use to aid the writing of software for robots. Gazebo is a simulation program aimed at helping test robots.

The VFH method stores information about the environment in a grid, which is transformed into directions where obstacles are located and which direction are free. Based on those directions the VFH method steers around obstacles and towards the target.

Our implementation of the VFH method was based on how the VFH method is described in [BK91b]. The paper is not clear on certain parts of the VFH method, so we had to create our own specification for those parts. The VFH method uses a lot of parameters during the calculations, but doesn't specify how to obtain values for those parameters.

[BK91b] discusses a couple of problems found with the VFH method, but also gives solutions to these problems. However, we found even more problems that were not discussed in that paper, causing the robot to either become stuck or crash into objects in certain environments. We also describe several ways to find values for all of the parameters of the VFH method, and describe how we found optimal values for our robot.

Contents

Chapter 1

Introduction

Through the years robots have become more and more important. A few years ago the only place you could find robots was in factories, today you find them everywhere, in hotels, at restaurants and possibly even in your own home. All of these robots require software to control their behaviour. This software is made by implementing several methods and algorithms. In this paper we will implement and evaluate one of the methods used for mobile robots, the Vector Field Histogram [BK91b] method, and determine in what situations the method does and does not work.

The vector field histogram (VFH) is a method that handles real time motion planning in mobile robot systems. We will be referencing to ”the VFH method” as ”VFH”. The creators of VFH found that there was a lack of methods tackling the issue of obstacle avoidance that met their specifications. They previously made another method, virtual force field, which tackled the same issue, but found that it did not work in several situations. VFH is the improved version of this older method. All of this can be found in [BK91b]. It explains how VFH is based on the older method and how it solves the problems found. Though other motion planning methods existed back in 1991, when [BK91b] was published, the writers found that those methods required the robot to stop, either because the method did not know how to go around the obstacle and gave up, or to take measurements of the obstacle to go around it. This is where VFH shines, as it does not have to stop, because it was designed to quickly calculate the best direction. Several versions of VFH exist, and most were made in order to improve the original VFH method. The two most noteworthy versions are VFH+ [UB98] and VFH* [UB00]. These are discussed in the related work section of this thesis.

VFH works by mapping the environment around the robot into a grid. This grid is called a histogram grid, and the cells inside the grid contain values that represent the chance of an obstacle being in said cell. The robot uses multiple sensors to detect where obstacles are located and update the map accordingly. A small section of the grid that is around the vicinity of the robot is then transformed into a polar histogram. This polar histogram can be seen as a circle around the robot, divided into several equally sized sectors. The cells in the grid create a virtual force on the robot, which increase the value of the sector that covers the center of the cell. A threshold parameter is then compared with all of the sectors. A sector with a value that is higher than

the threshold is considered closed, while a sector with a lower value is considered open. A consecutive group of open sectors is called a valley, while a group of consecutive closed sectors is called a peak. The method chooses one of the valleys and steers the robot into the chosen valley. This means the robot can travel in an unknown environment, driving towards a certain location while avoiding obstacles.

We use a lot of data and references to the original VFH paper [BK91b], so we will be using the term ”original paper” to refer to the paper itself and use ”original robot” to refer to the robot used in the paper.

At the time VFH was created processors were a lot slower and memory was much more expensive. Technology has advanced a lot so better methods that have higher requirements can be executed on robots, but we will still discuss some hardware limitations as some cheaper robots use very limited hardware. For example, the turtle robot platform [Che] is made for the Arduino Uno, which only has 2KB of RAM and a clock speed of 16 MHz. The original robot had a computer running at 20 MHz.

VFH depends on a lot of parameters. The parameters control the strength of the forces the cells create, the size of the area the robot can operate on, the speed the robot can reach, the amount of cells there are in the histogram grid and the amount of sectors in the polar histogram, how to determine which directions are safe inside a valley, and a few more complex things discussed later in the thesis. It is hard to find proper values for these parameters, according to [UB98]. These parameters determine the reliability and effectiveness of the method.

To create our implementation of the method we will be using ROS [ROSa], and testing it in Gazebo [Gazc]. ROS is a framework that simplifies the programming of robots. Gazebo is a robot simulation program. Our implementation will be tested with the IRobot Create, using a differential drive and a ring of 24 sensors that are able to scan 360° around the robot.

1.1 Thesis contributions

This thesis makes two contributions:

1. Provide methods, algorithms and functions that help with finding values for the parameters of VFH. The performance of VFH relies significantly on its parameters. We base our methods, algorithms and functions on the values used in the original paper and our own evaluations of VFH.

2. Create a working implementation of VFH. The implementation will be as close as possible to the method described in [BK91b]. The paper does not explain several parts of the method, so we will use our own solutions for those parts.

1.2 Thesis overview

In Chapter 2, some similar methods and VFH related articles are discussed. Chapter 3 contains information about ROS and Gazebo. In Chapter 4, VFH is explained in detail. Chapter 5 describes our implementation of the robot. Chapter 6 contains an explanation and the results of the experiments we did to evaluate the robustness of the method. In Chapter 7, we sum up our results. In Chapter 8, we discuss some more work that can be done to evaluate VFH further and other issues that might need to be investigated.

Chapter 2

Related Work

This chapter discusses some other real time motion planning methods that are related to VFH.

2.1 Motion Planning in Robotics

Motion planning in robotics is the process of determining where the robot should go at any time in order to prevent hitting obstacles while attempting to reach a target. Motion planning is done by using some form of input and sending a motion command to the steering controller. In robotics the input of a motion planning method is often created using sensors, and the research that happens on the front of this input is called robot vision. Research into robot vision is split in two fronts: indoor robots and outdoor robots [DK02] [G13]. As VFH is an indoor motion planning method we will be focusing on that. The several approaches to indoor robot vision can be put into three groups:

1. Mapless Navigation
2. Map-based Navigation
3. Map-building-based Navigation

Mapless Navigation does not use a map when determining where to go. Only the actual results of the sensors is used to steer towards the target and around obstacles. Any robot that does not use a representation of a map belongs in this category, including robots that keep going forward, only turning if it would result in a collision. One way to implement mapless navigation is based on comparing the speed of objects on the left side of the robot with objects on the right side. To implement this two cameras need to be used, one pointed at the left wall and one towards the right wall. If the robot is closer to one of the walls the image will appear to move faster on that side, and the robot steers to the other wall [BSV98]. Another approach is Appearance-Based matching, where the robot compares objects in an image with a sequence of previously stored images or templates. The robot uses the comparison to predict where to move in order to find the next image [GJZ+97].

A slightly different way is tracking objects in the images, where the robot uses a moving object to determine where to go, like following the object [HSA89]. The advantage of these methods is that they save memory by not including a map of the working area. As these methods do not store any data about the environment the used data for path planning is up to date, and the methods using this approach do not have to deal with false positive obstacles created by an outdated map. The downside is that they do not have a map, which means that you can not give them the coordinates for a room. This means that this approach is good for robots where observation is more important than navigating to a specific part. This approach is used for Mars rovers [CRFS].

Map-based Navigation uses a map of the area in order to determine where the robot is currently located on the map. This is done using cameras and recognizing objects in the pictures. These recognized objects will be referred to as landmarks. The robot tracks the landmarks in the images in order to determine its own position on the map, while also keeping track of any obstacles in the image. This way the robot knows where to drive and can detect obstacles with the cameras [KK92]. An advantage of this approach is that the robot knows the shortest route to its target position once it knows its own position. A downside is that a 3D map has to be made in advance in order to detect the landmarks, and that their have to be enough landmarks the robot can track. The creation of the map can take a lot of time depending on the complexity of the map.

Map-building-based Navigation uses sensors to build a map of the environment instead of having a map made for it. Early research [Tho83] showed that creating a 3D map of the environment was slow, leading to most research using approaches that employ 2D maps. Research showed that it was easy to use a 2D grid, where every cell in the grid represents a small portion of the working area of the robot. One of the approaches that uses a 2D grid is the occupancy grid [ME85], where sonar sensors are used too fill the grid, and the value of the cells correlate to the chance of the cell being occupied with an obstacle. The map contains data from multiple sensors, and the path-planing only uses the map. This simplifies using multiple sensors. The downside of this approach is that it is hard to correct mistakes in the robots location caused by the odometry sensor limitations [BAM18]. Another downside is the fact that the grid can take up a decent amount of space in the robot's memory. Another approach is using topological representations [CKK]. They use cameras to detect landmarks in the environment and store information about them in order to recognize them when they encounter them again. However, this approach is a lot harder to implement, as it has to recognize specific objects. A general advantage of Map-building-based approaches is that they can be used without having to build a map yourself. This is helpful in events where such a map is very hard to create or even impossible. A disadvantage is that the robot will not always choose the best route if it is moving in a new environment. Another disadvantage with both approaches is that it is hard to differentiate between static and mobile obstacles, however, both approaches can be adjusted to negate this problem [BK91a] [CKK].

VFH is a Map-building-based Navigation approach, which uses a similar style grid as the occupancy grid [ME85]. The difference lies with filling the grid. When a sensor detects an obstacle the robot calculates which cells in the occupancy grid could contain the obstacle based on several factors. The histogram grid of the VFH method only increases the value of the cell in the center of the sensor's cone at the detected distance. A different grid based method is the dynamic window approach. Instead of using forces the robot uses the dynamics of the robot to determine at what speed the robot can travel and if it can fit through a

gap. This means that, given an accurate grid, the dynamic window approach is usually better in determining the right way to handle. The downside of the dynamic window approach is the speed of the method. The robots used to test the methods are different, but the one used for the dynamic window approach was better in terms of computations [BK91b] [FBT97]. The VFH method took only 27 milliseconds to compute, while the dynamic window approach took less than 250 milliseconds. These grid based can be used in environments where obstacles are plenty and the environment unknown [BK91b] [FBT97]. The idea of using a grid and 1-dimensional range sensors is also easy to understand and implement in contrast to using cameras and a 3D world, which require much more complicated techniques to use. Due to the accumulative error in odometry sensors robots using these sensors and maps can not be used for long times, unless precautions are taken to make sure that the robot can detect old data and throw it away when it does not accurately represent the environment. As the target position is usually some point in the area, and therefore a point in the grid, this point will shift due to the same odometry problem. Therefore it is important that for longer usage an additional sensor is used to synchronize the position and orientation of the robot with the area. As we will use Gazebo this will not be a problem, as it uses the model's location and orientation for the odometry. Another slight advantage is the fact that these grid based approaches can work with some very simple and cheap hardware, which means that the robot can be rather simple.

2.2 Virtual Force Field

The predecessor of VFH, virtual force field (VFF) [BK89] is a real time motion planning method that was made by Johann Borenstein and Yoram Koren, the creators of VFH. It was made to give a robot fast, continuous and smooth motion when obstacles are encountered. VFF uses a histogram grid. The cells in the grid represent a certain area, and the numerical value they hold represents the chance of an obstacle in that area. The cells can contain any non-negative integer. If a cell has a value of 0 no obstacle has been detected in that area. Higher values mean a higher chance of an obstacle being located in the area of the cell. The robot uses range sensors to detect the obstacles. If an obstacle is detected the value of the cell that corresponds with the area of detection is increased by 1. With sonar sensors the cell that is at the detected range and in the middle of the sensor's cone is increased. A window slides over the grid, with the center of the window corresponding with the robot's location in the grid. All cells inside the window apply a virtual repulsive force on the robot. The strength of these forces are proportional to the value of the cell and inversely proportional to the distance of the cell. All forces are added together to create a resulting force F_r. The target location applies an attractive force on the robot F_t. F_r and F_t are added together to create the final force and steering direction. The problem with this method was that when obstacles were separated less then 1.8 meters from each other the generated forces would create a F_r large enough to "push" the robot away from the target direction. Another problem that occurred was the difference in strength of the forces when the robot moved from cell to cell. When the robot moves a cell over the magnitude of certain vectors become larger, which causes drastic changes in the steering control. Another problem occurs when the robot is traveling in a narrow corridor. If the robot does not follow the center line, one of the walls will have a larger repulsive force which makes the robot turn away

from the wall, towards the other wall. When the robot crosses the line it will still be turned away from the wall and drives towards the other wall. This might make the robot travel a path that oscillates. These found limitations are discussed in [BK91b] [KB91]. VFH first transforms the histogram grid into a polar histogram with k amount of sectors. These sectors represent a direction, and their value the chance of an obstacle being located in that direction. Then based on some parameters a direction is chosen and the robot travels in that direction. This resulted in a smoother trajectory and smaller gaps that the robot can fit through. [BK91b] theorizes that the new minimum gap size is determined by the robots size, with their robot having a minimal gap size of 1.34 meters.

2.3 VFH+

VFH+ [UB98] is an adaptation of VFH. The paper mentions that VFH+ is better than VFH because it takes the dynamics of the robot into account and is more reliable. It also mentions that VFH has a lot of parameters that need to be set empirically and tuning these parameters can be very hard and time consuming. Even if the parameters are properly set VFH has a tendency to cut corners. They improve the method by adjusting several aspects of the method and adding 2 more data transformations. The robot's size is now taken into consideration when the grid is mapped onto the polar histogram. The polar histogram is then mapped onto a second polar histogram that is binary using two Threshold parameters. The second polar histogram is then masked based on the robot's speed in order to take into account the robot's turning radius. The selection of the direction is changed as well. VFH selects the valley that matches the most with the direction of the target, VFH+ uses a cost function to select the valley. These fixes result in a smoother trajectory. It also prevents a certain issue where the robot gets too close to obstacles when it detects many narrow gaps that have sectors that close and open constantly due to the Threshold parameter. The masked polar histogram prevents the robot from attempting to drive straight into an obstacle.

2.4 VFH*

VFH* [UB00] is an adaptation of VFH+. It improves VFH+ by trying to predict the best path using A* [HNR68], a global path finding algorithm. Instead of fixing shortcomings of the method itself it tries to adapt VFH to support situations that are problematic for local obstacle avoidance methods. The paper also mentions that VFH became quite popular and that this lead to several shortcomings of the method being found by researchers, but does not name a source and only shows one situation where the method might fail. VFH* checks if a certain steering direction is a good choice by using A*.

From the related works it is clear that problems with VFH have been identified. However, the papers do not mention exactly all found problems with VFH, and if these are all the problems observed with VFH. We can not compare the difference in performance between VFH and VFH*/VFH+, as different robots were used for evaluating. VFH mentions that the computation of the method takes a very low amount of time, and that

sampling the sensors took the most time. However, based on the description of the methods, we can assume that the improved versions do more, and require more time to compute. As some robots might not have the processing power to compute the improved versions it is still important to know exactly what problems there are with VFH.

Chapter 3

Gazebo & ROS

As mentioned in the introduction we will be using ROS and Gazebo to implement and evaluate VFH.

3.1 ROS

The Robot Operating System [ROSa] is a framework for writing robot software. It provides a collection of useful tools, libraries and conventions. ROS is a system that is used for the communication between different parts of the robot. The backbone of ROS is a message system with standardized messages. In Chapter 1, we mentioned that the software for robots consists of several methods and algorithms, which all control several parts of the robot. Similar robots often need the same parts, and a lot of these parts either require or create the same kind of data. If these parts are implemented in the code with ROS they can make use of the standardized messages. This means that robot programmers can more easily program the code for robots, as they only have to learn to work with the message system of ROS, instead of learning how to use a specific part of the robot. This means the programmer only has to implement his own functions and a ROS layer that communicates with the necessary parts of the robot. The way the ROS message system is setup is simple. Every part that implements ROS requires an ROS node. A node can use subscribers and publishers to communicate with the other parts of the robot. These work through the use of so called ”topics”. A part can send messages over a certain topic with a publisher. When a topic is published the ROS system is notified that the topic exists. The publisher can then send messages containing the created data. Parts can also use subscribers to receive the messages send over topics. If the subscriber receives a message it calls a function that was specified at the creation of the subscriber. This function then handles the received message.

3.2 Gazebo

Gazebo [Gazc] is a robot simulation environment. Gazebo used to be a part of ROS. Gazebo is now a standalone project, but many aspects of Gazebo still support ROS. It features many different robots, but also features a model editor so a new robot can be modeled and used in the simulations.

Simulation has a number of advantages for testing robots. In Gazebo you can specify what kind of area you want to use at the start of the simulation and change the scene while it is running. Gazebo provides models that can be placed in scenes for the robot to interact with. Models can also be made with the model editor. It also provides a tool to scale these models. This makes it easy to create scenes quickly. As the simulator can be restarted with a different scene you do not need to spend time setting up the new scenario if you have made the scene once before.

Gazebo features many types of sensors and actuators. Sensors and actuators are implemented into a robot with a description, and does not need an actual model that represents the sensor or actuator. The description describes which plugin the part uses and the settings of the plugin and the sensor. The plugin describes how the sensor or actuator should behave. A model uses a sensor type that comes with Gazebo, which sends the sensor's results to the plugin, where the result is used to create information for the robot, like a ROS message. The plugins are made with C++ and often implement ROS. An ultrasonic sensor also has settings for a minimum and maximum range, a sensor cone angle, a position and a direction. The position and direction are relative to the model the sensor is attached to. Actuators work in a similar fashion. A differential drive is implemented by specifying the joints, a specific type of attachment between two parts of the model, that connect the wheels to the rest of the robot.

Chapter 4

VFH

The vector field histogram method is a real-time motion planning method designed to move a robot towards a target position. While it travels in the direction of the target it attempts to steer around obstacles. It uses range sensors to detect these obstacles. The sensor data is put in a histogram grid. The VFH method is based on the VFF method, which is described in Chapter 2. VFF has several issues that are described in [BK91b], and VFH is an improvement of the VFF method. According to the paper too much data is transformed into the steering rate, resulting in a big reduction of data, which causes the problems. The paper discusses and evaluates VFH, where a second step is introduced to reduce the data to a smaller amount before calculating the steering rate. As the steps reduce the resolution of the data they are known as the ”first data reduction” and ”second data reduction”. The steering rate determines how fast the robot turns.

4.1 The Method

The two data representations used by the VFH method are both histograms. The cells of the histogram grid represent the chance of an obstacle being located in a certain position. The polar histogram represents the chance of an obstacle being located in a certain direction.

VFH uses a histogram grid to store information about the environment. A histogram is a way of representing a distribution of numerical data. The value of the cells inside the histogram grid represent the chance of an obstacle being located in the cells position. The cell can contain any non-negative integer value, where zero means that no obstacle has been detected and any higher number means that an obstacle has been detected. The robot detects obstacles through the use of range sensors. Only sonar sensors are discussed in [BK91b]. Sonar sensors are parsed by calculating which cell is located in the middle of the sensor’s cone at the measured

distance. The location of this cell c_{ij} can be found with

$$i = \left\lfloor \frac{x_r + \cos(a_s) \cdot r_d}{c_s} \right\rfloor$$
$$j = \left\lfloor \frac{y_r + \sin(a_s) \cdot r_d}{c_s} \right\rfloor$$

where

i	is the obstacle's column in the grid,
j	is the obstacle's row in the grid,
x_r, y_r	are the robot's current x- and y-coordinate,
a_s	is the sensor's current direction,
r_d	is the detected distance to the obstacle,
c_s	is the size of the cells in the grid.

Note that a_s depends on the direction of the sensor and the current heading of the robot, and that the units used for c_s, x_r, y_r and r_d need to be the same. r_d is the distance from the robot's center point, so the sensor's position relative to the robot has to be taken into account.

Also note that that the method assumes that the object is in the middle of the cone of sensor, which causes the grid to be an inaccurate representation of the environment.

For example: say that we have a sensor with a direction a_s of 210° that detects an obstacle at a distance r_d of 70 cm; our cell size c_s is 10 cm; and our robot is currently located at (x_r, y_r) = (10.05 m, 10.05 m). The corresponding cell of this location is (100, 100). The obstacle is then detected in cell c_{ij} where $i = \left\lfloor \frac{1005+\cos(210)\cdot 70}{10} \right\rfloor = 94$ and $j = \left\lfloor \frac{1005+\sin(210)\cdot 70}{10} \right\rfloor = 96$. Note that we used degree instead of radian. Figure 4.1 illustrates this example. A small portion of the grid is shown. The light blue cell is the robot's position. The transparent gray area in the bottom left of the figure is the detected obstacle. The empty cells are cells with a value of zero, the other cells show their value. The black outline is the sensors cone and detected range. The red line is the direction of the sensor. The red cell is where the obstacle is detected and its value is increased by one.

4.1.1 First data reduction

In the first data reduction the histogram grid is mapped onto the polar histogram. The polar histogram can be seen as a circle located around the robot that is split in several sectors. The sectors have a numerical value that corresponds with the chance of an obstacle being located in the direction of the sector. All sectors have the same size, so a sector will cover $360/n°$ with n sectors. In order to map the histogram grid to the polar histogram we use a window that slides over the grid alongside the robot. This window is a square, with a size of $w_s \times w_s$ cells. The window is centered on the robot, which means that the robot is always in the middle of the window. The size of the window is determined by the sensor range and the cell size, so w_s is usually much smaller than the grid size, which means that we only see a small square portion of the grid. Though a circular window would make more sense as the sensors also cover a circular area, the original paper claims that it

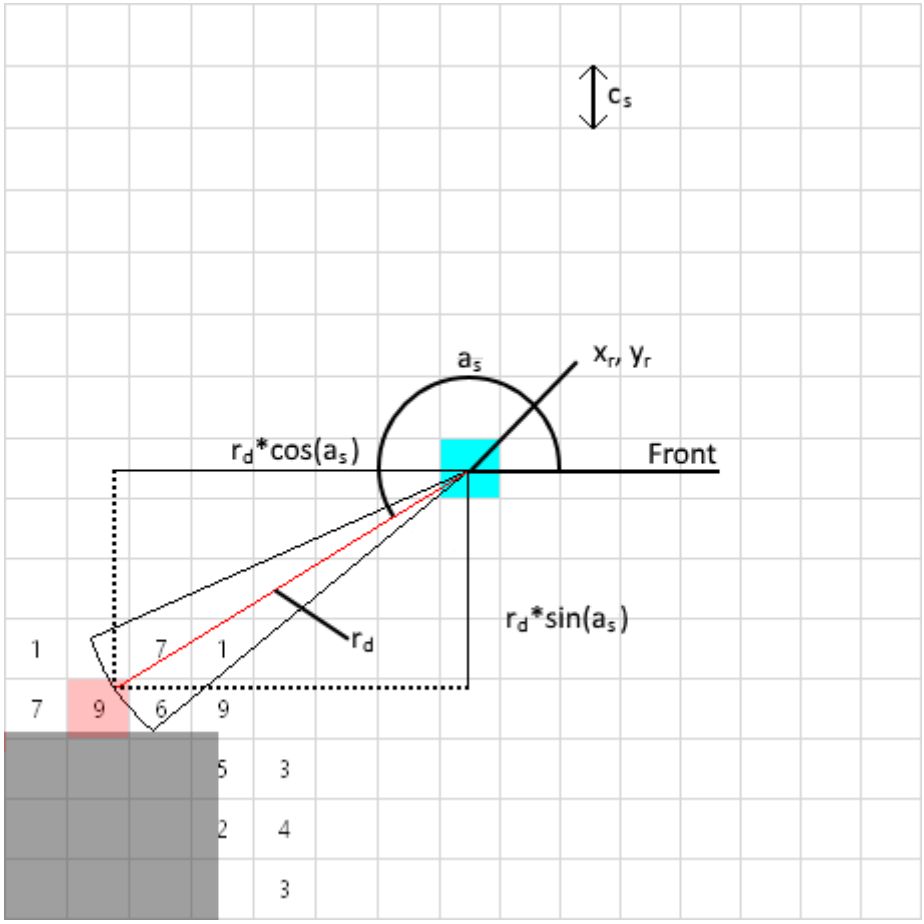


Figure 4.1: An example of filling the histogram grid.

is computationally more expensive to use. Then we map all cells inside the window to the polar histogram. All cells fall into a sector of the polar histogram and create a force that influences the value of the sector. The sector that corresponds with cell c_{ij} can be found with

$$k = \left\lfloor a_{ij} / \frac{360}{n} \right\rfloor$$

where a_{ij} is the angle of cell c_{ij} to the robot and can be calculated with

$$a_{ij} = \arctan \frac{y_i - y_r}{x_i - x_r}$$

where

x_i, y_i are the cell's x- and y-coordinates,

x_r, y_r are the robot's current x- and y-coordinates.

The amount the cell influences the sector is called the magnitude and can be calculated with

$$m_{ij} = (c_{ij})^2 (a - bd_{ij})$$

where

c_{ij} represents the value of the cell c_{ij},

a, b are both positive constants,

d_{ij} is the distance between the robot and c_{ij}.

There are some important things to note about the magnitude function. According to [BK91b] $a - bd_{max} = 0$ has to hold, where d_{max} is the maximum distance of a cell in the window. As our window is square and the robot is in the middle of the window the corner cells are the furthest away. We can also rewrite $a - bd_{max} = 0$ as $a = bd_{max}$ or $\frac{a}{d_{max}} = b$. This means that the corner cells of the window always generate a magnitude of 0. Another thing to note is that c_{ij} is squared, so higher values create stronger forces. This helps with the uncertainty of the histogram grid. The values in the grid only represent the chance of an obstacle location, with higher values meaning a higher chance of an obstacle. It also means that a value of zero results in a force with a magnitude of 0.

After all cells in the window have been considered, the polar histogram represents the obstacles around the robot. But because the histogram grid can be inaccurate we need to apply a smoothing function to the polar histogram. This smoothing function is defined by

$$h_k^{'} = \frac{h_{k-l} + 2h_{k-l+1} + ... + lh_k + ... + 2h_{k+l-1} + h_{k+l}}{2l + 1}$$

where

$h_k^{'}$ is sector k of the smoothed polar histogram h',

h_x is the value of sector x of the polar histogram h,

l is the smoothing factor.

This smoothing function attempts to clear sensor misreadings and other inaccuracies in the histogram grid.

4.1.2 Second data reduction

In the second data reduction we go from the smoothed polar histogram to the steering rate and desired speed. The steering rate determines how many degrees we turn per second. The desired speed is the speed the robot will accelerate or decelerate towards. We do this by using a simple threshold value and comparing it to the values in the sectors of the polar histogram. Any sector with a value that is higher than the threshold is considered closed, the others are considered open. A consecutive group of open sectors is called a valley, a consecutive group of closed sectors a peak. One valley is selected based on the valley's direction and the path monitor.

The direction of a valley depends on the width of the valley. A narrow valley has only one valley direction, while a wide valley has a range of valley directions. Whether a valley is wide or narrow is determined by the wide valley parameter s_{max}. The name of this parameter was chosen in [BK91b], and is used to determine

if a valley is wide or narrow. Any valley with more sectors than s_{max} sectors is considered wide. The valley direction θ_d of a narrow valley is found with

$$\theta_d = \frac{(k_n + k_f)}{2}$$

where

- θ_d is the resulting direction,
- k_n is the direction of the first sector of the valley,
- k_f is the direction of the last sector of the valley.

Every sector has two corresponding directions, defined by the left and right side of the sector. If we incrementally number the sectors in a counterclockwise fashion, then the left side of sector k has the same direction as the right side of sector $k+1$. If we do this for the narrow valley calculation then k_n is the rightmost sector and k_f the left most sector. Therefore, we use the right side direction of k_n and the left side direction of k_f. This results in the method attempting to drive towards the middle of the narrow valley.

For wide valleys we use a different method. By adding $s_{max}/2$ sectors to the first sector and subtracting $s_{max}/2$ from the last sector we get a range of valley directions. Any of the directions in the range can be chosen.

The robot computes the target direction, which is the direction of the robot to the target position. For every valley a direction that is closest to the target direction is chosen as the valley's direction. As narrow valleys only have one direction we do not have to compute anything. For wide valleys we have to compute the closest direction based on the target direction.

The path monitor keeps track of the method's current diversion direction. The reason for the path monitor is that in some situations the method would get trapped in a cycle. If the method decides that it is best to divert to the left to avoid an obstacle the path monitor is set to ”left diversion” mode. The same goes for diverting right and ”right diversion” mode. If the robot can go straight to the target the path monitor is set to ”no diversion” mode. If the method is in ”left diversion” mode it will only select directions that are to the left of the target direction. Again the same goes for the ”right diversion” mode and directions that are to the right of the target. When the method is in ”no diversion” mode it can select any direction. If the method is in left or right diversion mode, but there are no available valleys a global path planner is used to determine where to go, and the path monitor is reset to ”no diversion” mode.

In the end we have a list of valley directions that correspond with the diversion mode. Of these directions the one that is closest to the target direction is selected as the desired direction. The desired direction is the direction the robot will steer towards. The difference between the desired direction and the target direction is then used to compute the steering rate Ω. This is computed with $\textit{desired direction} - \textit{target direction} = \Omega$. If Ω is higher than 180°, 360° will be subtracted from it until it is lower than 180°. If Ω is lower than -180°, 360° will be added to it until it is higher than -180°. This ensure that Ω has a range of $(-180^\circ, 180^\circ)$.

In order to compute the desired speed we use two functions. The method uses two parameters, maximum speed adjustment V_{max} and minimum speed V_{min}. V_{min} is the minimum speed of the method, the maximum speed adjustment V_{max} is the maximum speed on top of the minimum speed. Because of this the method's maximum speed is $V_{max} + V_{min}$. The first function is

$$V' = V_{max}(1 - h_c''/h_m)$$

where

h_m is an empirically chosen parameter which is called the sector speed adjustment,
h_c'' is equal to $\min(h_c', h_m)$.

h_c' is the value of the polar histogram sector that corresponds the most with the robot's direction. With this function the speed is lowered depending on the current direction the robot is travelling in. If the robot is heading towards a big obstacle or obstacle that is close the value of the sector is most likely going to be high so the robot will slow down.

The second function lowers the speed according to the current steering rate and the maximum steering rate of the robot and gives us our final speed.

$$V = V'(1 - \frac{\Omega}{\Omega_{max}}) + V_{min}$$

where

Ω is the current steering rate in $^\circ$/s,
Ω_{max} is the maximum steering rate in $^\circ$/s.

A check is performed at the end of the speed calculation to make sure that the computed speed V is above the minimum speed, if not V is set to V_{min}. V is then used as our desired speed.

4.2 Parameters

Now that we have introduced the method we need to discuss the parameters of the method. The parameters can be found in Table 4.1. The table contains a short description of the parameter, and shows the relationship between the parameters. The rest of the section gives a detailed description of every parameter and why it is related to other parameters.

The size of the grid is determined by the Area size and the Cell size. The Area size is a rectangle, and the Cell size is a square. The amount of cells in the grid is the Area size divided by the Cell size. The grid requires the highest amount of memory among all parts of the VFH method. If we use a large amount of grid cells it will have a huge impact on the robot's memory, so the robot's available memory has to be taken into account when choosing the parameter values.

Parameter	Description	Relations
Area size	Size of the area.	Cell size
Cell size	Size of a cell.	Area size
Sensor range	Range of the sensors.	
Sensor positions	Positions in reference to the center of the robot.	
Window size w_s	Size of the window.	Sensor range
Constant a	Amplifies the magnitude of forces.	Constant b
Constant b	Amplifies the magnitude of forces.	Constant a
Polar sectors n	The amount of sectors in the polar histogram.	Smoothing factor
Smoothing factor l	The smoothing factor used for the smoothing function.	Polar sectors
Threshold	The threshold valley used to determine if sectors are closed or open.	
Wide valley s_{max}	Determines if a valley is wide or narrow, and determines the range of the directions in a wide valley.	
Constant h_m	Determines the slowdown when the robot is heading into an obstacle.	
Constant Ω_{max}	The maximum steering rate of the robot.	
Minimum speed V_{min}	The minimum speed the robot should go.	Update time
Maximum speed adjustment V_{max}	How much faster the robot is allowed to go than the minimum speed V_{min}.	Update time
Update time	The time it takes the robot to parse the sensors and execute the VFH method.	Robot Speed

Table 4.1: All parameters used by the method.

The range sensors have a maximum detection range. The sensor's position has to be taken into account when calculating the detected position of an obstacle. Longer sensor ranges will of course be able to detect obstacles earlier, but due to the way the sensors are parsed, as explained in Section 4.1 just before Subsection 4.1.1 starts, sonar sensors will be less accurate at longer ranges.

[BK91b] discusses that all cells inside the window should be covered by the sensors while it should still be as big as possible. When the range sensors are mounted on a circular ring the area they cover is circular, so we can compute the diameter of the sensors coverage c_s with 2·Sensor range + ring diameter. The sides of the largest square that fits in the sensor area would then be $c_s/\sqrt{2}$ [Squ]. If we then divide the size of the square by the Cell size and round the number down we find the size of the window sides in the amount of cells. This method ensures that the window fits inside sensors covered area.

Constants a and b only depend on each other. We already know that the value of a is equal to bd_{max}. We can use $b = 1$, which means that $a = 1\dot{d}_{max} = d_{max}$. Choosing different values for the constants does not really change the way the method works. This can be proven with a simple equation. If we take c as the value of c_{ij}, a and b as our values for the constants, and d as d_{ij}, we get a final magnitude of $c(a - bd)$. If we then multiply our constants with k we get $c(ka - kbd) = ck(a - bd) = kc(a - bd)$, which means that the magnitude multiplies with k as well. This happens to every generated force, which results in the values of the polar histogram to be multiplied with k as well. In the end the only things that interact with the values of the polar histogram are the parameters Threshold and constant h_m. If we assume that both parameters have a very specific good

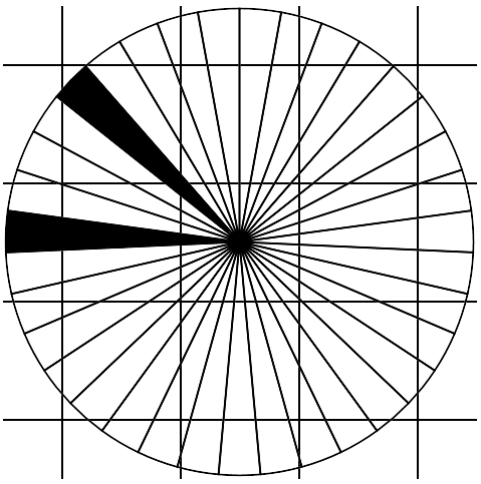


Figure 4.2: Sectors are skipped as the polar histogram has a much higher resolution than the histogram grid.

setting, then multiplying the polar histogram's values means that the parameters also need to be multiplied by k. Therefore, we can say that every value for the constants are good, as long as $a = bd_{max}$ holds.

The Polar sectors and Smoothing factor parameter have to be carefully chosen. The amount of sectors in our polar histogram determine in what directions we can steer and how accurate the polar sector is. When only a few polar sectors are used gaps between obstacles will be hidden as the polar histogram does not have the resolution to see the gap. The same happens if the smoothing factor is too high, smoothing the sectors out too much and raising the values of the sectors containing the gap above the Threshold. A very high amount of sectors has barely any use however, as the accuracy of the polar histogram is affected by the resolution of the grid. If this happens sectors are "skipped" when computing the force of the cells, as is illustrated by Figure 4.2. In the image the Cell size is relatively large. As the middle of a cell is used to calculate which sector the cell belongs to 3 sectors are skipped by adjacent cells. This can be fixed by increasing the smoothing factor, but that also increases the necessary computations for the smoothed polar histogram, slowing the method down and potentially hiding the gaps again. Depending on the amount of sensors this problem might occur in a different way. Because there are only a few sensors, the cells where the obstacles are detected are likely not adjacent cells. This causes the same issue. If this happens the smoothing function should cover the gap that 2 adjacent sensors might make.

The Threshold parameter, Wide valley, and Constant h_m need very specific settings. The Threshold parameter and Constant h_m depend purely on the values of the polar histogram. Wide valley has 2 functions: decide if a valley is wide or narrow; or aid in steering away from an obstacle. In both functions the threshold plays a major role. It is not easy to predict what kind of values the polar histogram will have, so we are not able to say anything about the relations between these parameters and the others.

Constant Ω_{max} is the maximum steering rate of the robot. It is defined by the robot, so we can test what the maximum steering rate of the robot is.

For the Minimum speed and Maximum speed adjustment we need to take the Update time into account. If the method takes a long time to execute, the robot needs to travel at a low speed. If the robot goes too fast the

method can not keep up and the robot might crash into an obstacle. It is not a problem if we go too slow for the method, however, as that just means that the robot can sample the sensors more, increasing the values of the polar histogram and histogram grid. This increases the accuracy of the polar histogram and histogram grid, which increases the reliability of the method. The reason for the 2 speed parameters is that some robots need to move in order to turn. It should be noted that [BK91b] mentions that there are a lot of factors that determine the maximum speed of the robot, and that the relation between the factors is rather complicated.

Chapter 5

Implementation

The evaluation of the robustness of VFH needs three implementation parts: the implementation of VFH itself; the setup of the testing robot; and the ROS layer connecting the robot parts and VFH. First, we explain what robot setup we use and the ROS layer, then we will explain the VFH implementation. As ROS works with C++ we wrote our implementation in C++.

We base our parameter values on the parameter values that were used by the original robot. Our robot, the IRobot Create [IRo], is smaller than the robot used in the original paper [BK91b], so we need to make sure that we account for that where necessary. Another difference is the fact that [BK91b] used real sonar sensors and we must make our own "sonar" sensors, as Gazebo does not have properly working sonar sensors. Our own "sonar" sensor is using a ray sensor with 10 rays that are aligned horizontally, which simulate the cone of the sensor. The ray that reports the shortest distance is taken as our "sonar" reading. This way we get similar results as with a real sonar sensor, but a ray sensor can detect obstacles more easily as it is not affected by the angle of the obstacle, where real sonar sensors will not receive the "ping" back and can not measure the distance if the angle is too low to reflect the "ping" towards the robot. We will refer to these "sonar" sensors as sonar sensors. The robot uses twenty-four of these sonar sensors. The robot's drive controller also has an odometry sensor which is used to track the location of the robot.

We will use a size of 10×10 cm for our cells and 72 sectors for the polar histogram based on the robot of [BK91b]. We will also use 24 sonar sensors as the original robot did as well.

Both sensor types are sampled at 10 Hz, so they generate a ROS message every 100 ms. The sensors are independent from each other, which means that we do not know in what order the sensors are sampled by ROS, and could lead to slight issues because of outdated information. This might be similar to the original robot, as it took 160 ms to parse all the sensors and we do not know if the samples of the sensors were from the same time or if it sampled and parsed one by one. As our update time is lower we can make our robot slightly faster. The original robot has a maximum speed of 0.78 m/s with an update time of 160 ms. If we assume the simple inverse relationship $a/speed = updatetime$ between the two we get a maximum speed of 1.248 m/s for our update time of 100 ms. We go for 1.0 m/s however, as [BK91b] mentions that the relationship

between various factors affecting the speed is rather complicated. With this lowered speed we are most likely below the actual maximum speed we could go, but we will have to test the robot to make sure that it works.

When ROS receives a sonar sensor message we parse the information of the message by calculating in what cell the obstacle is located, as is described in Chapter 4. When we receive an odometry message we take the following actions: first we update our location in the grid, then we calculate our new desired direction and speed, the desired direction gets transformed into a steering rate, and finally our drive controller is updated with the steering rate and speed.

Our implementation of the method is as close as possible to the original version, but some details are not explained in the original paper so there could be some big differences between our implementation of the method and the original method. Our implementation of the method is completely explained in Section 5.1. All differences between the original method and our implementation can also be found there.

Table 5.1 contains all the values of the parameters that were used for the original robot and what we use for our robot. Some of the original values are unknown, but we can work around this. Instead of a list of sensor positions we give the Robot's diameter, as both robots have a ring of sensors so the distance between the sensor and the center is the same for every sensor. We do not consider the robot memory and will just assume that we have enough.

For our cell size and polar sectors we simply copy the original values. For the area size we use 20×20 m, which is large enough for us to do our experiments in and gives us an easy size that is consistent, so we do not have to adjust the value every experiment. Our robot has a very high top speed and steering rate, so the values for those parameters are limited by the method. The speed has already been discussed, and the difference between the current heading and desired direction can at most be $180\ ^\circ/s$, so we use that as our Constant Ω_{max}. For the constants a and b we use the values d_{max} and 1, as was discussed in Chapter 4. For the smoothing factor we need to think about how it works and what it affects. The smoothing factor l smooths $2l - 1$ sectors over a single sector. The smoothing factor is used to make sure that any inconsistencies in the grid or misreadings are hidden. An easy way to visualize this is by imagining the robot to be stuck in a single spot where it does not turn. If the robot is completely surrounded by obstacles the same cells will be adjusted by the sensors, resulting in only a few sectors of the polar histogram being influenced. For our robot there are 72 sectors and 24 sensors, so $\frac{1}{3}$ of the sectors will have a value. Therefore, the smoothing factor has to be at least 2 in order to cover every sector. That said if we use 2 as our smoothing factor the sectors in between the "sensor sectors" will have lower values than the sensor sectors, but it is much more likely that the obstacle also covers those sectors. In order to evenly spread the sector values you need to use the amount of sectors a single sensor covers. In our case that is 3. Note that the smoothing function is smearing peaks over multiple sectors. Because of this, valleys become smaller and the resulting direction will steer the robot further away from the obstacles. This is helpful with inaccurate sensors that can not detect obstacles under certain angles, but setting it too high will result in small gaps being hidden. This might explain why the original value is 5, but our robot could be fine with 2. However, to be safe we will use 3, risking the chance that very small gaps might not be

Parameter name	Original value	Our value
Area size	Unknown	20×20 m
Cell size	10×10 cm	10×10 cm
Sensor range	2 m	Varies
Robot diameter	80 cm	30 cm
Window size w_s	33 cells	Varies
Constant a	Unknown	d_{max}
Constant b	Unknown	1
Polar sectors n	72	72
Smoothing factor l	5	3
Threshold	Unknown	Varies
Wide valley s_{max}	18 sectors	Varies
Constant h_m	Unknown	Varies
Constant Ω_{max}	$120\ ^\circ/s$	$180\ ^\circ/s$
Maximum speed adjustment V_{max}	0.74 m/s	0.96 m/s
Minimum speed V_{min}	0.04 m/s	0.04 m/s
Update time	160 ms	100 ms

Table 5.1: The values of the parameters used for the original robot and our robot.

picked up by the method. The other parameters will be changed during the experiments, these parameters are marked with ”Varies” in the table.

5.1 The method

We do not use any external libraries aside from the ones provided by ROS. This means that we made the implementation of VFH ourselves. Doing this means that we have complete knowledge and control of the code, which helped in understanding the method. This also means that we can easily change the code, allowing us to quickly implement functions and variables necessary for debugging and testing. Though there are open source versions available, most of them are not finished. There is also one for gazebo, but the ROS version of the method was outdated and does not work anymore.

Before the method works we have to set the parameters. For the sonar sensors we created a single function. For the rest of the calculation a few functions have to be called in order to find the speed and turning rate. As was explained at the start of the chapter the sonar sensors use the sensor function, while the odometry event updates the robot’s location and performs the method.

The second data reduction is slightly changed in the way how the valleys are chosen, but the result is the same. In the original method all valleys are considered and the best direction is chosen. With our method we try to find the best valley by considering at most two valleys. This is done by first determining the target direction, which is the direction of the target. Then we check if all sectors are open and if so give the target direction as our desired direction. If all sectors are closed we instead report that the robot is blocked, effectively stopping the method. If there are both valleys and peaks in the polar histogram we map the target direction to a sector, which we will call the target sector. This target sector can be seen as a middle point that determines what sectors are to the left of the robot and which sectors are to the right. If the target sector is part of a valley the best direction of the valley is calculated. If the target direction is equal to the best direction or corresponds

with the method's current diversion mode we use that best direction. If the best direction does not correspond with the diversion mode or the target sector is not part of a valley the next valley is found. Depending on the diversion mode the method will search either on the left side or the right side of the target sector. If no valley could be found or the valley that was found does not have a potential direction inside the current diversion mode the method reports that the robot is blocked, otherwise the method gives the best direction of the valley as the desired direction. In the end the diversion mode is set by finding the difference between the target direction and the desired direction. As mentioned before if the robot is stuck because it could not find a direction within the current diversion mode it changes the diversion mode to the other side. This way we can quickly compute the desired direction instead of considering every sector.

Another possible difference between the original and our implementation has to do with the grid and the window. The original paper does not mention what the method should do when the window slides over the boundaries of the grid. In our implementation all "cells" outside the window have a value of 10, regardless of an obstacle being located there or not. This should create enough magnitudes to steer the robot away.

The biggest difference is probably found in the path monitor. The path monitor makes sure that the robot keeps going in the same direction around an obstacle, but when the robot can not travel in the same direction anymore the original method uses a "Global Path Planner (GPP)" to find a new path, while we simply switch diversion mode. The reason for this is that the GPP is only mentioned and not explained. Therefore, we do not know what kind of result the GPP should return, what to do when we can not reach a point, if the GPP considers the robot's dimension, and most likely a few other things. Another problem is that we do not know what to do when the target position is located somewhere unreachable. Our robot keeps trying to reach the target position until it either reaches it, by getting within 20 cm of the target, or is told to stop.

All other parts of the method are implemented in the same way as the original paper describes.

Chapter 6

Experiments

Before we can begin the experiments we need to set our parameters values. We switch between three different sensor ranges for our experiment, in order to evaluate the effect of the parameters. We already know that some of the other parameters depend on the sensor range and we will have to change those too for our experiments. We will be using a sensor range of 1.35, 2.25, and 3.5 meters. Our 2.25 meter sensor range is based on the original robot. The original robot had sensors with a range of 2 meter and a diameter of 0.8 meters. This means that the diameter of the area the sensors cover is $2 \cdot 2 + 0.8 = 4.8$ meters. In order to match coverage with our robot our sensors need to have a sensor range of $\frac{4.8-0.3}{2} = 2.25$. The other ranges are chosen as they are higher and lower than our reference sensor range, and we can then evaluate the effect of the sensor range. With the window size computation function given in Chapter 4 we get the resulting window sizes of 21, 33, and 51 cells, respectively. This leaves us with three parameters that do not have values: threshold, wide valley s_{max}, and constant h_m. For these parameters we will do a calibration experiment that we will call the polar calibration. In order to make testing easier, in all our experiments the robot will start at position (5, 10) and have a target position of (15,10), unless stated otherwise.

Some of the experiments will have illustrations of the scenario that is used. These illustrations are rough sketches to aid in the understanding of how we have the scenario set up. The black circle represents the robot, the red square is the target position for the robot, and the obstacles are represented with black outlines.

6.1 Polar calibration

The polar calibration will help us find values for the last three parameters. The robot drives parallel to a flat surface towards its target position. An image of this scenario can be seen in Figure 6.1. For our experiment we use three different distances to the wall, which we refer to as the calibration distances. They are $\frac{1}{3}$, $\frac{1}{2}$, and $\frac{2}{3}$ of the sensor range. These calibration distances are empirically chosen, but are based on a few calculations. For the calibration to work the obstacle should be detected inside the window. If we assume that the range sensors measure exactly where the obstacle is, instead of the inaccuracy of the sonar sensor, the minimum

distance before an obstacle falls completely out of the window is half the window size. As the size of the window is based on the sensor range, as said in Chapter 4, we can also say that the minimum distance before the object disappears is equal to the *sensor range*$/\sqrt{2}$. We then use the mathematical trick $1/\sqrt{x} = \frac{\sqrt{x}}{x}$, so that *sensor range*$/\sqrt{2}$ becomes *sensor range* $\cdot \frac{\sqrt{2}}{2}$. As our calibration distances are also based on the sensor range (they have to be multiplied with sensor range as well) we can now say that $\frac{\sqrt{2}}{2}$ is our maximum calibration size. As $\frac{2}{3}$ is slightly smaller than $\sqrt{2}$ we choose this as our longest calibration distance. The theoretical minimum is a distance of 0 cm, but that would mean the robot is touching the obstacle, which is what we are trying to avoid. For our minimum calibration distance test we choose $\frac{1}{3}$ of the sensor range. This means that there is a difference of $\frac{1}{3}$ between the minimum and maximum calibration distance. As we do not know what calibration distance is better we also use $\frac{1}{2}$, which is in the middle of the two numbers.

At the end of our evaluation we want to be able to say that one of our three calibration distances is the best in terms of reliability and performance. In order to give values to our parameters we save the final state of the histogram grid and the final location of the robot once the target has been reached. This way we can easily recreate the polar histogram of the robot at the final position, where we can find our values for the threshold and wide valley parameters. The values found using this method should ensure that if the robot is driving alongside a flat surface it should never suddenly crash into the obstacle. If we recreate our polar histogram there should be one peak and one valley. For our wide valley parameter we will double the difference between the robot's current sector and the third sector of the peak that is closest to the robot's current sector. We choose this sector because of our smoothing factor value. If a cell contains a value higher than zero we know that an obstacle is detected in that cell, and this cell is then mapped onto the polar histogram. The polar histogram is smoothed out , which means that the cell is smoothed out over the polar histogram with the smoothing function. With our smoothing factor of 3 the sector of the cell is "smeared" out over two sectors in both directions. When an obstacle is detected an area in the grid will have values. At the edges of the area we do not know if the obstacle continues, and we assume it does not, but the smoothing function will smooth the sectors out. This means that we need to ignore two sectors in a peak in order to find the correct representation of an obstacle. For example, if the robot's current sector is 1 and the closest sector of the peak is 9, which means that sector 11 is the third sector, we get a wide valley parameter of 20, as $(11 - 1) \cdot 2 = 20$. For the threshold parameter we take a value that is in between the values of the second sector and third sector of the peak. This leaves us with constant h_m. The only time this parameter should influence the speed of the robot is if the robot is heading straight towards an obstacle. If it does this it will probably already slow down because it is turning, so the values will keep increasing as long as the robot needs to turn. Therefore, we decide to create a polar histogram of the final grid in the middle of the path of the robot and use the highest sector value we could find. However, a few quick tests showed that this made the robot slow down very erratically, so we instead use the highest found value for the sensor range and double its value. The results can be found in Table 6.1. Note that the highest value was rounded to the nearest 100. This was done as the highest value changed slightly over multiple test or if we take a slightly different position than the middle of the path. Therefore, it did not really make sense to give it an exact value. For the rest of the experiments we refer to the calibrations with "(sensor range):(calibration distance)", for example, calibration 2.20:$\frac{2}{3}$, which means the calibration of the 2.20 m sensor range with the wall at a distance $\frac{2}{3}$ of the sensor's range.

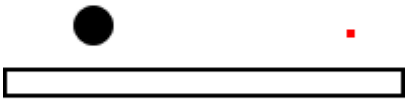


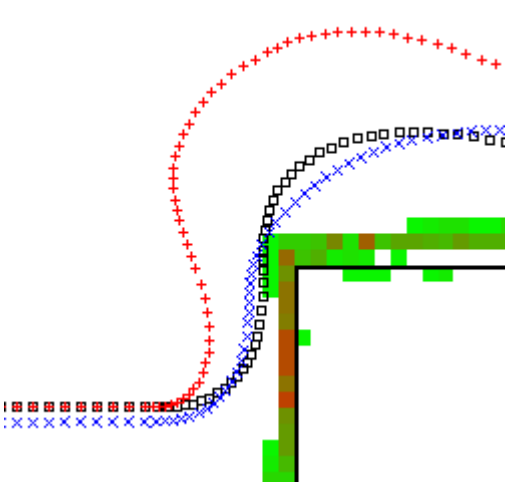
Figure 6.1: Polar calibration setup.

Sensor range	Distance	Wide valley	Threshold	Highest value
1.35 m	0.45 m	8	6	500
	0.675 m	12	3	400
	0.90 m	20	1	100
2.20 m	0.73 m	8	16	1300
	1.10 m	12	5	500
	1.47 m	18	5	300
3.50 m	1.17 m	8	30	1600
	1.75 m	12	16	1000
	2.33 m	18	10	600

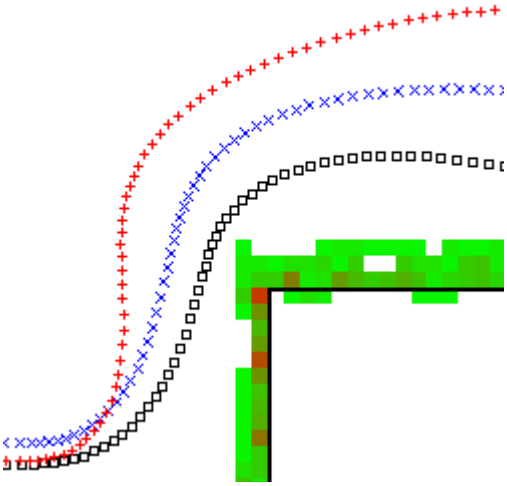
Table 6.1: Results of the polar calibration.

6.2 Calibration test

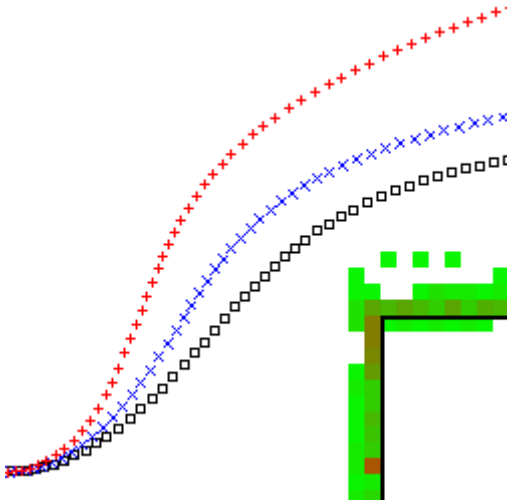
In order to test our calibrations we test if the method will steer around a block when trying to reach its target. This should show if our calibration and other parameter values are working. For this experiment, we use a scenario where a 2×2 meter block is placed right between the robot and its target. This means that the robot will have to go around the obstacle in order to reach the target position. If the parameters are correct the robot will drive around the block, if the parameters are incorrect the robot will likely get too close to the block. In case this happens the robot will have to drive away from the block instead of simply going around it. Therefore, we will keep track of the path of the robot in this experiment. The approach portion of the path is shown in Figure 6.2. The path is represented by shapes where the odometry event occurred. The black squares represent the $1/3$ calibrations, the blue X's $\frac{1}{2}$, and the red pluses $\frac{2}{3}$. The colored blocks are the certainty values of the histogram grid. Note that the shown grid is the final state grid of one of the calibrations. The black bar is a representation of the actual block, so we can see where the actual block is located. The calibrations of both the 2.20 m and 3.50 m sensor ranges work rather well, with calibration 2.20:$\frac{1}{3}$ getting rather close to the obstacle but still avoiding it. The 1.35 m calibrations got too close to the obstacle and the robot needs to drive away from the obstacle. This is due to the fact that the obstacle appears too suddenly for the method to react in time. Calibration 2.20:$\frac{2}{3}$ drives away from the obstacle as well, but only very slightly. This was not caused by the robot getting too close but a slight oscillation in the desired direction.



(a) The paths of the 1.35 m calibrations.



(b) The paths of the 2.20 m calibrations.



(c) The paths of the 3.50 m calibrations.

Figure 6.2: The results of the calibration test experiment.

Note that a single certainty value block is 10×10 cm and that the robot has a radius of 15 cm. This means that calibration $1.35{:}\frac{1}{3}$ was only 5 to 10 *cm* away from crashing into the block. This is way too close for the method. Therefore, we need to find a solution for the 1.35 m calibrations. As the problem has to do with the fact that obstacles appear too quickly for the method we can do several things to slow the robot down: lower the Maximum speed adjustment; lower constant h_m; or lower constant Ω_{max}. As the two constants have ways to set them we lower the Maximum speed adjustment first. As the calibrations of the other sensor ranges seem to work we make another guess at a relation, this time between the sensor range and the maximum speed of the robot. We can easily determine that if the sensor range becomes shorter, the robot has less time to react to obstacles. If we slow the robot down we increase the time the robot has to detect the obstacle. If we would half the sensor range, the obstacle would be detected for half the time at the same speed. In order to get the same amount of time we would have to half the robot's speed. This suggests a linear relationship between the two. Our sensor ranges need to include the robot's radius, so our sensor ranges are $2.20 + 0.15 = 2.35$ and $1.35 + 0.15 = 1.50$. The radius of the robot has to be included because the window calculations are based on the center of the window and the cells where obstacles are located. As we drive at a speed of 1.0 m/s with a

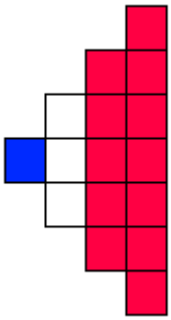


Figure 6.3: Amount of cells in 90 degrees.

sensor range of 2.20 m, our new top speed should be $\frac{1.0}{2.35} \cdot 1.5 \approx 0.64$ m/s. As our minimum speed is 0.04 m/s our new maximum speed adjustment is 0.60. Another problem we have is that the density of the grid is not high enough for the distance between the robot and obstacles. The cells are too close to the robot and do not spread well over the polar histogram. If the distance of an obstacle to the robot is around 30 cm it will only be four cells away from the robot. Considering this and the accuracy of the sensors, then the cells with certainty values will be about two or three cells away from the robot's center. If we assume that this obstacle is a wall it will be an entire line of these close cells. If we look at the amount of cells in a 90 ° angle close to the robot we find that the amount of cells is lower than the amount of sectors. Figure 6.3 illustrates all cells in those 90 °. The red cells are where the obstacle was detected and the blue cell is where the robot is located. There are 12 red cells that have to be mapped to 18 polar sectors. Though the smoothing function should help with this we can not guarantee that the obstacle was detected in every cell. Therefore, we need to increase the density of the grid, which we can do by reducing the cell size. We half the cell size, to 5×5 centimeter, multiplying the density of the grid four times. If the explanation was not entirely clear try to think of this extreme example. We have a distance of 1 m to an obstacle, but our cell size 1×1 meter. This means that there will only be three cells at most where the obstacle was detected. The smoothing factor can not cover this up without severely affecting the correct direction.

Since we have these new settings we need to redo the polar experiment for the 1.35 m calibrations. Because of the new settings the window size has increased from 21 to 42. The results of this experiment can be found in Table 6.2. The paths of the calibration test experiment can be seen in Figure 6.4. We can see that the robot stays further away from the obstacle, but $1.35{:}\frac{2}{3}$ still gets too close and drives away. Because the robot drives away from the obstacle it will take longer before the robot reaches the target position. This is not a severe problem however, as the robot is just too careful, trying to stay away as far as possible from the obstacle, instead of crashing into the obstacle.

Sensor range	Distance	Wide valley	Threshold	Highest value
1.35 m	0.45 m	8	16	800
	0.675 m	12	12	700
	0.90 m	18	6	300

Table 6.2: The paths of the fixed 1.35 m calibrations.

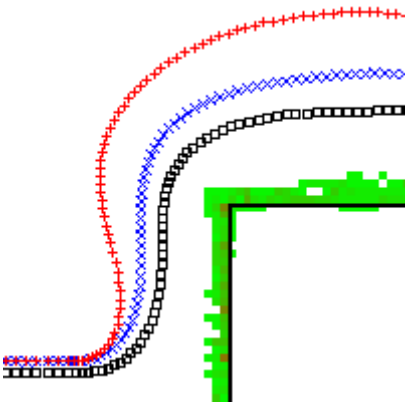


Figure 6.4: Paths of the 1.35 m calibration

6.3 Minimum gap size

Now onto the actual experiments. First, we do an experiment to test the minimal gap size the robot can go through. We can do this by simply creating a gap with two obstacles. For this we use two 10×0.5 m blocks. The blocks are placed as shown in Figure 6.5, creating a doorway-like gap. For the experiment we keep decreasing the distance between the obstacles until the robot can not find a way through.

However, while doing the test a problem occurs and the robot only fits through very large gaps. When the robot approaches the gap the robot finds itself inside a narrow valley. This narrow valley creates a direction that is either slightly to the left or right of the target direction. This makes the robot enter one of the diversion modes. However, at some point the narrow valley might switch from direction, meaning that the valley does not correspond with the path monitor anymore. This causes the method to go for the next potential valley which results in the robot steering away from the gap.

We need to use a different setup in order to find the gap sizes. To not run into the same problem we create a scenario where the target direction will always be on the same side. Therefore, we use the setup shown in Figure 6.6. For this we need a 10×0.5 m and a 8×0.5 m block. The robot's starting position is (4, 14) and the target position is (15, 14), except for the 3.50 calibrations, where we use (4, 15) as our starting position. This is necessary because otherwise the robot selects the wrong valley and goes around the hallway.

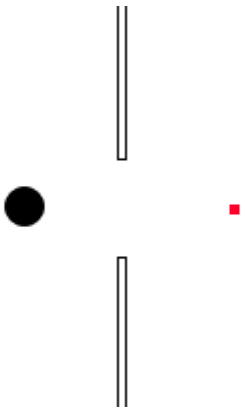


Figure 6.5: Setup of our gap experiment.

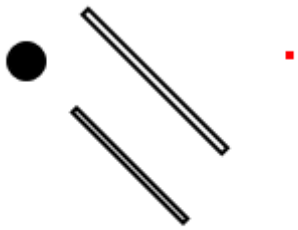


Figure 6.6: Setup for the hallway gap experiment.

The results can be found in Table 6.3. From the results we gather that the minimum gap size seems to only depend on the sensor range and not necessarily the settings. The found gap sizes can also be quite easily explained with some math. The cone of the sonar sensors covers a certain area at the furthest distance it can detect. We can calculate the diameter of this area with the law of cosines, which is $x^2 + y^2 - 2xy\cos(Z) = z^2$, where x and y are both the sensor's range, Z the angle of the cone of the sensor and z is the diameter of the cone. As our cone angle is 15 $^\circ$ we have a diameter of approximately 0.35, 0.57, and 0.91 m, which can be seen as the minimum gap. But for this to work the robot has to be exactly in the middle of a path and the smoothing function might even then hide the gap. It is much more likely that we can travel in a gap if the gap size is about two sensors areas wide, as then one sensor should always be free. If we look at the the area that is covered by 2 sensors, 0.69; 1.13; and 1.81 m we see that they are very close to our results. The difference is most likely related to misreadings and the smoothing function. That said, we already mentioned in Chapter 5 that our sonar sensors are capable of detecting things normal sonar sensors will not detect, including a wall in the distance that is parallel to the robot's path. This might mean that a robot with normal sonar sensors will not be able to detect the obstacle in cells further away, increasing the valley's width, so the robot might fit through tighter gaps as long as the the robot stays on the straight path.

Sensor range	Calibration	Min. gap size
1.35 m	0.45 m	0.80 m
	0.675 m	0.80 m
	0.90 m	0.80 m
2.20 m	0.73 m	1.20 m
	1.10 m	1.20 m
	1.47 m	1.40 m
3.50 m	1.17 m	1.90 m
	1.75 m	1.90 m
	2.33 m	1.90 m

Table 6.3: The results of the hallway gap experiment.

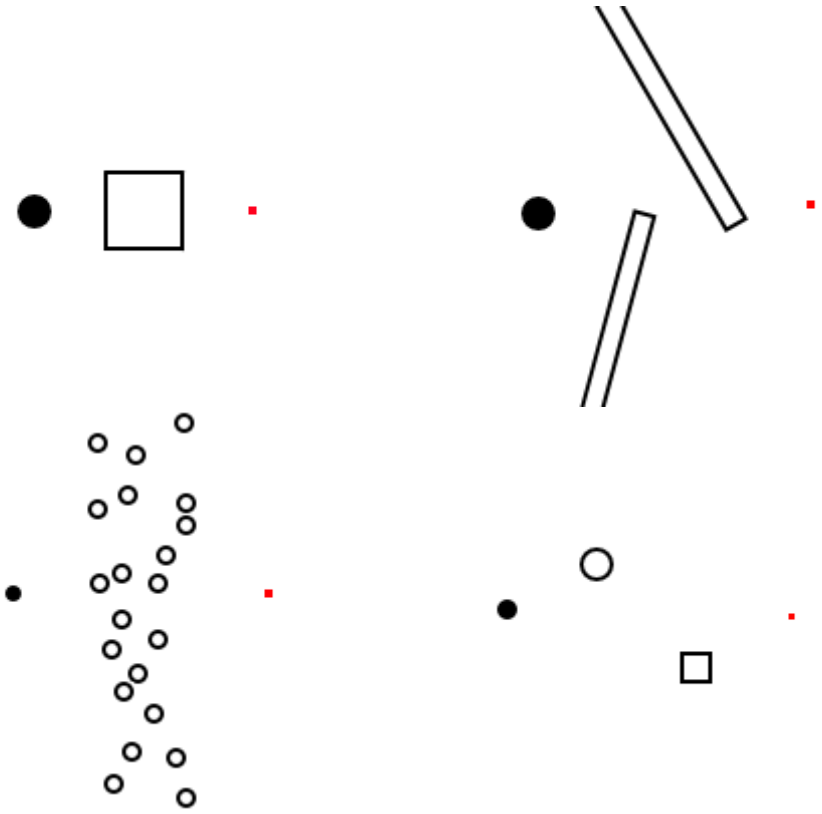


Figure 6.7: All scenarios used for the time experiment.

6.4 Time test

For our last experiment we measure the time it takes for the robot to get to the target position in several scenarios. We measure the time in seconds. We let the robot go through four scenarios that are roughly illustrated in Figure 6.7. These scenarios were chosen as they give us information about the robustness of the method. What info the scenario provides is discussed in the specific paragraphs of the scenarios. Scenario 1 is the same as the one used in the calibration test. Scenario 2 features two slanted long obstacles. Scenario 3 is a dense field with many cylinders. Scenario 4 has a cylinder and a block. With this experiment we want to show the relationship between the parameters and the performance of the method. We repeat all tests 5 times and put the average of that time in our resulting graphs.

Scenario 1 is a good starting test, all calibrations will need to take the same path around the obstacle. The results of the test can be seen in Figure 6.8. It shows the relation between the wide valley parameter and the time it took for the robot to get to the target position. A relation can be seen in the grid, and we can conclude that a higher wide valley parameter will result in slightly more time. This is because the wide valley parameter influences the robot's path, taking a longer path with higher values. This can be seen in Figure 6.2 and Figure 6.4.

Scenario 2 has two slanted long obstacles which are placed in a way that the robot is required to drive left around the first obstacle and right around the second. The results of the test can be seen in Figure 6.9. It shows

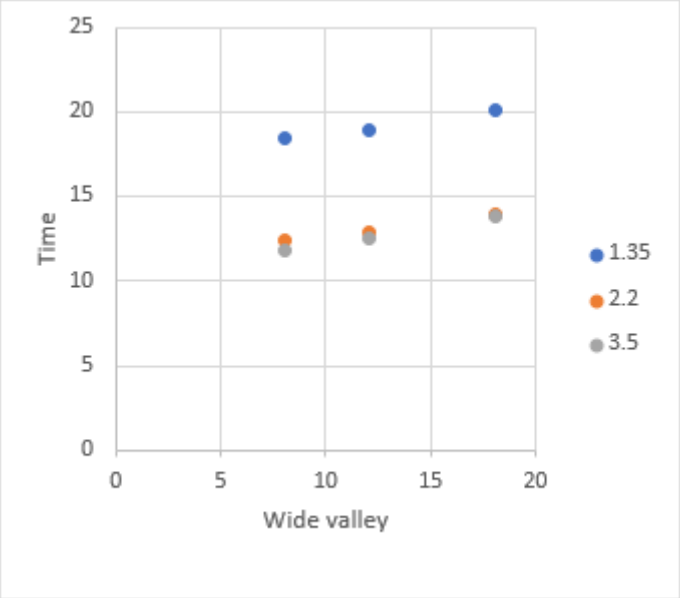


Figure 6.8: Results of the first scenario.

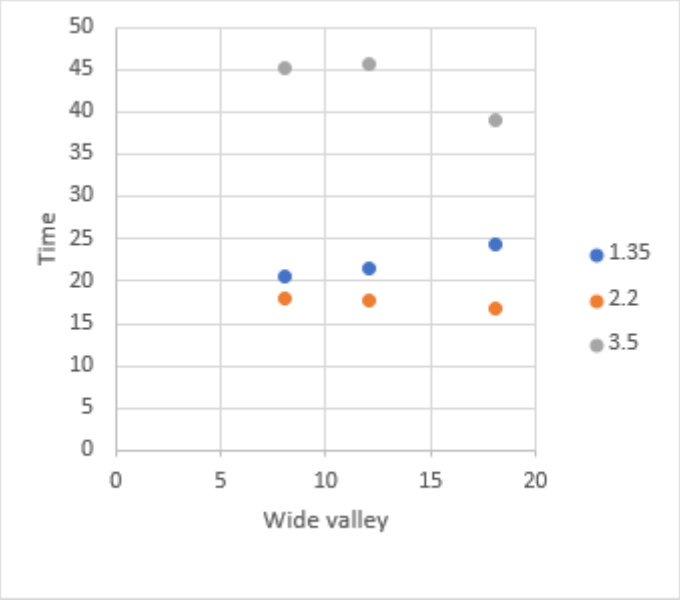
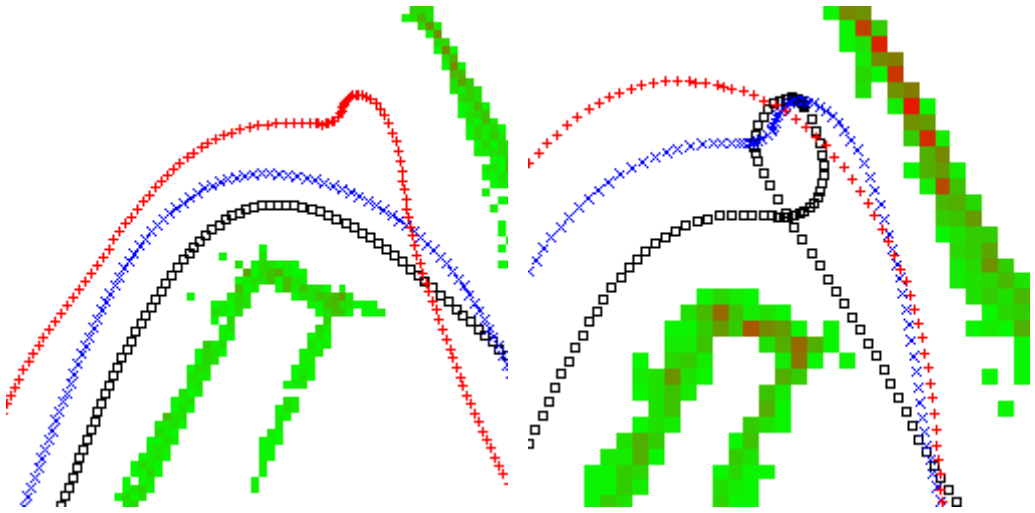


Figure 6.9: Results of the second scenario.

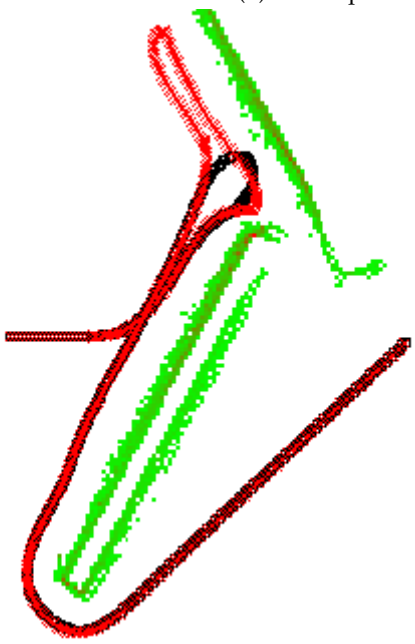
again the relation between the wide valley parameter and the time, but in this case the relation seems more erratic. This is because of the difference in paths the robot took.

Figure 6.10 shows the different paths the robot took. Though 1.35:$\frac{1}{3}$ and 1.35:$\frac{1}{2}$ take the intended route perfectly where the robot goes around the first obstacle on the left side and around the second on the right. First of all, 1.35:$\frac{2}{3}$ makes a slight S like move after it passed the first obstacle. Trying to find out what causes the S maneuver reveals that when the robot reaches the second obstacle the robot is still in ”left diversion” mode, so the robot makes a slight turn to the left. While it does this however, it moves forward just far enough to make it so there are no left valleys, which makes the robot switch to ”right diversion” mode, finishing the S shape and reaching the target position. A similar situation happens to 2.20:$\frac{1}{3}$ and 2.20:$\frac{1}{2}$, where the robot



(a) 1.35 m paths for the second scenario.

(b) 2.20 m paths for the second scenario.



(c) 3.50 m paths for the second scenario.

Figure 6.10: All paths of the second scenario.

first drives away from the target and the gap when they reach the second obstacle. 2.20:$\frac{1}{3}$ finally reaches a point where the robot turns left making a loop, while 2.20:$\frac{1}{2}$ simply turns right making a similar S maneuver. 2.20:$\frac{2}{3}$ heads straight into the gap. The 1.35 meter calibrations do take more time to reach the end position, as they use a lower maximum speed adjustment. The 3.50 m calibrations do not use the gap as it is too small for them. Therefore, they have to go around the first obstacle but the method does not know this, so it first has to find that the gap is too small and that there is no other way around the second obstacle. This results in two paths, one where the robot gets close enough to the second obstacle resulting in no possible valleys to the left, and the other, where the robot has to go to the edge of the area where the window slides over the boundary of the area in order to switch to "right diversion" mode resulting in a longer path. Both paths are taken by both 3.50:$\frac{1}{3}$ and 3.50:$\frac{1}{2}$, but 3.50:$\frac{2}{3}$ only takes the shorter path. For 3.50:$\frac{1}{3}$ and 3.50:$\frac{1}{2}$, we used the average of the longer path, not taking into account the shorter path. Though we can not really conclude anything from the

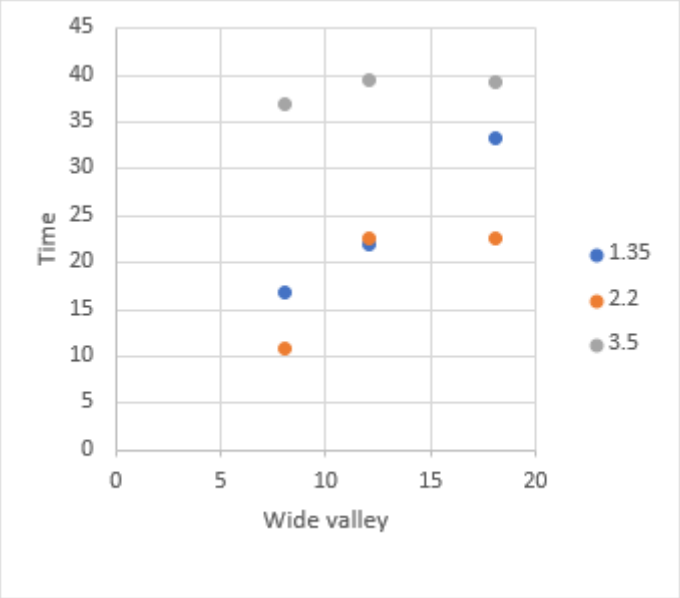


Figure 6.11: Results of the third scenario.

measured time we can conclude another problem with the path monitor. The path monitor prevents the robot to take the better path, which increases the time the robot needs to get to the target position, but in a different setup this might prevent the robot from ever reaching the target position. As we identified two problems with the path monitor we can conclude that the path monitor is one of the weaker parts of the method.

The third scenario has many deliberately placed cylinders. The first cylinders the robot will encounter will make the robot go down first. In the bottom portion of the scenario the cylinders are packed somewhat densely but spread out far enough that the robot can easily fit through. If the robot can not fit through it will loop back to the top portion. There, a specific part was left open to function as a possible path. The results can be found in Figure 6.11. The robot takes a different path for all of the 1.35 m calibrations, resulting in incomparable times.

The paths can be seen in Figure 6.12. 2.20:$\frac{1}{3}$ uses the same path as 1.35:$\frac{1}{3}$. 2.20:$\frac{1}{2}$ and 2.20:$\frac{2}{3}$ use the same path as 1.35:$\frac{2}{3}$, but the 1.35 calibrations use a lower maximum speed so they take a longer time to reach the target position. All of the 3.50 m calibrations take the top path. Because of all of these different paths the results do not show a real relation, but it is rather clear that the $\frac{1}{3}$ calibration takes the best path for every sensor range. This is the same relation we saw in scenario 1 so there might be a good relation, but scenario 2 did not show this relation, so we can not say for sure that the relation holds up every time.

Our fourth scenario is a very simple test. A block and a cylinder are placed in a way that the robot can easily fit through. This means that as we increase the calibration for every sensor range the robot should move more to avoid the obstacles. This results in a longer path and increases the time the robot takes to reach the end. The results can be seen in Figure 6.13. Again the $\frac{1}{3}$ calibrations are the fastest. As this has been the case for 3 of the 4 tests we will assume that in most cases this calibration will be the fastest. All calibrations have been equally reliable in our experiments, so the $\frac{1}{3}$ calibrations are the best calibration in our experiments.

(a) Path of 1.35:$\frac{1}{3}$.

(b) Path of 1.35:$\frac{1}{2}$.

(c) Path of 1.35:$\frac{2}{3}$.

Figure 6.12: Paths of the 1.35 m calibrations for the third scenario.

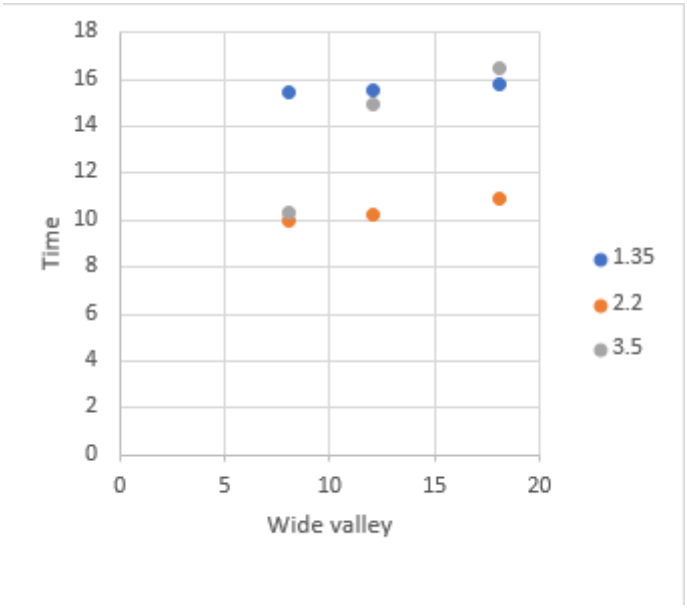


Figure 6.13: Results of the fourth scenario.

6.5 Results

With our results, we can now give values to all our parameters. For our robot we base some parameters on the robot used in [BK91b], others are based on formulas, and the last few parameter values were found with our calibration experiment. Most of our formulas use the sensor range, and we also know that the polar sectors and cell size are connected, and that the robot's distance to obstacles also plays a role.

With that said some points have to be made clear.

- Because of the permanent grid the robot can not handle mobile obstacles. They would just permanently be written down as a static obstacle.
- Our experiments pointed out that the path monitor causes most of the reliability problems, but it is necessary for a lot of scenarios.
- We had to slow the robot down for the 1.35 m sensor range as the method could not keep up. This might mean that we can speed up the robot for the 3.50 m sensor range, but we did not do this. The reason for this was that it would have made it harder to compare the results of our experiment, and seeing if an oversized window and sensor range in comparison to the other settings has any effect on the reliability or performance is more interesting for this thesis. It only resulted in a longer path.

And with these results we can now make a list of recommendations on how to set every parameter based on the sensor's properties:

- Update time: According to [BK91b] by decreasing the update time the robot can go faster. If we let the robot parse everything as fast as possible the robot could go faster, but this might cause the cell values to skyrocket, which might cause problems. We have not tested this however.
- Area size: As big as necessary. Keep in mind that a big field with small cells will take a big chunk of memory.
- Cell size: The size of your cells has to be set so that when your robot is driving along a wall it does not start to see "gaps" in the wall. Based on our experiments we propose that you should take $1/3$ of your sensor range and make sure that if a wall was in that position it would be detected about 6 to 10 cells away from the robot. This experiment works for 72 polar sectors. There should be at least twice as many cells in the 90° angle than there are polar sectors. You can find an approximation by dividing the amount of polar sectors by 10, then divide $1/3$ of the sensor range with the result. We have not tested a different amount of polar sectors, however, so we can not be sure if this works.
- Polar sectors: 72 seems to work fine. If you increase it you might have to reduce the cell size too.
- Smoothing factor: set it to $polar\ sectors/sensor\ amount$ as a bare minimum.
- Constants a and b: As mentioned in Chapter 4 you can set a to d_{max} and b to 1, though any other values should not change the behaviour of the robot as long as $a = bd_{max}$ holds.
- Constant Ω_{max}: Just set it to the fastest your robot can turn or 180 $^\circ$/s if it can turn faster than that.
- Minimum speed V_{min}: This one is still personal choice, for us 0.04 worked fine, but if your robot has a maximum speed of 0.10 m/s, you probably want to lower it. It is probably best to keep it around 5% of the maximum speed.
- Maximum speed of the robot: if we assume a simple relationship, you can calculate it with $\frac{sensor\ range+robot\ radius}{0.022 \cdot update\ time} = max\ speed$. We base this on the fact that our max speed was 1 m/s, our sensor range 2.20 m, and our update time was 100 ms, which holds up with the formula. For the 1.35 m sensor

range settings the formula holds up as well, and in the original paper they mention that if they halve their update time they could double their speed. However, they also mention that the relationship is actually quite complicated, so the maximum speed might be lower than the result of this formula. On the other hand the original robot was faster than what the formula would have as a result, so there is some leeway.

- Wide valley, Constant h_m and Threshold can be set with the polar experiment. From our experiments we conclude that the calibration that uses $1/3$ of the sensor range has the best results.
- Window size, we can simply use $\frac{sensor\ range}{cell\ size} / \sqrt{2}$ to calculate the window size.

Chapter 7

Conclusions

In our experiments, we found that the VFH method is reliable and performs very well if the right parameter settings are used in most cases. In order to find these parameter settings we made the polar calibration experiment. The calibration test that used $\frac{1}{3}$ of the sensor range was found to be the best for most cases, as it often took the best path possible and was slightly faster than the other calibration distances. The downfall of the VFH method is the path monitor, the monitor that makes sure the robot does not get stuck in certain conditions. Though it is absolutely necessary it often causes problems if a valley produces directions very close to the target direction. If the method is currently in a diversion mode and the valley switches sides in comparison with the target direction the valley will not be seen as a potential valley and the method chooses the next best valley. This is problematic with narrow valleys. We also found that the minimum gap size is related to the sensor range and cone angle. As we do not use real sonar sensors this might be wrong. Our sensors can detect the wall further away than real sonar sensors can, which means that the valley will be wider with real sensors. As the valley is wider the robot might be able to travel through even narrower gaps, but we can not test this with our setup.

Chapter 8

Future Work

Due to the results of our evaluation we now have an understanding of the robustness of the VFH method. There are still a few things that can be tested however. First of all the experiments could be redone but with actual sonar sensors. Another scenario that could be tested is if laser sensors instead of sonar sensors would influence the performance or reliability. We also quickly went over the smoothing factor in this thesis, and the side effects of this parameter could be researched. Other methods to find appropriate values for the parameters can also be found.

Aside from researching VFH further other experiments could be done with VFH+ and VFH*. One of those experiments could test if VFH+ is a proper improvement of VFH, solving the issues we found during our experiments. As most of our problems were caused by the path monitor, and VFH+ does not use the path monitor, VFH+ is likely better, but we do not know if the problems are actually circumvented or new problems are introduced.

Different methods based on VFH can be made, like VFH+ and VFH*. As mentioned in this thesis the sonar sensors become less accurate at longer ranges, which means that the accuracy of the grid is lower with longer sensor ranges. The detected range could be taken into account when changing the values of the grid. Another improvement that could be made is to take into account the sensors in the polar histogram. Using the detected ranges from the sensors directly in the polar histogram might help with ignoring inaccurate grid cells, and might even allow the method to deal with moving objects.

We had the issue of the 1.35 meter sensor range going too fast, which caused it to get very close in our tests. We solved this by lowering the robot's speed so the robot had more time to detect obstacles. However, the robot does not really need time, it just needs the histogram grid and the values inside it, which are filled with the sensors, that run at 10 Hz with our robot. Maybe increasing the sampling frequency of the sensors would have solved our problem as well.

All of the experiments done by the initial VFH developers use robots with 24 ultrasonic sensors, and for this paper we used a robot with 24 ”ultrasonic” sensors as well, so the effect of more, less or using different sensor types can still be researched for any VFH type.

Bibliography

[BAM18] Mordechai Ben-Ari and Francesco Mondada. *Robotic Motion and Odometry*, pages 63–93. Springer International Publishing, Cham, 2018.

[BK89] J. Borentstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Transaction on Systems, Man, and Cybernetics*, 7:1179–1187, 1989.

[BK91a] J. Borenstein and Y. Koren. Histogramic in-motion mapping for mobile robot obstacle avoidance. *IEEE Transactions on Robotics and Automation*, 7(4):535–539, Aug 1991.

[BK91b] J. Borenstein and Y. Koren. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7:278–288, 1991.

[BSV98] Alexandre Bernardino and Jos Santos-Victor. Visual behaviours for binocular tracking. *Robotics and Autonomous Systems*, 25(3):137 – 146, 1998. Autonomous Mobile Robots.

[Che] *Turtle: 2WD Arduino Mobile Robot Platform*. `https://www.dfrobot.com/product-65.html`.

[CKK] Eric Chown, Stephen Kaplan, and David Kortenkamp. Prototypes, location, and associative networks (plan): Towards a unified theory of cognitive mapping. *Cognitive Science*, 19(1):1–51.

[CRFS] Joseph Carsten, Arturo Rankin, Dave Ferguson, and Anthony Stentz. Global planning on the mars exploration rovers: Software integration and surface testing. *Journal of Field Robotics*, 26(4):337–357.

[DK02] G. N. Desouza and A. C. Kak. Vision for mobile robot navigation: a survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(2):237–267, Feb 2002.

[FBT97] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, March 1997.

[Gaza] *Gazebo Installation tutorial*. `http://gazebosim.org/tutorials?tut=install_ubuntu&cat=install`.

[Gazb] *gazebo_ros_pkgs Installation tutorial*. `http://gazebosim.org/tutorials?tut=ros_installing&cat=connect_ros`.

[Gazc] *http://gazebosim.org/*. `http://gazebosim.org/`.

[GJZ+97] P. Gaussier, C. Joulain, S. Zrehen, J. P. Banquet, and A. Revel. Visual navigation in an open environment without map. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97*, volume 2, pages 545–550 vol.2, Sept 1997.

[G13] Mehmet Serdar Gzel. Autonomous vehicle navigation using vision and mapless strategies: A survey. *Advances in Mechanical Engineering*, 5:234747, 2013.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[HSA89] R. C. Harrell, D. C. Slaughter, and P. D. Adsit. A fruit-tracking system for robotic harvesting. *Machine Vision and Applications*, 2(2):69–80, Mar 1989.

[IRo] *IRobot Create manual*. `https://www.irobot.com/filelibrary/create/Create\%20Manual_Final.pdf`.

[KB91] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 1398–1404 vol.2, April 1991.

[KK92] Akio Kosaka and Avinash C. Kak. Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties. *CVGIP: Image Understanding*, 56(3):271 – 329, 1992.

[ME85] Hans Moravec and A. E. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 116 – 121, March 1985.

[rep] *Ubuntu repositories*. `https://help.ubuntu.com/community/Repositories/Ubuntu`.

[ROSa] *http://www.ros.org/*. `http://www.ros.org/`.

[ROSb] *ROS Installation tutorial*. `http://wiki.ros.org/kinetic/Installation/Ubuntu`.

[Squ] *Largest size of a square inside a circle*. `http://mathcentral.uregina.ca/QQ/database/QQ.09.04/bob1.html`.

[Tho83] Charles E Thorpe. An analysis of interest operators for fido. 1983.

[UB98] I. Ulrich and J. Borenstein. Vfh+: reliable obstacle avoidance for fast mobile robots. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, volume 2, pages 1572–1577 vol.2, May 1998.

[UB00] I. Ulrich and J. Borenstein. Vfh*: local obstacle avoidance with look-ahead verification. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 3, pages 2505–2511 vol.3, April 2000.

Appendix A

Getting started

We go through all the steps that are necessary to install Gazebo and ROS and explain how to use the ROS implementation of VFH. This thesis comes with a ROS package that contains all the necessary files.

A.1 Software versions

For this thesis we used the following software:

Ubuntu	16.04 LTS
ROS	distribution kinetic, version 1.12.13
Gazebo	version 7.0.0

Our implementation was evaluated with these software versions. ROS and Gazebo are currently very active projects that are constantly updating, and sometimes a package gets deprecated or a bug occurs in the newer versions. Our implementation might work on newer versions, but should always work with the versions above. Extra care should be taken when installing ROS. Every version of ROS works on a specific version of Ubuntu. A newer or older version of Ubuntu than version 16.04 LTS most likely will not support ROS kinetic 1.12.13. All instructions were taken from the ROS installation page [ROSb], Gazebo installation page [Gaza], and gazebo_ros_pkgs installation page [Gazb]. If problems occur during the installation process check the specific page to see if they changed the installation procedure.

A.2 Installing ROS and Gazebo

To install ROS kinetic we have to go through multiple steps. All these steps are taken from [ROSb].

Before we begin installing we have to make sure that Ubuntu allows the ”restricted”, ”universe”, and ”multiverse” repositories. Steps on how to do this can be found in [rep].

In order to install ROS kinetic the following bash commands have to be executed:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
    main" > /etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv\-key
    421C365BD9FF1F717815A3895523BAEEB01FA116
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-desktop-full
$ sudo rosdep init
$ rosdep update
$ sudo apt-get install python-rosinstall python-rosinstall-generator python-
    wstool build-essential
```

We have taken the Gazebo installation steps from [Gaza]. In order to install Gazebo we have to execute the following bash commands:

```
$ sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable
    `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
$ wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install gazebo7
$ sudo apt-get install libgazebo7-dev
```

The next steps were taken from [Gazb]. Before we install gazebo_ros_pkgs we need to make sure that ROS and Gazebo are functioning properly. For gazebo we can simply use

```
$ gazebo --version
```

If the version of gazebo is displayed gazebo is installed properly. If an error pops up something went wrong. ROS needs one more command to test:

```
$ source /opt/ros/kinetic/setup.bash
$ rosversion -d
```

If the last command displays "kinetic" ROS is properly installed.

To install gazebo_ros_pkgs we use the following bash command:

```
$ sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-ros-
    control
```

After it is done installing we can test if it was successful by running the command:

```
$ roslaunch gazebo_ros empty_world.launch
```

If Gazebo launches, the packages are installed properly. Note that the ROS environment has to be set up in order for this to work. Look at how we test if ROS is properly installed for the command to set bash in the right environment.

A.3 Building the package

Before you can build the package you need to create a catkin workspace. This can be done with a few commands. Before you start, navigate to the folder where you want your catkin workspace to be. Create a folder called ”catkin_ws”, and a folder inside the catkin_ws folder called ”src”. Make sure your terminal is setup properly with the ROS environment, otherwise the commands will not work. Then run the following commands:

```
$ catkin_make
$ source devel/setup.bash
```

This new source command adds more to the ROS environment, and should be the only one necessary after this.

With our ROS environment properly setup we now need to copy the content of the attached ”src” folder into the src folder of your catkin workspace. In order to then build the package we only need to execute the command

```
$ catkin_make
```

A.4 Running the VFH implementation

With everything now properly setup you can use the VFH implementation. In order to use it you need to run a few commands.

```
$ roscore
$ export GAZEBO_MODEL_PATH="$(pwd)/src/vfh/models":$GAZEBO_MODEL_PATH
$ roslaunch vfh empty.launch
$ rosrun vfh vfh
```

These commands need to be run every time you need to use the robot. ”empty.launch” can be replaced by another world. Note that the robot has been given the name RAY1 in the world. This is important as the ROS layer is subscribed to ”/RAY1/odom” and ”/RAY1/debug”. The sonar sensors are named ”/sonar_1”, ”/sonar_2”, ”/sonar_3”, ..., ”/sonar_24”. If you want to place the robot in another world you need to make sure that the odometry sensor is called ”/RAY1/odom” for the ROS layer to work correctly. ”/RAY1/debug” is only used for emergency stops and is not required for the robot to function properly. The ”vfh.h” file in ”vfh/include” contains the parameter values.

A.5 Using VFH with other robots

The supplied ROS implementation only supports very similar robots. The amount of sensors is limited to 24 and assumes that every sensor is spaced out evenly. In order to use the VFH method for other robots changes have to made to the code. If the other robot uses 24 sensors setup in the same way as our evaluation robot only some of the parameters have to be changed. For completely different robots the ROS layer needs to be rewritten, and possibly a part of the VFH method.

To implement the VFH implementation into a different robot or even a non ROS robot a few methods have to be called in order to use the method. The method has its own class, VFH. The necessary parameters can be set using setters. All setters update 1 parameter, aside from constant a and b. There is also a convenience function that sets all the parameters at once. The target position can be set using the function "setTargetCoordinates(target_x, target_y)". The function "setRobotCoordinates(robot_x, robot_y, yaw)" can be used to update the robot's position and heading. The range sensors should be parsed by the function "parseMeasurement(range, angle)", where the range is the detected range, and the angle is the direction of the sensor including the robot's heading. In order to compute the desired direction and desired speed the robot needs to call a few functions. First the best valley direction can be found by calling "calculateDirection()". This function returns an angle between 0 and 2π radians. If no directions could be found the function returns $4 * \pi$, so if this is the case the robot has to stop. Note that it might find a direction the next time the function is called. After you found the direction you need to find the steering rate, based on the difference with the target direction. You can use the function "angularDifference(found direction, target direction)" for this. Then you need to find the desired speed by multiplying the maximum speed adjustment with the function "getSpeed(steering_rate)" and adding the minimum speed.

A.6 Package files

Aside from the necessary files to run the VFH method there are a few other files in the package. The most important files aside from the method files are the gazebo_sensor_ray_plugin files and the robot model files. The plugin files are necessary for the robot's sonar sensors. Without those files the evaluation robot does not work.

There are also a couple of world files in the worlds folder. They can be launched through the launch files. All the worlds include the robot and a specific scenario. The worlds are the ones we used for the experiments.

There is also a debugging version of the VFH implementation included. It has a couple of printing functions to print the data representations of the method. It also changes a few methods to keep track of the robot's location.