Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Kristoffer Nesland

Mobile Robot for Inspection on an Unmanned Offshore Production Platform

A proof of concept, developed with ROS

Master's thesis in Mechanical Engineering
Supervisor: Olav Egeland
June 2019

NTNU
Kunnskap for en bedre verden

Kristoffer Nesland

Mobile Robot for Inspection on an Unmanned Offshore Production Platform

A proof of concept, developed with ROS

Master's thesis in Mechanical Engineering
Supervisor: Olav Egeland
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Acknowledgements

I would like to thank my supervisor Olav Egeland for the interesting and relevant challenges he has given to us students. It has been motivating to work on the research field of mobile robots. A field that is in rapid evolution thanks to the autonomous car industry. Further, my co-supervisor Gunleiv Skofteland at Equinor has been essential for defining the scope of the thesis. His experience from the O&G industry has significantly influenced the industry relevance of this project. My gratitude also goes to Cecilia Haskins for showing interest in the project and for sharing valuable insight on the testing process.

Lastly, the comradeship at Valgrinda has been much appreciated. Dag, Eirik, Trym, Hans Kristian and Didrik have all helped keeping the spirit strong and contributed to the thesis. What to include in the thesis, how to position figures side-by-side in LaTeX, the steps of making a cold brew coffee and a lot more has been discussed. Thanks to all of you and thanks to my family!

Abstract

Discoveries of offshore oil and gas reservoirs far from the shore, present new challenges for the industry. Transportation costs increase as the distance to the coastland increases and the risk of stationing human operators far out in the sea is considerable. This calls for new thinking in the industry. Unmanned offshore production platforms can be the solution to the mentioned challenges. However, to design a totally maintenance free platform is impossible. This is where the idea of a mobile inspection and maintenance robot enters. If designed properly, it should reduce the need for human intervention on the platform and let the operators stay on the shore as long as possible.

This thesis focuses mainly on the inspection features of a mobile robot. The Robot Operating System (ROS) and the TurtleBot3 mobile robot is used to experiment with methods for detection of gas leakages and unwanted physical objects on the platform. All the implemented functionalities are verified through experiments in a room designed to replicate some of the features found on an unmanned production platform. Path planning and execution is achieved with the ROS Navigation stack and additional experiments are conducted to assess the performance of this software in an environment that will be relevant for an inspection robot on an offshore platform. As can be interpreted from this, the focus is on the software design and not the physical design of the robot.

Testing of the features implemented in this thesis show promising results. The robot is able to generate a heatmap of the gas concentration based on simulated measurements. Also the novel obstacle detection shows promise and is able to estimate the location of 17 out of 19 obstacles within 0.3 m given data from the TurtleBot's LiDAR. However, it must be taken into account that these tests are performed in a controlled environment and many unforeseen challenges may become apparent on a real platform. All in all, the thesis serves as a proof of concept and showcases some of the useful features that can be quickly developed with ROS given a mobile robot with a LiDAR, cameras and a gas detector.

Sammendrag

Oppdagelser av nye offshore olje og gass felter lokalisert langt fra kysten byr på nye utfordringer for industrien. Transportkostnader øker i takt med økt avstand til land og risikoen ved å stasjonere mennesker langt ut på havet er betydelig. Dette inviterer til nytenking innen bransjen. Ubemannede offshore produksjonsplattformer kan være løsningen på de nevnte utfordringene. Allikevel så vil det være umulig å designe en helt vedlikeholdsfri plattform. Det er her ideen om en mobil inspeksjon- og vedlikeholdsrobot kommer inn. Ved riktig design skal en robot kunne redusere behovet for menneskelige inngrep på plattformen og la operatører bli værende på land så lenge som mulig.

Denne oppgaven fokuserer i hovedsak på inspeksjonsegenskapene til en mobil robot. Robot Operating System (ROS) blir tatt i bruk sammen med den mobile roboten TurtleBot3 for å eksperimentere med metoder for deteksjon av gasslekkasjer og uønskede fysiske gjenstander på plattformen. Den implementerte funksjonaliteten blir testet gjennom eksperimenter i et rom innredet for å etterlikne noen av trekkene man vil finne igjen på en ubemannet produksjonsplattform. Planlegging av rute og utførelse er oppnåd med ROS Navigation stack og en rekke eksperimenter er utført for å teste ytelsen til denne programvaren i omgivelser som vil være relevante for en inspeksjonsrobot på en offshore plattform. Som man kan forstå ut i fra dette, så vil fokuset være på design av programvare, ikke på det fysiske deignet av roboten.

Testing av de implementerte funksjonene i denne oppgaven viser oppløftende resultater. Roboten er i stand til å generere et kart over gass-konsentrasjon basert på simulerte målinger. Også den nyutviklede metoden for å detektere hindringer ser ut til å fungere bra og systemet var i stand til å detektere 17 av 19 hindringer med en nøyaktighet på dårligst 0.3 m, gitt data fra robotens LiDAR. Det må merkes at testene er gjennomført i et kontrollert miljø og mange uforutsette utfordringer kan vise seg på en ekte plattform. Alt i alt så fungerer oppgaven som et "proof of concept" og belyser noen nyttige funksjoner som kan raskt utvikles i ROS, gitt en mobil robot med LiDAR, kameraer og gass-sensor.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

1.1. Background and Motivation

This Master's thesis is written in cooperation with Equinor ASA and is based on the Peon project. Peon will be an unmanned oil and gas production platform on the Norwegian continental shelf. Recent trends suggest that unmanned platforms, like Peon, will become influential in the years to come [23]. As new production fields are opened further away from the shore than previously, it becomes more costly to transport operators and equipment to and from the platforms. This is especially true for smaller reservoirs since, in these cases, keeping the cost of building and operating the platform low is even more important to keep the break-even price low. Unmanned platforms also increase the safety of the production as human operators spend less time in the harsh offshore environment.

Unmanned platforms present new challenges for inspection and maintenance. Human operators are versatile and able to carry out a variety of tasks on a platform. Testing of smoke detectors, replacement of parts, tightening of screws and cleaning are among the tasks that must be performed regularly. The need for these tasks can be reduced through appropriate design of an unmanned platform. "Equipment requiring periodic inspection or re-certification shall to the extent possible be avoided" [23]. The mean time between failures (MTBF) should be maximized through the use of high quality materials and only including the systems that are needed the most. Mean time to repair (MTTR) should be minimized by having a modular design where whole modules can be replaced. However, there will still be need for occasional repairs.

For daily inspection and smaller repairs, a mobile robot can be useful. Equipped

with a camera, gas detector and more, the robot can monitor the platform and detect deviations. For example, if a gas alarm triggers, the robot can navigate to the reported area and perform its own independent measurement. If it was just a false alarm, operation can continue as normal or at least until human operators have the time to travel to the platform. Being able to continue the operation in these situations can have a great economic impact.

1.2. Thesis Outline, Videos and Code

Chapter 2 presents previous research in the field of mobile robots on O&G facilities. Some of the main challenges that a robot will face on an offshore platform are highlighted before the scope of this project is defined. From here on, the physical design of the robot is not in focus and the spotlight is shifted to the software.

Chapter 3 lays the theoretical foundation for the remaining of the thesis. As autonomous navigation is a crucial part of autonomous robots, the focus is on explaining much of the essential theory in this research field. However, the Robot Operating System (ROS) has been used extensively throughout this project and the execution of the navigation is handled by already available ROS packages.

Chapter 4 presents the robot used in the project and some of the most important concepts of ROS.

Chapter 5 describes the environment that was designed to test the robot. Furthermore, it explains how the features developed in this project are combined with already existing ROS functionalities to form five use-cases of the robot.

Chapter 6 and 7 explain the new functionalities developed in this project. All these functionalities are developed as extensions of ROS. Chapter 6 focuses on measurement of gas concentration while chapter 7 is concerned with mapping of obstacles. The chapters are independent of each other and deal with the developed methods, experiment setup, results, discussion and suggestions on further work.

Chapter 8 presents additional tests that were conducted to challenge the robot's navigational capabilities.

Chapter 9 combines the contents of chapter 6, 7 and 8 to conclude on the work in this thesis

The appendix contains tables with the results from the experiments, some additional theory and a description of how the ROS package developed in this project is structured. It also includes a link to the GitHub repository where all the code developed during the project can be found.

Because a great part of this thesis was devoted to write code and perform testing, videos from the testing can be found in the attachments of the thesis. The videos aim to give the reader a better understanding of how the developed system works as a whole.

Chapter 2.

Mobile Robots in the O&G Industry

This chapter starts out wide by presenting several different mobile robots that have been developed specifically for use in the O&G industry. Some aspects one needs to have in mind when designing a robot for this use are presented before the focus shifts to possible applications. Lastly, three main topics are chosen and the rest of the thesis focuses on these from a software perspective.

2.1. Related Work

Remotely operated vehicles (ROVs) have been utilized within the offshore O&G industry for several decades already. Replacing a human diver, an underwater ROV greatly reduces the risk of inspecting subsea installations. With teleoperation, the operator can stay above the waterline and control the ROV through the umbilical. In the most simple case, the ROV is only equipped with a camera and propellers, effectively giving the operator a moving underwater camera. These simple systems are still in use and can for example be used to inspect pipeline leakages [32]. However, more complicated ROVs [25] have evolved. These typically include manipulators that are able to grab onto structures, move components, turn valves and more. Further, autonomous underwater vehicles (AUVs) are now increasingly being used. These are robots that can perform pre-programmed tasks without needing human input during the operation.

Despite extensive submarine applications, ROVs have not been utilized topside to any major extent. As there are human operators on the oil rigs, there has not been

Figure 2.1.: Sensabot [34].

a need for this. However, changes are imminent as ideas of unmanned platforms are planted. Bindingsbo had a vision of a robotized field operator back in 2009 [2]. "An orange robot moves around the process site, performs a combination of routine inspections, and replaces a safety valve. This robot works along side two others. A human operator located hundreds of miles away in the process control center supervises all three". What is described here is a general-purpose autonomous robot operating on an unmanned production platform. Transeth [38] took a step in this direction and worked on a system for remote inspection and maintenance (I&M). Using 3D vision, the system was able to replace batteries in an industry standard Rosemount wireless sensor.

Sensabot [34] was among the first mobile robots to be tested at an O&G facility. It is a ROV that is designed to work in flammable and explosive environments. The extendable arm seen in Figure 2.1 allows it to perform sensor measurements at different heights and angles. Furthermore, it can work across multiple floors by moving onto a rail to ascend or descend. The Taurob Tracker [36] is a newer alternative and is also designed for hazardous missions. Unlike Sensabot, the Taurob Tracker utilizes belts and is said to have a good maneuverability, even on rough surfaces. It is able to pass 35 cm tall obstacles and move in stairs that are up to 45° steep. Both the robots are intended to operate in potentially explosive atmospheres (ATEX).

DORIS [5] is a concept that differs greatly from the previously mentioned robots. It is describes as "an autonomous rail-guided robot designed for inspection and

Figure 2.2.: DORIS [5].

monitoring of Oil and Gas (O&G) facilities". Like seen in Figure 2.2, the robot moves along a fixed rail that is laid in a loop around the facility. This greatly simplifies the autonomous navigation problem as obstacle avoidance and planning is not needed. A key idea behind DORIS is that to have multiple sensor in a movable robot reduces the need for fixed sensors at the facility. Laser scanner, HD camera, stereo camera, infrared camera, microphone, gas sensor and manipulator are among the features that make DORIS powerful. The robot is able to detect anomalies by comparing the current incoming signals to signals that were recorded during normal operations. This way, it can detect when objects have been moved and if the machinery at the facility emits irregular sound. Through tests in an industrial environment, DORIS proved capable of building 3D maps from laser scans and detect anomalies in audio and video.

A company that has caught a lot of attention lately is Boston Dynamics. Contrary to the aforementioned robots, this company's robots are mostly without wheels or tracks. One of their most production ready robots, the SpotMini [10], is a so called "legged" robot. Its four legs enable it to handle rough terrain and use stairs. Additionally, as can be seen in Figure 2.3, it has a 5 degree-of-freedom arm which lets it pick up objects, operate door handles and more. Like the other robots, SpotMini also comes with a 3D vision system. As legged robots like the SpotMini become more mature, they can become alternatives to robots like Sensabot, DORIS and the Taurob Tracker.

Figure 2.3.: Boston Dynamics' SpotMini [10].

2.2. Offshore Considerations

The conditions at offshore platforms are known to be harsh. Powerful wind, water splashes, salty air with high humidity and direct sunlight are some of the challenges that both humans and machinery face offshore. In arctic environments there is even an additional risk of ice building up on structures. These are all challenges related to the geographical position of the platforms, but also the layout of the platform must be considered when working on an autonomous robot.

Bengel [1] has worked on identifying challenges related to the layout of a typical offshore platform. The use of gratings as floor is characteristic and should be taken into account. A robot will need wheels with a sufficient size to move on the gratings. Also, slopes can be present and a material with high friction against steel should be used as tire to avoid slippage. Slippage can cause excessive drift of the odometry and in the worst case make the robot incapable of reaching certain positions. Another key challenge is the limited space available on platforms. Bengel operates with a minimum corridor width of 0.7 m and the size of the robot is hence restricted. A last challenge that is highlighted in the paper is the complex structures that can be found at offshore installations. Intricate stairways, pipelines, flanges, tanks and fences make it harder to navigate than in simpler environments with flat walls. Robust sensor systems are required to make out what are obstacles and what are passages.

2.3. Identified Use-Cases

Inspection will be the main task of the robot. It should search for anomalies, like gas leakages, on the platform. Furthermore, it may even be used for tightening valves and cleaning tasks. In conversation with Equinor ASA a number of use-cases like these were identified. These use-cases are explained in the following.

Teleoperation is convenient to have available in a mobile robot. Having direct control of the robot's actuators make it flexible to perform tasks that are not pre-programmed. Also, in case the robot's autonomous navigation breaks down, teleoperation is useful. Closely connected to teleoperation is the ability to stream the robot's camera feed to the onshore operators. This is a prerequisite for the operators to know how to remotely control the robot. Furthermore, it is impossible to account for all possible situations when programming the robot and operators watching the video stream can notice interesting situations that are not picked up by the robot. Equipped with a camera, the robot can also monitor gauges. One can imagine that sensors should automatically transmission their state over a network, but if this stops working, the robot can read of values.

A robot equipped with a manipulator will have extended use-cases. Instead of fixing a large number of sensors on the robot, it is possible to store sensors at a station and let the robot pick up the sensors when needed. Gas sensors, microphones and infrared cameras are not needed all the time and can be stored safely when not in use. Further, a manipulator will allow the robot to take samples and move equipment on the platform. If equipped with a rotational tool, it is also possible for the robot to open and close valves. Normally, such an operation will be performed by actuators built into the valve, but if this actuator fails it is convenient that the robot can operate the valve. Another case where a manipulator will come into use is if there are multiple robots on the platform. If one robot is stuck, runs out of battery or similar, another robot can come to the rescue and help the robot back in service.

Autonomous operation is necessary to minimize the need for human intervention and thus increasing the efficiency. The robot should be able to navigate autonomously. This requires the robot to localize itself on a map, plan a route to a goal and move to the goal while avoiding collisions. Pre-defined inspection routes can be programmed and the robot can follow routes regularly and control that normal operation is maintained. If deviations are found, the robot should mark this on a map and report back to the operators. In the simplest case, the robot can transmit an image of the deviation to the operators, but also image classification techniques can be used to detect typical fail modes and automati-

cally classify the type of fail. Deviations can be everything from objects that have loosened to nests built by birds on the platform.

2.4. Scope of the Thesis

The work in this project has been highly iterative and experimental. A fixed scope was not set when the work begun and the focus in the beginning was to see what was possible with a mobile robot on an unmanned offshore platform. After some weeks, it was decided to focus on three main topics that were found especially interesting in the initial phase of the project.

Topic number one is detection of gas leakages. How can a mobile robot equipped with a gas detector and an infrared camera detect gas leakages? The second topic is concerned with the detection of unwanted physical objects on the platform, like bird nests and built up ice. How can a mobile robot give the onshore operator an overview of possible unwanted objects on the platform? The last topic is more open and is concerned with the navigational capabilities of the robot. What are the potential challenges when it comes to the robot's ability to reach all parts of the platform?

This project does not focus on the development of a physical robot, but rather on demonstrating possible applications through the development of software. Therefore, it will not be laid significant focus on discussing waterproof equipment, choice of material for tyre, a compact design and so on. However, it is important to note that the conditions faced on an offshore platform are different from those found in, for instance, an indoor factory.

Chapter 3.

Robotics and Image Processing

The Robot Operating System (ROS) is used to develop the needed applications in the thesis. Many useful features are already available in ROS and this chapter will present the theory needed to understand how the ROS features used in this project work. Several solutions to robotics' problems such as navigation exist, but the focus will be on the solutions that are implemented in the ROS Navigation stack. In general, these methods are widely used and well-proven. Most of them are described in detail by Thrun [37] and several of the figures in this chapter are adopted from this book. Additionally, this chapter presents some image processing theory, mainly focusing on morphological methods. Methods that are developed during this thesis will be presented in later chapters.

3.1. Camera Model

The camera model explains how a point in space is projected onto an image captured by a camera. Ultimately, the goal is to relate a point in space to a specific pixel position in the image. Let a point be defined by its position relative to the camera frame,

$$\boldsymbol{r}_{cp}^{c} = \begin{bmatrix} x \\ y \\ z \end{bmatrix},$$

where the camera frame is a coordinate frame fixed to the camera. Often, points will be described relative to some other frame that is typically fixed in the environment. This can for example be a frame fixed in the corner of a room. Let this frame be called the object frame. Given that we know the position of a point in the object frame, a homogeneous transformation matrix between the object space

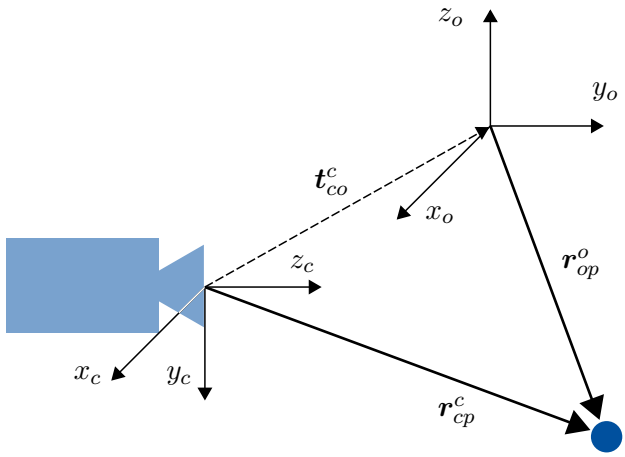


Figure 3.1.: The geometry of the camera frame and the object frame.

and the camera space is needed,

$$T_o^c = \begin{bmatrix} R_o^c & \boldsymbol{t}_{co}^c \\ \mathbf{0}^T & 1 \end{bmatrix}.$$

R_o^c is a rotation matrix that captures the orientation difference between the object frame and the camera frame. $\boldsymbol{t}_{\boldsymbol{co}}^{\boldsymbol{c}}$ is the translation from the camera frame to the object frame in the coordinates of the camera frame. The geometry can be seen in Figure 3.1. In total, the homogeneous transformation matrix T_o^c allows for a coordinate transformation from the object space to the camera space,

$$\tilde{\boldsymbol{r}}_{cp}^c = T_o^c \tilde{\boldsymbol{r}}_{op}^o. \tag{3.1}$$

When we now have established a transformation from the object frame to the camera frame, the next step is to normalize the camera frame coordinates. $\tilde{\boldsymbol{s}}$ is introduced as the normalized image coordinates,

$$\tilde{\boldsymbol{s}} = \begin{bmatrix} \frac{x}{z} \\ \frac{y}{z} \\ \frac{z}{z} \\ 1 \end{bmatrix} = \frac{1}{z} \boldsymbol{r}_{cp}^c, \tag{3.2}$$

where x, y and z are the elements of $\boldsymbol{r}_{cp}^c$. This normalization is a prerequisite before we can perform the last step of the projection onto the image plane.

The last step is to relate the normalized image coordinate to a pixel in the image. Because cameras can have different lenses and different image sensors, this

transformation will vary from camera to camera. Let u_0 be the number of pixels from the left edge of the image to the center and v_0 be the number of pixels from the top to the center. ρ_w is the horizontal width of a pixel and ρ_h is the vertical height of a pixel. f is the focal length, the distance from the camera frame to the image plane. Together, these form the five intrinsic parameters of the camera and they make up the camera parameter matrix,

$$K = \begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix}.$$

All that remains now is to calculate the actual pixel positions,

$$\tilde{\boldsymbol{p}} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\tilde{\boldsymbol{s}}. \tag{3.3}$$

Note that these pixel coordinates should be rounded to obtain the final set of two integers that tell us at which pixel position we can find $\boldsymbol{r}_{op}^{o}$ in the image. The whole procedure can be summarized as (3.1), (3.2) and finally (3.3).

3.2. Differential Drive Kinematics

Kinematics is the study of motion. Contrary to dynamics, it does not incorporate forces into the equations, only the geometrical constraints of the system. Further, one can separate between direct and indirect kinematics. As shown in Figure 3.2, direct kinematics is the calculation of how the robot's pose changes given the control variables. Inverse kinematics is the opposite, it is the process of finding the control variables that lead to a given change of the robot's pose. Direct kinematics is needed for odometry. Odometry is the use the data from motion sensors to calculate the change of pose over time. This way, one can estimate a new pose of the robot if the old pose and how the wheels have moved since the old pose is known. Inverse kinematics is useful for path planning. If the robot is set to follow a route, it should solve the problem of how to rotate the wheels to successfully follow the route.

The pose, or configuration, of a mobile ground robot is completely described by

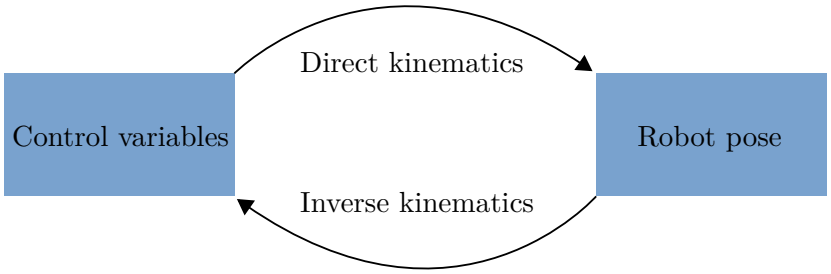


Figure 3.2.: Direct and inverse kinematics.

a vector,

$$\boldsymbol{q} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}.$$

If we consider a unicycle, the configuration of the robot can be visualized as in Figure 3.3. (x, y) are the Cartesian coordinates of the center point of the wheel and θ is the orientation of the wheel. Let v be the modulus (with sign) of the contact point velocity vector (between wheel and surface) and let ω be the angular speed around the vertical axis. Then, Siciliano [22] describes the motion of the unicycle by,

$$\dot{\boldsymbol{q}} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega.$$

Together, v and ω form the control variables of the unicycle model. However, most mobile robots are not unicycles.

An alternative to a unicycle is the differential drive robot. It is a robot characterized by two wheels on the same fixed axis which have independent motors and a third supporting wheel, Figure 3.4. If we simplify the differential drive robot by replacing the two driving wheels with a single wheel in the middle of the two, we are back at the unicycle model. Because the unicycle has the two control variables v and ω while the differential drive robot has the control variables v_L and v_R (Figure 3.4), the relationship between the two is needed. For a robot with distance d between the wheels, the control variables of the differential drive is transformed into the control variables of the unicycle by the two following equations,

$$v = \frac{v_R + v_L}{2} \qquad \omega = \frac{v_R - v_L}{d}. \tag{3.4}$$

The equations show that to drive in a straight line ($\omega = 0$), we have $v_R = v_L$. To rotate on the spot ($v = 0$), we have $v_R = -v_L$. Also, solving for v_L and v_R, the

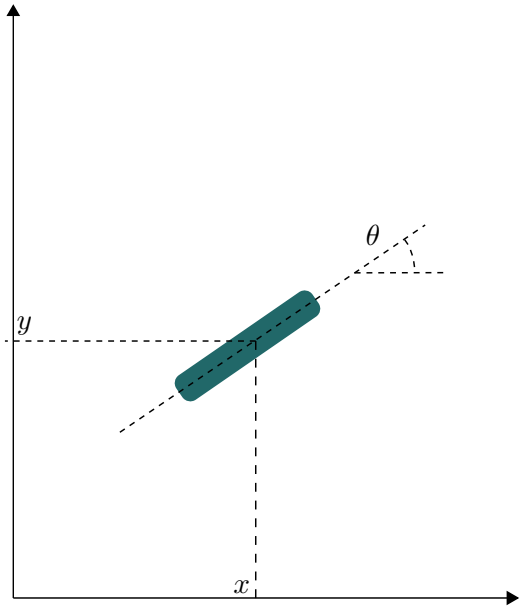


Figure 3.3.: The pose of a unicycle.

opposite transformation is obtained,

$$v_L = v - \frac{d}{2}\omega \qquad v_R = v + \frac{d}{2}\omega. \tag{3.5}$$

Equations (3.5) are used for the inverse kinematics. Inverse kinematics is typically hard to solve and a common tactic is to compose the path of the robot as a series of rotations in place and linear movements. This is generally not the optimal choice in terms of time usage and energy efficiency, but the easiest to implement. Equations (3.4) are necessary in the direct kinematics. The encoders on the wheels will measure v_L and v_R and these are transformed into v and ω. Finally, the direct kinematic equations for a differential drive robot can be expressed as

$$\theta(t) = \int_0^t \omega(t')dt',$$

$$x(t) = \int_0^t v(t')\cos(\theta(t'))dt'$$

and

$$y(t) = \int_0^t v(t')\sin(\theta(t'))dt'.$$

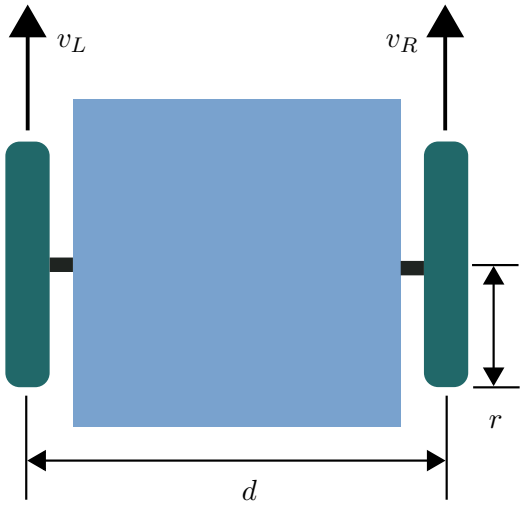


Figure 3.4.: Geometry of a differential drive robot.

3.3. Odometry Motion Model

To predict how the pose of a robot changes, there are two popular approaches. The velocity motion model calculates the change in pose based on the velocity commands given to the motors. This way, one can approximate the pose of the robot forwards in time. However, if the motors are not able to follow the commands, there will be estimation errors. The other method is called the odometry motion model and it uses the information provided by the encoders to calculate the change of pose. Contrary to the first approach, this method cannot predict the pose forwards in time, but it is typically more accurate in its estimation. This is the method that is implemented in ROS.

The encoders provide estimated poses relative to some initial pose. As input to the motion model, the estimated current pose and the estimated last pose is provided,

$$\boldsymbol{u}_t = \begin{bmatrix} \boldsymbol{x'}_{t-1} \\ \boldsymbol{x'}_t \end{bmatrix}.$$

To approximate the robot motion, a decomposition into three simple actions is performed. First, a rotation by δ_{rot1}, then a translation of δ_{trans} in the heading direction of the robot and lastly a rotation by δ_{rot2}, as illustrated in Figure 3.5.

$$\delta_{\text{rot1}} = \text{atan2}(y'_t - y'_{t-1}, x'_t - x'_{t-1}) - \theta'_{t-1}$$

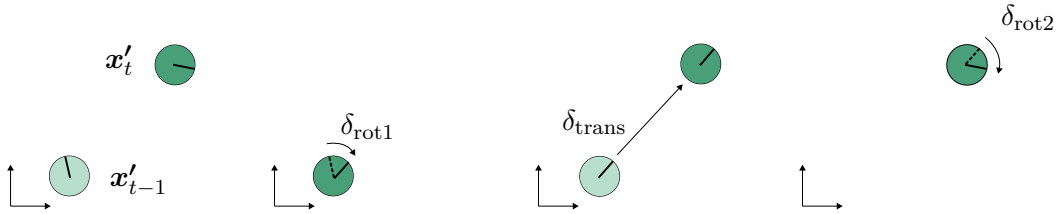


Figure 3.5.: The odometry input followed by the decomposition into three simple actions (rotation by δ_{rot1}, translation by δ_{trans} and rotation by δ_{rot2}).

$$\delta_{\text{trans}} = \sqrt{(x'_t - x'_{t-1})^2 + (y'_t - y'_{t-1})^2}$$
$$\delta_{\text{rot2}} = \theta'_t - \theta'_{t-1} - \delta_{\text{rot1}}$$

The decomposition is used to account for uncertainty. By adding noise to each of the three actions the odometry model is no longer deterministic. Let $\hat{\delta}_{rot1}$, $\hat{\delta}_{trans}$ and $\hat{\delta}_{rot2}$ be the three actions with added noise and let x, y and θ be the components of the state estimate. The updated estimate is calculated with the following formulas.

$$x_t = x_{t-1} + \hat{\delta}_{\text{trans}} \cos(\theta_{t-1} + \hat{\delta}_{\text{rot1}})$$
$$y_t = y_{t-1} + \hat{\delta}_{\text{trans}} \sin(\theta_{t-1} + \hat{\delta}_{\text{rot1}})$$
$$\theta_t = \theta_{t-1} + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$$

The derivation of the formulas can be found in [37].

3.4. LiDAR and Beam Model

"Lidar (light detection and ranging) is an optical remote sensing technology that measures properties of scattered and reflected light to find range and/or other information about a distant target" [20]. The technology has gained a lot of interest lately because of the rapid development of self-driving cars and their extensive use of LiDARs [39]. Compared to the common and cheaper ultrasonic range finder, LiDARs have longer range. However, one should be aware that LiDARs have some drawbacks. Mirrors that are not normal to the laser beam will deflect the laser away from the sensor. Transparent materials will let the laser through. Lastly, matte black materials can absorb parts of the light and make it hard to perform a measurement.

A range scan will never be perfectly accurate. In the particle filter, which will be explained later, we would like to know the probability of a sensor reading given the current state of the robot and the map, $P(z_t|\boldsymbol{x}_t, \boldsymbol{m})$. This conditional probability distribution should be designed such that it accounts for noise in the sensor. Thrun [37] considers four types of noise and these types will be presented in the following. z_t^* denotes the correct distance from the sensor to the obstacle. $z_{\max}$ is the maximum range of the sensor.

The sensor's limited resolution, atmospheric effects and more will cause local measurement noise, Figure 3.6a. This uncertainty is modelled as a Gaussian with a mean of z_t^* and standard deviation σ_{hit},

$$P_{\text{hit}}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) = \begin{cases} \eta\mathcal{N}(z_t; z_t^*; \sigma_{\text{hit}}^2), & \text{if } 0 \leq z_t \leq z_{\max} \\ 0, & \text{otherwise} \end{cases}.$$

The Gaussian distribution in the above equation can be written as

$$\mathcal{N}(z_t; z_t^*; \sigma_{\text{hit}}^2) = \frac{1}{\sqrt{2\pi\sigma_{\text{hit}}^2}} e^{-\frac{1}{2}\frac{(z_t - z_t^*)^2}{\sigma_{\text{hit}}^2}}.$$

η is a normalization coefficient responsible for making $P_{\text{hit}}(z_t|\boldsymbol{x}_t, \boldsymbol{m})$ integrate to one.

Maps used for navigation are typically static. However, the environment might be dynamic. A vacuum cleaner robot can have a static map of the living room, but in reality, chairs and other furniture will be present and move around. This can cause unexpectedly short sensor readings and we will treat this as sensor noise. The noise is modelled by an exponential distribution. The reason for this is that if two or more obstacles are lined up along the sensor beam, only the first one will contribute to the unexpectedly short sensor reading. Figure 3.6b shows how the probability decays with increasing distance. The distribution is given in the following equation where η is once again a normalization term.

$$P_{\text{short}}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) = \begin{cases} \eta\lambda_{\text{short}} e^{-\lambda_{\text{short}} z_t}, & \text{if } 0 \leq z_t \leq z_t^* \\ 0, & \text{otherwise} \end{cases}$$

Sometimes sensors produce readings that cannot be explained by low local measurement noise or unexpected obstacles. For example, a LiDAR might pick up a laser from a different source than itself and regard it as a true sensor reading. This can lead to completely random readings and is modelled as in Figure 3.6c.

Mathematically, we formulate it as an uniform distribution,

$$P_{\text{rand}}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) = \begin{cases} \frac{1}{z_{\max}}, & \text{if } 0 \leq z_t \leq z_{\max} \\ 0, & \text{otherwise} \end{cases}.$$

The last component that should be modelled is the event of a maximum range reading. If the beam is absorbed, reflected or similar, the signal that was sent out will not be returned to the sensor. This will lead to a reading of $z_t = z_{max}$. Figure 3.6d illustrates this point-mass distribution. We draw it as a narrow uniform distribution. The distribution can be written as

$$P_{\max}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) = \begin{cases} 1, & \text{if } z_t = z_{\max} \\ 0, & \text{otherwise} \end{cases}.$$

By combining the four probability distributions presented in Figure 3.6, the distribution shown in Figure 3.7 is obtained. This distribution is a weighted average of the four components,

$$\begin{aligned} P(z_t|\boldsymbol{x}_t, \boldsymbol{m}) = z_{\text{hit}} P_{\text{hit}}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) + z_{\text{short}} P_{\text{short}}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) \\ + z_{\text{rand}} P_{\text{rand}}(z_t|\boldsymbol{x}_t, \boldsymbol{m}) + z_{\max} P_{\max}(z_t|\boldsymbol{x}_t, \boldsymbol{m}), \end{aligned}$$

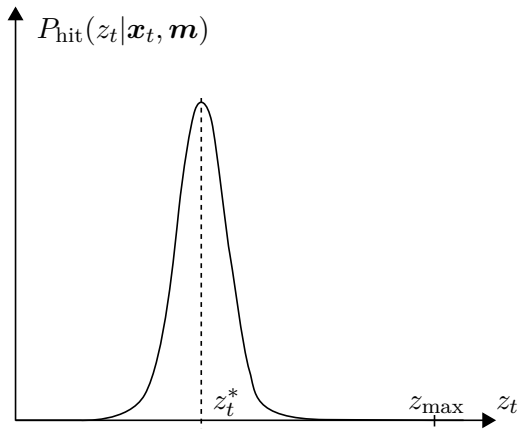
where the zs are the weights. If a single scan at time t consists of several beams, a single beam reading is denoted by z_t^k. Assuming conditional independence between the beam readings, the probability of K beam readings is

$$P(z_t|\boldsymbol{x}_t, \boldsymbol{m}) = \prod_{k=1}^{K} P(z_t^k|\boldsymbol{x}_t, \boldsymbol{m}).$$

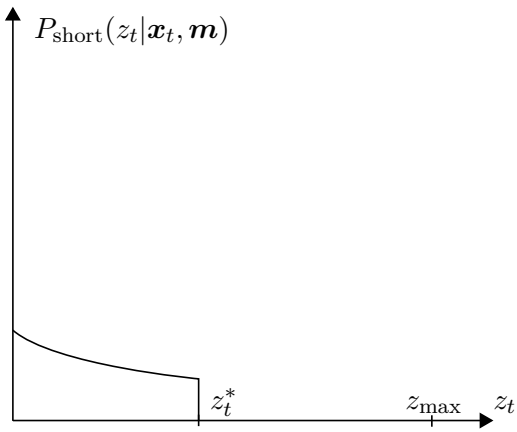
Figure 3.8 displays an example of a reading consisting of K beams.

The weighting parameters z_{hit}, z_{short}, z_{rand} and $z_{\max}$ in addition to σ_{hit} and λ_{short} are the intrinsic parameters of the model. These parameters can be determined by inspecting the data collected from the sensor. By performing measurements where the expected distance, z_t^*, is fixed, the data can be plotted in a histogram and the parameters can be adjusted such that the beam model fits the histogram. Thrun [37] presents an analytical approach where maximum likelihood estimation is applied to estimate the parameters given the observations.

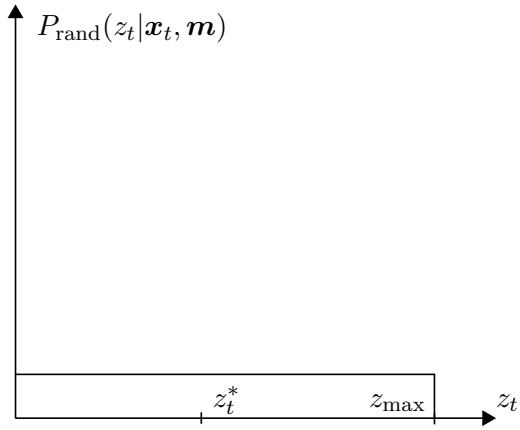
A beam model implementation presents some challenges, mainly related to performance. As seen, the calculation of $P(z_t|\boldsymbol{x}_t, \boldsymbol{m})$ implies calculation of $P(z_t^k|\boldsymbol{x}_t, \boldsymbol{m})$



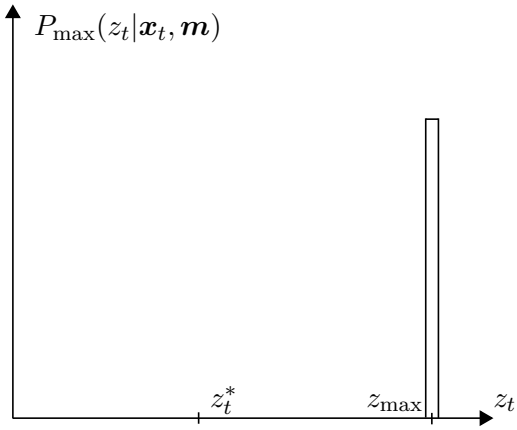
(a) Measurement noise.



(b) Unexpected obstacles.



(c) Random measurement.



(d) Maximum range.

Figure 3.6.: Components of the beam sensor model, [37].

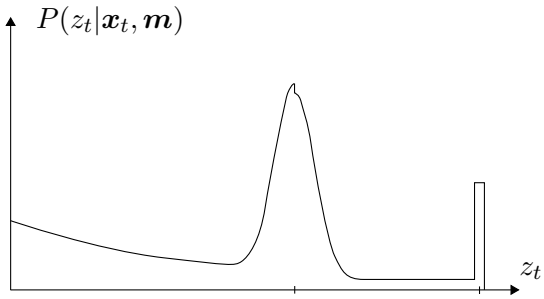


Figure 3.7.: The combined and complete beam sensor model.

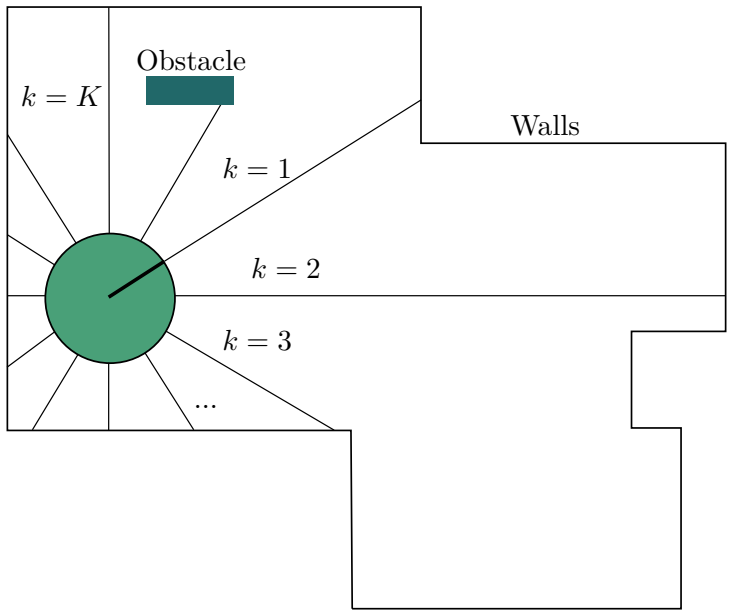


Figure 3.8.: Example of several beams forming a reading at time t.

for K beams. A typical LiDAR will have a K of 360 or greater and several scans are typically performed every second. For every beam, z_t^{k*}, the expected reading must be found by virtually tracing the beam from the robot until it hits an edge of the map. To reduce the computation, one can reduce the number of beams. This will linearly decrease the computational complexity. Another effort to increase the performance is to discretize the state-space and pre-compute the expected measurements for the states. There will here be a trade-off between memory usage and resolution of the grid. Lastly, it is worth noting that the beam model suffers from a lack of smoothness. Imagine that a beam is obstructed by a thin pillar. If the robot changes orientation just a little, the beam is no longer obstructed and a completely different reading is obtained. This can lead to convergence problems and is a major drawback of the method.

3.5. Localization with Particle Filter

Localization is defined by Thrun [37] with "the process of establishing correspondence between the map coordinate system and the robot's local coordinate system". Given a map of the environment, knowledge of own movement and sensor

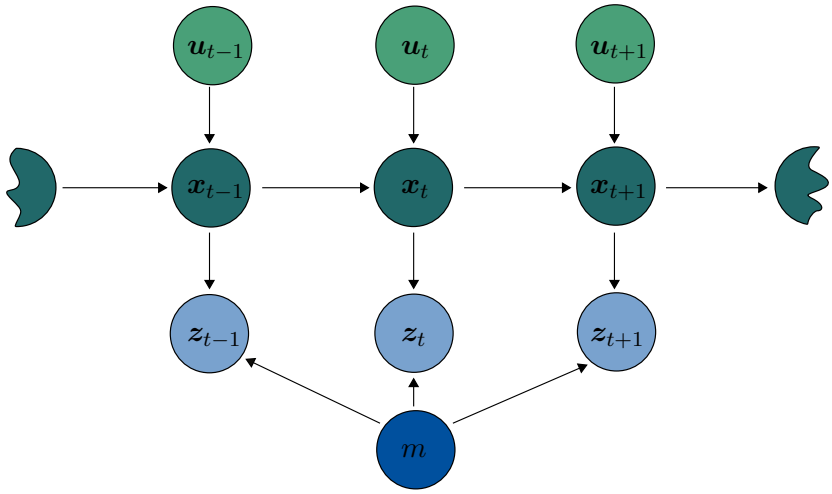


Figure 3.9.: Illustration of how the states $\boldsymbol{x}$, measurements $\boldsymbol{z}$, actions $\boldsymbol{u}$ and map m depend on each other in a localization problem, [37].

inputs, the robot should be able to determine its pose,

$$\boldsymbol{x}_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix},$$

at all times t. To localize the robot accurately, sensor data has to be gathered over time. Two very different poses can produce similar sensor readings and to determine which of the poses that are correct, the robot must move and perform new measurements to see if these new measurements match the expectation given the two old poses and the movement. The process is illustrated in Figure 3.9. The pose depends on earlier poses and actions. The belief in a pose is determined by the belief in the last pose and is updated by a measurement.

The ideas behind Bayes filtering are important in several of the solutions to the localization problem and will be presented briefly. To denote different discrete times, the subscript k will be used. The following equation describes the evolution of the robot's state over time,

$$\boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, m, \boldsymbol{w}_k). \tag{3.6}$$

The process noise $\boldsymbol{w}_k$ is assumed to be a white noise sequence with known probability density function, pdf [33]. In ROS, this system equation is implemented with the odometry motion model as discussed in section 3.3. Next, the measurements

relation to the map and state is expressed with,

$$\boldsymbol{z}_k = h_k(\boldsymbol{x}_k, m, \boldsymbol{v}_k), \tag{3.7}$$

the measurement equation. Like $\boldsymbol{w}_k$, the measurement noise $\boldsymbol{v}_k$ is assumed to be white noise. Equation (3.6) will let us calculate $P(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k, \boldsymbol{u}_k, m)$, how the state of the robot changes over time given an action. Likewise, equation (3.7) will let us calculate $P(\boldsymbol{z}_k|\boldsymbol{x}_k, m)$, how likely we are to observe a measurement given the state of the robot. With a LiDAR, this is done using the already discussed beam model (3.4). Note how these conditional probabilities correspond with the dependencies shown in Figure 3.9.

As mentioned, to localize the robot, a series of state transitions and measurement updates must be performed. The a priori distribution is found with

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k+1}, m) = \int P(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k, \boldsymbol{u}_{k+1}, m)P(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k}, m)d\boldsymbol{x}_k, \tag{3.8}$$

where $1:k$ denotes a set of k sequential variables and $P(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k, \boldsymbol{u}_{k+1}, m)$ is given by the system equation (3.6). A priori means that it is the belief in the state before a measurement update of this state is conducted. An intuitive explanation of this is that the probability of transitioning into a new state is the probability of being in a prior state times the probability of a transition from this prior state to the new one, given the chosen action, summed up for all possible prior states. When a new measurement is obtained, the posterior distribution is calculated,

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k+1}, m, \boldsymbol{z}_{k+1}) = \frac{P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}, m)P(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k+1}, m)}{P(\boldsymbol{z}_{k+1}|\boldsymbol{z}_{1:k})}. \tag{3.9}$$

The factor $P(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k+1}, m)$ is the a priori distribution, equation (3.8). $P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}, m)$ is available from the measurement equation (3.7). The denominator is a normalization term and is found as an integral of the numerator for all state vectors,

$$P(\boldsymbol{z}_{k+1}|\boldsymbol{z}_{1:k}) = \int P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}, m)P(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k+1}, m)d\boldsymbol{x}_k. \tag{3.10}$$

This completes what is needed from Bayes filtering. Equation (3.8) is applied when an action is performed and (3.9) updates the belief using a newly arrived measurement. A full derivation of the Bayes filter can be found in the appendix, section B.5.

Several methods exist for solving the localization problem. Applying the Bayes filtering directly on the problem is called Markov localization. This involves dis-

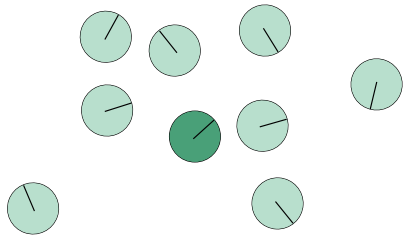


Figure 3.10.: High uncertainty in the pose estimation.

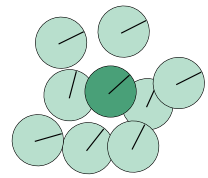


Figure 3.11.: Low uncertainty in the pose estimation.

cretizing the state space. Unfortunately, for large maps and a sufficiently fine discretization, it becomes unfeasible to update all the states as fast as needed. Another solution is to use the Extended Kalman Filter (EKF), a modification of the Kalman Filter that can be used also on non-linear problems. Similarly, the Unscented Kalman Filter can be applied. However, a particle filter is what is being used for localization in ROS. Know for being computational expensive, particle filters have traditionally not been suitable for real-time applications. Fortunately, they are highly parallelizable and with modern day GPUs packing well over 1000 cores [24], real-time applications are feasible.

A particle filter implementation of the localization problem is often called Monte Carlo localization in the literature. The algorithm uses a set of particles to estimate the correct pose of the robot. Here, a pose will consist of the three variables x, y and θ. Every particle will hold a value for each of the three variables. A particle can be seen as a hypothesis of what is the correct pose of the robot. Our best guess of the correct pose will then be the mean of all the particles. The uncertainty of the estimate is reflected in how the particles are spread out in the state-space. When the uncertainty is high, the particles will hold very different values. Typically, the uncertainty is high when the algorithm is initialized. Over time, as sensor readings are gathered, the particles will start to condense and the uncertainty drops. If the particles condense around the correct pose, we have convergence. In Figure 3.10 and Figure 3.11 eight particles (light green) are illustrated together with the correct pose (dark green). Even though the mean of the particles in Figure 3.10 might be close to the correct pose, Figure 3.11 illustrates a superior situation as the uncertainty is lower.

Monte Carlo localization represents the belief, found in the Bayes filtering, as particles. The belief, $bel(\boldsymbol{x}_t)$, is simply an abbreviation of the posterior distribution. When initializing the particle filter, the particles are sampled randomly

within the borders of the map. Every particle will have a corresponding weight and these are set to $w^i = \frac{1}{N}$ for all N particles, $i \in [1, N]$. Now the iterations of the particle filter can begin. First, for all particles, the state is updated with the system equation (3.6). Then, the weights are updated with the measurement equation (3.7). So far, all the particles have been treated independently. Now the particles are resampled. Every particle is resampled as a clone of particle i by a probability of w^i. This will make the particles condense around the poses which best conform to the measurements. Lastly, the weighted mean is calculated $\sum_{i=1}^{N} w_k^i \boldsymbol{x}_k^i$, our best guess at the correct pose.

The implementation that has been outlined here is the simplest possible. Several modifications are typically implemented to make the filtering more robust and less computational expensive. The resampling step is a performance bottleneck of the particle filter as it cannot be easily parallelized. Also, it causes what is called sample impoverishment [33] because particles become exact copies of each other. Several particles with identical states have no more estimation power than a single particle. Therefore, we want to perform the resampling only when needed. Instead of resampling at every iteration, weights are updated by multiplying the old weight with the new sensor update before a normalization is performed. To determine for which iterations resampling should be performed, the effective number of particles,

$$\hat{N}_{\text{eff}} = \frac{1}{\sum_i (w_k^i)^2}, \tag{3.11}$$

can be used. w_k^i is the weight of particle i at time step k. We can set a threshold, N_{th} such that if $\hat{N}_{\text{eff}} < N_{\text{th}}$, we should perform resampling. This ensures that resampling is performed only when needed. Additionally, a method called roughening [16] is used to avoid sample impoverishment. Roughening is about adding artificial process noise in order to spread out the particles in the state space. For every iteration, a set of particles is chosen randomly and noise is added to modify the states. Keeping the particles a little spread increases the robustness because even after sudden jumps in the true pose, there will likely be a particle close by.

The Navigation stack in ROS implements an adaptive variant of Monte Carlo localization [14]. For every particle in every iteration, there is either a random sampling of the pose or a resampling. The algorithm keeps track of the average weight in the long run and the short run. Low weights indicate that the particles are unlikely to reflect the true pose. If the short run average weight is low compared to the long run one, this is a sign that there might be divergence and an injection of random particles can aid the filter back to convergence. In what is known as the kidnapped robot problem, the robot is picked up, transported to another part of the map and put down. An injection of random particles is here

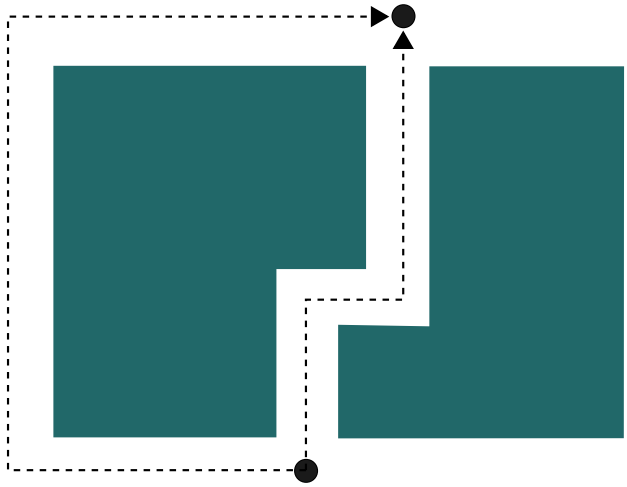


Figure 3.12.: Two routes were one is shorter, but also more narrow.

necessary to spread out the particles and regain convergence.

Contrary to localization methods that use Gaussian techniques, Monte Carlo localization can handle raw sensor measurements. With injection of random particles it solves the kidnapped robot problem. Additionally, it has a great potential for parallel implementations. Together, these benefits make Monte Carlo localization a popular choice for localization.

3.6. Global Path Planning

In an ideal world, the robot would be able to follow a planed route perfectly. If it was requested to move one meter straight forward, it would move exactly one meter straight forward. However, no robot is able to follow execute the requested actions perfectly, their actions are stochastic, not deterministic. Figure 3.12 shows a typical example of planning in the real world versus in a deterministic world. The straight on alternative is definitely shorter, but it is at the same time more narrow. For a robot with uncertainty in the movement, the longer, but simpler route might be faster depending on the degree of uncertainty. Trying to navigate the narrow path with a low quality robot can be more time consuming because of collisions.

A common strategy in path planning is to discretize the state-space down to a set of equally sized squares. Every state, $\boldsymbol{x}$, can then be determined by a x- and a

y-coordinate. The goal of the planning is then to generate a policy, π, that states what is the best action $\boldsymbol{u}$ for every state,

$$\pi : \boldsymbol{x} \longrightarrow \boldsymbol{u}. \tag{3.12}$$

Here, a Markov decision process is assumed. This requires the measurement model $P(\boldsymbol{z}|\boldsymbol{x})$ to be deterministic. In this setting this means that the position of the robot is known. If the localization is relatively precise and the grid discretization is coarse, this can be a reasonable assumption. Also, a static world is assumed as the policy is time-independent.

To express which actions that are preferred when in a certain state, a payoff function, r, is used. If performing action $\boldsymbol{u}_i$ in state $\boldsymbol{x}_j$ causes the robot to move to the goal state, this state-action pair should have a high payoff. Thrun [37] proposes a payoff function,

$$r(\boldsymbol{x}, \boldsymbol{u}) = \begin{cases} 100, & \boldsymbol{u} \text{ leads to the goal} \\ -1, & \text{otherwise.} \end{cases} \tag{3.13}$$

By punishing the robot with -1 if it does not reach the goal, the robot will have an incentive to move towards the goal as fast as possible.

One way of determining the optimal policy is through value iteration. "Every policy has an associated value function, which measures the expected value (cumulative discounted future payoff) of this specific policy" [37]. The calculation of the value function is an iterative process and is conducted with the following update.

$$\hat{V}(\boldsymbol{x}_i) \longleftarrow \gamma \max_{\boldsymbol{u}} \left(r(\boldsymbol{x}_i, \boldsymbol{u}) + \sum_{j=1}^{N} \hat{V}(\boldsymbol{x}_j) P(\boldsymbol{x}_j | \boldsymbol{u}, \boldsymbol{x}_i) \right) \tag{3.14}$$

N is the number of states and γ is a fixed discount factor. $r(\boldsymbol{x}_i, \boldsymbol{u})$ is the expected immediate payoff of performing action $\boldsymbol{u}$ in state $\boldsymbol{x}_i$. $\hat{V}(\boldsymbol{x}_j)P(\boldsymbol{x}_j|\boldsymbol{u}, \boldsymbol{x}_i)$ is the value of state $\boldsymbol{x}_j$ multiplied by the probability of a transition to this state when performing action $\boldsymbol{u}$ in state $\boldsymbol{x}_i$. Summing this up for all states reveals the expected future payoff. The updated value function is then the expected immediate and future payoff resulting from the best action multiplied by the discount factor, γ. This is a value between 0 and 1. Setting a low γ value results in more shortsighted decisions where immediate payoffs are more important than future payoffs compared to a high γ. Initially, $\hat{V}(\boldsymbol{x}_i)$ is set to some small value for every state. An iteration is to perform the update (3.14) for all states once. Several iterations will make the estimated value function, $\hat{V}(\boldsymbol{x}_i)$, converge towards the true value function, $V(\boldsymbol{x}_i)$. When converged, the best action in a state i can be

found by

$$\arg\max_{\boldsymbol{u}} \Big(r(\boldsymbol{x}_i, \boldsymbol{u}) + \sum_{j=1}^{N} \hat{V}(\boldsymbol{x}_j) P(\boldsymbol{x}_j | \boldsymbol{u}, \boldsymbol{x}_i) \Big). \tag{3.15}$$

3.7. Local Path Planning

The global path planner will not be able to account for all possible situations in a probabilistic environment. Therefore, local path planning is necessary for a robot to react to unexpected obstacles and to re-plan the route. Possible objects can be humans, chairs that have been moved and parts that have fallen over. A well-established method for local path planning is the Dynamic Window Approach (DWA) [13].

The Dynamic Window Approach can be summarized in four steps [21]:

1. Discretely sample the robot's control space (v, ω).

2. For each sample, perform a simulation from the current state for a fixed duration forwards in time to predict what would happen if the control variables in this sample were chosen.

3. For each simulation, score the trajectory. The score can be based on multiple factors like speed, proximity to global path and clearance from obstacles. Samples that result in simulations with collisions are removed.

4. Pick the (v, ω) sample that results in the highest scoring trajectory and execute.

5. Repeat the process.

Because the samples contain both a linear and an angular velocity, the trajectories will be circular and the search space is two-dimensional. Further, DWA takes the robot's dynamic properties into consideration. Only linear and angular velocities that are reachable within the short simulation time are sampled, $v_n \in [v_{n-1} - \Delta v, v_{n-1} + \Delta v]$ (similar for ω). This restricts the search space and makes sure that samples implying infeasible accelerations are avoided.

Which trajectory that is chosen is determined by the objective function of step 3.

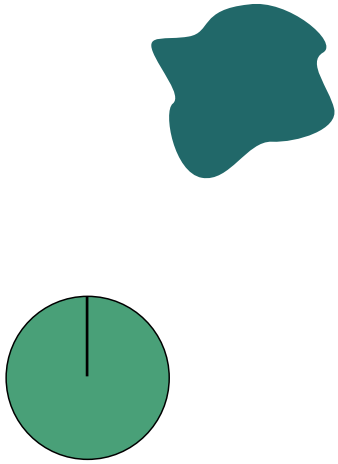


Figure 3.13.: A robot and an obstacle.

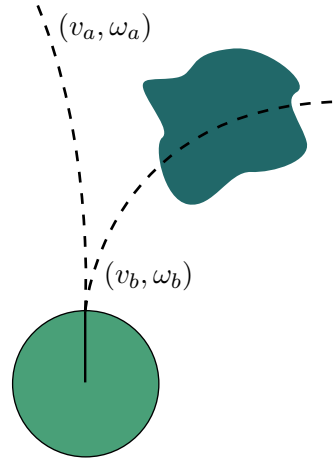


Figure 3.14.: Illustration of the trajectories resulting from the velocities at P_a and P_b of Figure 3.15.

The function is defined as,

$$G(v, \omega) = \sigma(\alpha \cdot \text{heading}(v, \omega) + \beta \cdot \text{dist}(v, \omega) + \gamma \cdot \text{vel}(v, \omega)), \tag{3.16}$$

and should be maximized to find the optimal trajectory. "heading" favours the trajectories that progress the most toward the goal location. "dist" is simply the distance to the closest obstacle along the trajectory. Trajectories that keep a good distance to obstacles are less likely to cause collisions and receive high scores. "vel" is the linear velocity of the robot. High velocities will let the robot arrive at the goal location sooner and are hence favoured. α, β and γ control how important the different features are relative to each other. Exemplifying, if collisions must be avoided under all circumstances, one would pick a high β-value relative to α and γ. In general, how the three constants are set relative to each other will vary from environment to environment. Typically, the constants are set through experimentation. Lastly, σ is a function responsible for smoothing out the contribution from each of the three.

Figure 3.15 shows how the velocity state space might look. The white region, V_r, is the set of (v, ω) that will be scored by the objective function. This is inside the dynamic window, V_d, but not within the dark area which is the area that will cause collision with the obstacle. The width and height of the dynamic window is controlled by the robot's ability to accelerate and it is centered around the current velocity. V_s is the area that covers all the possible velocities that can be reached

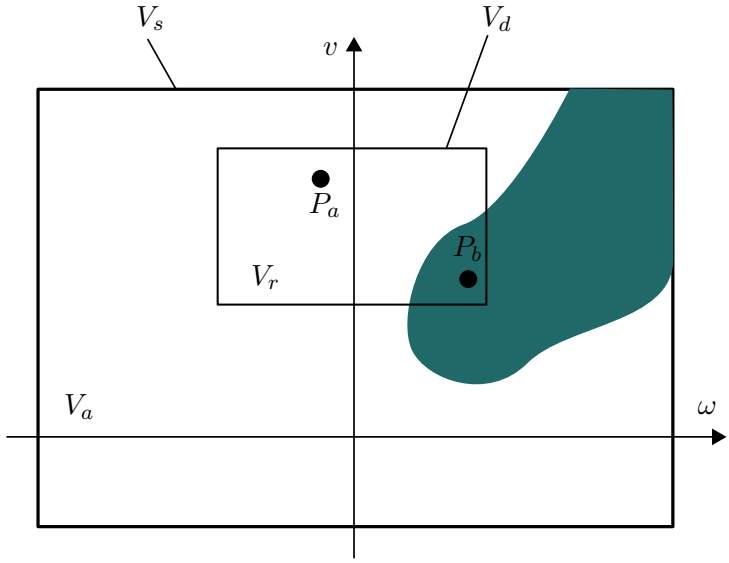


Figure 3.15.: Illustration of Figure 3.13 in the (v, ω) velocity space.

by the robot given enough time to accelerate. V_a is the area inside V_s that does not cause collisions. Summed up, $V_r = V_s \cap V_a \cap V_d$. As an example, the trajectories resulting from $P_a = (v_a, \omega_a)$ and $P_b = (v_b, \omega_b)$ are shown in Figure 3.14.

3.8. Bresenham's Line Algorithm

An algorithm is needed to simulate the LiDAR scan and find the expected distance readings based on the map. From a starting pixel, the algorithm should move in a direction dictated by an angle, from pixel to pixel, until either the current pixel is marked as occupied on the map or the end of the map is reached. Bresenham's algorithm [19] can be used for this purpose.

Let θ be the angle between the x axis and the heading direction of the line. The slope of this line can be found as $s = \tan(\theta)$. (x_0, y_0) is the starting position of the line. m is the map with a value of "true" at the pixels where there are walls or other obstacles. w and h are the width and the height of the map. The following pseudocode explains the algorithm.

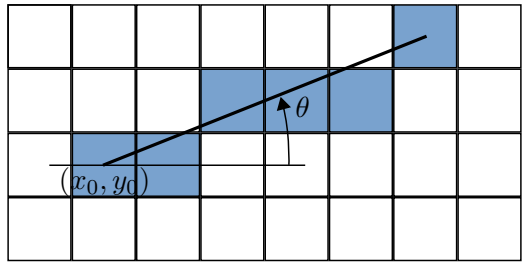


Figure 3.16.: The Bresenham line algorithm.

error ← 0.0
$y \leftarrow y_0$
$x \leftarrow x_0$
while $x < w$ **and** $y < h$ **and not** $m[x][y]$ **do**
  $x \leftarrow x + 1$
  error ← error + abs(s)
  if error ≥ 0.5 **then**
    $y \leftarrow y + 1$
    error ← error − 1
  end if
end while

The greater the slope, the faster the error increases and the more often we will increment y. Figure 3.16 illustrates the algorithm. Note that the algorithm discussed here only holds for the first octant. For the second octant, x and y will change roles such that y is incremented for every iteration. For the third octant, y is increased every iteration and x is decreased when the error exceeds 0.5. All of the octants will have their own variants.

3.9. Occupancy Grid Mapping

Occupancy grid mapping is a technique that is used to build probabilistic maps of the environment. For a robot operating in the plane, this map will consist of square cells. Every cell holds a value that expresses how probable it is that the corresponding space in the environment is occupied. The goal of the mapping is to update the cells with the information gained from range scans.

A range scan is characterized by a start point, an angle and a distance. From these, the end point can be calculated. Then, a line is drawn between the start point and the end point. All the grid cells that are crossed by the line are updated

by this particular scan. For these cells, the associated occupancy probability will decrease, except for the last cell. The cell that encircles the end point will see an increase of occupancy probability. Cells that are not crossed by the line are not updated by this scan.

Let $m_{x,y} = 1$ denote that the grid cell at position (x, y) is occupied. $m_{x,y} = 0$ is then that this cell is not occupied. For a range scan, all the cells crossed by the scan will get an associated binary value $z_{x,y}$. The value is 1 at the end point and 0 elsewhere. These are the values that will update the probability of a cell being occupied, $P(m_{x,y} = 1)$. The posterior probability is the updated probability, after the scan is performed, $P(m_{x,y} = 1|z_{x,y})$. Instead of trying to find an update formula for this, odds are typically used. The odds of an event A is,

$$\text{odds}(A) = \frac{P(A)}{1 - P(A)},$$

the probability of the event happening divided by the probability that it is not happening. In grid mapping, this becomes,

$$\text{odds}(m_{x,y} = 1|z_{x,y}) = \frac{P(m_{x,y} = 1|z_{x,y})}{P(m_{x,y} = 0|z_{x,y})}. \tag{3.17}$$

We are here assuming that the pose of the robot is known. Bayes formula is used to rewrite both the numerator and the enumerator,

$$P(m_{x,y} = 1|z_{x,y}) = \frac{P(z_{x,y}|m_{x,y} = 1)P(m_{x,y} = 1)}{P(z_{x,y})},$$

$$P(m_{x,y} = 0|z_{x,y}) = \frac{P(z_{x,y}|m_{x,y} = 0)P(m_{x,y} = 0)}{P(z_{x,y})}.$$

Inserting into (3.17) gives

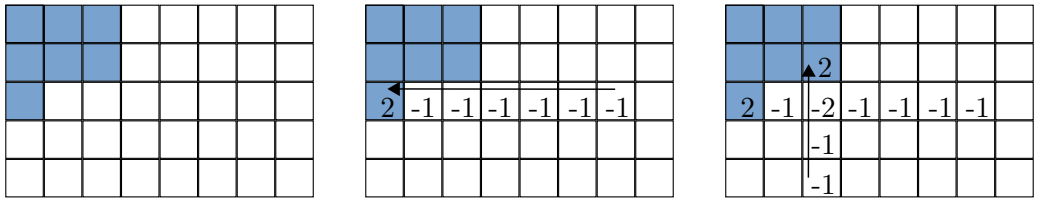
$$\text{odds}(m_{x,y} = 1|z_{x,y}) = \frac{P(m_{x,y} = 1|z_{x,y})}{P(m_{x,y} = 0|z_{x,y})} = \frac{P(z_{x,y}|m_{x,y} = 1)P(m_{x,y} = 1)}{P(z_{x,y}|m_{x,y} = 0)P(m_{x,y} = 0)}.$$

To simplify the update of the odds, the logarithm is applied,

$$\log\Big(\frac{P(m_{x,y} = 1|z_{x,y})}{P(m_{x,y} = 0|z_{x,y})}\Big) = \log\Big(\frac{P(z_{x,y}|m_{x,y} = 1)P(m_{x,y} = 1)}{P(z_{x,y}|m_{x,y} = 0)P(m_{x,y} = 0)}\Big).$$

The logarithm lets us decouple the expression into

$$\log\Big(\frac{P(m_{x,y} = 1|z_{x,y})}{P(m_{x,y} = 0|z_{x,y})}\Big) = \log\Big(\frac{P(z_{x,y}|m_{x,y} = 1)}{P(z_{x,y}|m_{x,y} = 0)}\Big) + \log\Big(\frac{P(m_{x,y} = 1)}{P(m_{x,y} = 0)}\Big). \tag{3.18}$$



(a) Initial (all zeros). **(b)** First update. **(c)** Second update.

Figure 3.17.: Two updates of an occupancy grid map.

The last term of (3.18) is the a priori, the odds before the update. This is added with the current evidence to form the posterior odds.

Initially, it is seen as equally likely for a cell to be occupied or not. This gives $P(m_{x,y} = 1) = P(m_{x,y} = 0) = 0.5$, an odds of 1 and a log-odds of 0. Like mentioned, $z_{x,y}$ can take on a value of 0 or 1. To perform the occupancy grid mapping, $P(z_{x,y} = 1|m_{x,y} = 1)$ and $P(z_{x,y} = 1|m_{x,y} = 0)$ must be determined from experience. $P(z_{x,y} = 0|m_{x,y} = 1)$ and $P(z_{x,y} = 0|m_{x,y} = 0)$ are simply calculated as the complements of these two probabilities. With these set, the updated occupancy grid map is calculated recursively from (3.18) every time a range scan is obtained. Figure 3.17 shows an example of how a grid map is updated. Initially, nothing is known about the map and all log-odds are zero. When a scan is performed it is traced on the grid and the values along the ray are decremented except at the end where there is an increment. As the next scan is performed, the values are updated in the same way. Over time, the values should become low in cells where there are no obstacles and high in cells where there are obstacles.

3.10. Morphological Image Processing

In morphological image processing, a binary image is interpreted as a set of foreground pixels [15]. Let A and B be two binary images. $C = A \cup B$ will then be the binary image that has ones (1-pixels) at the positions where either A or B has a one and zeros (0-pixels) elsewhere. Written with set theory, this becomes

$$C = \{(x, y)|(x, y) \in A \text{ or } (x, y) \in B \text{ or } (x, y) \in (A \text{ and } B)\}.$$

This representation in form of sets differs from the more traditional intensity $= f(x, y)$ in that it stores the position of the foreground pixels and everything else is considered as background.

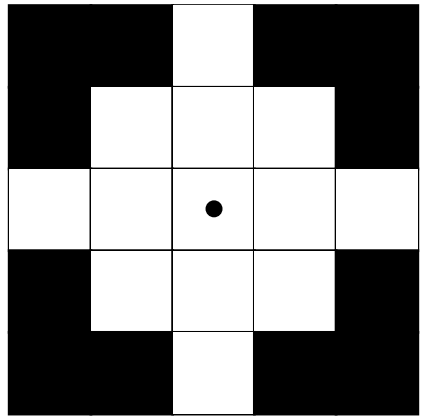


Figure 3.18.: A typical 5 by 5 circular structuring element.

Morphological image processing consists of more than only unions, intersections, complements and differences. A more complex operation known as reflection is an essential part of morphological image processing. Let B be a set, the reflection of B is defined as

$$\hat{B} = \{w | w = -b \text{ for } b \in B\}.$$

The operation reflects the set about both the horizontal and vertical axes. Exemplifying, the point $(5, -2)$ would become $(-5, 2)$. Another fundamental operation is translation. Let $z = (z_1, z_2)$, the translation of set A by z is then

$$(A)_z = \{c | c = a + z \text{ for } a \in A\}.$$

All pixels in A are moved z_1 pixels to the right and z_2 pixels downwards.

Several morphological methods utilize a structuring element. Structuring elements are typically represented as matrices consisting of zeros (background) and ones (foreground) with a defined center point. Figure 3.18 shows a circular structuring element with the defined center in the middle.

Dilation is an operation that is used to thicken structures in an image. It is achieved by translating a structuring element around in a binary image. When a foreground pixel in the structuring element overlaps with a foreground pixel in the image, the operation will leave a foreground pixel at the position of the center of the structuring element. Therefore, which pixel of the structuring element that is defined as the center and the shape of the structuring element will all affect the resulting image. The larger the element, the more the structures will grow. Mathematically, the dilation of an image A by a structuring element B is defined

as

$$A \oplus B = \{z|(\hat{B})_z \cap A \neq \emptyset\},$$

where $\emptyset$ is the empty set.

Erosion can be thought of as the opposite of dilation. Instead of thickening structures, it performs thinning. Only when all pixels of the structuring element overlap with the image at the same time, a foreground pixel in the resulting image is produced. The operation is formalized as

$$A \ominus B = \{z|(B)_z \cap \bar{A} = \emptyset\},$$

where $\bar{A}$ is the complement of A.

More complex operations can be achieved by combining dilation and erosion. Opening is an erosion followed by a dilation with the same structuring element,

$$A \circ B = (A \ominus B) \oplus B.$$

The erosion will thin the structures in the image and already thin structures might disappear altogether depending on the size of the structuring element. The dilation then thickens the structures back to original thickness. However, the structures that disappeared during the erosion are not recovered. Therefore, an opening operation can be used to remove thin or small structures like noise grains. When the order of the erosion and the dilation is switched, it is called a morphological closing,

$$A \bullet B = (A \oplus B) \ominus B.$$

The dilation can cause close-by structures to grow together and the following erosion will not separate the structures. Thus, a closing can be used to grow structures together. This is convenient when we want to connect broken lines or fill the interior of structures.

3.11. Contours in Binary Images

Finding the contours of a binary image is useful for labeling groups of pixels in a binary image. If we know that the image consists of multiple objects, the pixels within each contour can be assumed to belong to the same object. Figure 3.19 shows a binary image where there are drawn blue contours around groups of 1-pixels. Typically, every contour will surround unique objects and the number of objects in the image will therefore be equal to the number of contours.

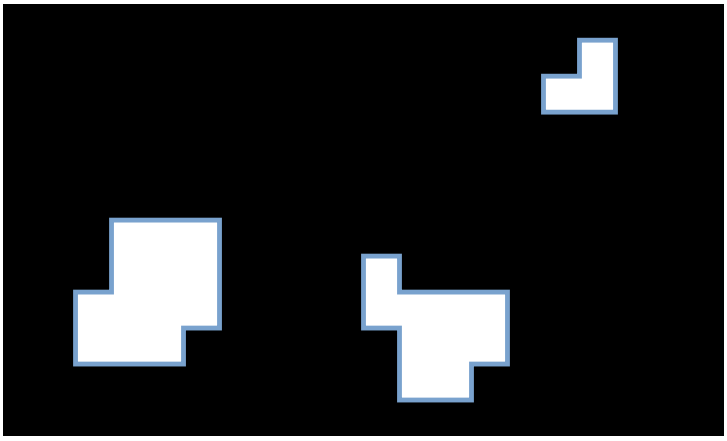


Figure 3.19.: The contours of three structures marked in blue.

The task of finding contours in binary images is a classical image processing problem and several methods exist. In the popular image processing library OpenCV, contours are calculated with a method developed by Suzuki and Abe [35].

3.12. Image Moments

An image moment is a weighted average of the pixel intensities of an image. Image moments are typically calculated for binary images, but can also be calculated for grayscale images. Here, we let $I(x, y)$ be the pixel intensity of the binary pixel at the Cartesian position (x, y). This value will either be zero or one. Then, an image moment is defined as,

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y). \tag{3.19}$$

The simplest possible image moment is for $i = j = 0$,

$$M_{00} = \sum_x \sum_y I(x, y).$$

As can be seen from the equation this is simply the number of pixels with a binary value of one.

Image moments can be used for the calculation of the centroid of an image. Let the centroid be denoted by $(\bar{x}, \bar{y})$. Utilizing image moments, the equations simply

become

$$\bar{x} = \frac{M_{10}}{M_{00}} \qquad \bar{y} = \frac{M_{01}}{M_{00}}. \tag{3.20}$$

3.13. Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is a method for mapping the environment when the localization of the robot is unknown. In this Master's thesis the map can be extracted from CAD-models and SLAM is not needed, but the method is paramount in many mobile robot applications and will therefore be outlined here. Details can be found in *Probabilistic Robotics* [37] by Thrun.

Compared to the localization problem of estimating $P(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k}, m)$, SLAM is harder. Now the map is no longer known and both the pose and the map must be estimated based on measurements and actions, $P(\boldsymbol{x}_k, m|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})$. This problem is known as online SLAM. In full SLAM one tries to estimate the entire history of poses, not only the current pose, $P(\boldsymbol{x}_{k:1}, m|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})$.

SLAM can produce both feature-based maps and grid maps. Feature-based maps are based on the detection of landmarks. A landmark can be an object with a specific colour or something else that the robot is able to detect reliably. The challenge is then to estimate the positions of these landmarks relative to the robot. If we operate in the plane where the robot's pose is described by x_r, y_r and θ_r, every landmark i need to be described by $x_{l,i}$ and $y_{l,i}$. Therefore, for N landmarks we need to estimate $3+2N$ variables and it becomes obvious why SLAM is harder than localization. While feature-based mapping is typically dependent on cameras and image processing, grid maps are suitable for robots with LiDARs. Then the problem becomes to estimate the robot's pose and the probability of each grid cell being occupied. Several algorithms exist for solving both SLAM with feature-based maps and grid maps. In ROS, SLAM using grid maps is implemented with a method called GMapping [17]. The method is based on a particle filter where every particle holds an estimate of the robot's pose and the entire grid map.

Chapter 4.

TurtleBot and ROS

ROS can seem a little overwhelming at first and this chapter tries to provide the reader with some basic knowledge about the inner workings of an application developed with ROS. Some of the pre-existing ROS packages that are paramount for the new functionality developed in this project are mentioned together with a list of the new functions. This servers as a short introduction for what is to come in the following chapters. Before this, the robot used in the project, the TurtleBot3, is presented.

4.1. TurtleBot

The TurtleBot3 Waffle Pi, Figure 4.1, was chosen as the mobile robot to be used in this project. It is an open source robot that is used in education, research and product development. Among the most important features are an affordable price, a small form factor and possibilities to add more sensors. Nonetheless, the most important feature is the out of the box integration with ROS. Installation of the software on the on-board Raspberry Pi is well documented [29] and it allows the user to quickly begin the development. A thorough walk-through of the already existing TurtleBot applications, including SLAM mapping, is also available in the manual. With encoders on the wheels, a LiDAR and a RGB camera, the robot has all the most important sensors for mapping and navigating. More details about the robot's hardware can be found in the appendix, Table C.1.

Several alternatives to the TurtleBot exist. ROSbot 2.0 by Husarion is similar to TurtleBot in size, but is more solidly built. It also has a stereo camera setup that allows it to assign a depth value to every pixel. Clearpath Robotics delivers a

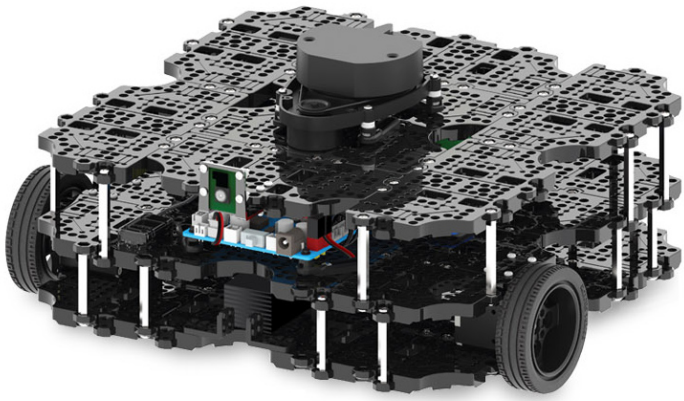


Figure 4.1.: TurtleBot3 Waffle Pi [27].

variety of UGVs for outdoors use that are larger in size than the TurtleBot. Like the other robots mentioned, these robots have API support for ROS. An even more advanced option is the KMR iiwa from KUKA. However, this robot does not have the same support for ROS.

As mentioned, a TurtleBot was the chosen robot. More precise and robust alternatives exist, but an AGV operating on an O&G platform would most likely have to be custom made. This is due to strict regulations that aim to prevent explosions and the extreme weather conditions faced offshore. The TurtleBot was considered the best for a proof of concept where ease of development is more important than robustness.

Many projects involving the TurtleBot can be found online. A good example of its many uses is the AutoRace RBIZ Challenge [28]. In the challenge, teams compete to navigate through an obstacle course as fast as possible. To succeed, the robot must be able to detect traffic light signals, follow lane lines, perform parking, detect traffic signs, move through a dark tunnel and more. This requires the use of image processing, artificial intelligence and processing of LiDAR data. In other words, the use-cases of the robot are many.

4.2. ROS Concepts

The Robot Operating System (ROS) is a middleware for programming robots. It runs on Linux, but provides many of the same features as a standard operating

system [3]. Most importantly it allows for easy communication between processes and threads. Exemplifying, a process controlling the camera can send a message containing image information to a process that does further image processing before this process sends a message to a process controlling the motors, and so on. This exchange of information through messages can be performed even with processes running on different computers. ROS provides a good overview of how the different processes communicate, making it easier to debug applications. The software is designed to be modular and packages developed by other users can easily be implemented in your own application. In the following, the main ROS concepts necessary to get started with ROS will be explained briefly.

Nodes are the processes. A process can be as simple as adding two numbers. The most important is that the node is designed to have a single and limited scope. This way, the nodes can more easily be re-used in other projects. In the world of self driving cars, a node can be responsible for finding the lane lines in an image. The node will receive the image from another node, perform image processing operations like Canny Edge Detection and Hough Lines, and pass the mathematical description of the lines to yet another node. The code run by a node can be written in C++, Python and several other programming languages.

Messages are what the nodes use to communicate between each other. These must be well-defined, meaning that it is not possible for a node to send an arbitrary combination of floats, strings and integers to another node. If a node should distribute an image, it must use a pre-defined image message structure, specify the width and height of the image, the pixel values and possibly more. The messages help standardizing the interface between nodes, once again making the applications more modular and easier to reuse.

Topics are what the messages are being sent over. A node can publish a message on a topic and other nodes can subscribe to this topic. Publishing nodes will not know which nodes that receive their messages and subscribing nodes will not know which nodes sent the messages. However, nodes are not limited to publish only on a single topic. A node can subscribe to several topics and publish messages to several topics. Only messages of the pre-defined type can be distributed on a topic. A single integer cannot be published on the same topic as an image.

Services are an alternative to the already described publisher-subscriber method. Instead of publishing a message on a topic, a node can request another node to perform a service. This can be as simple as a node sending a list of numbers to a node, requesting it to add the numbers and returning the sum. Like messages, services must be well-defined. In the example, a service must exist that specifies that the request is a list of numbers and that the reply is a single number.

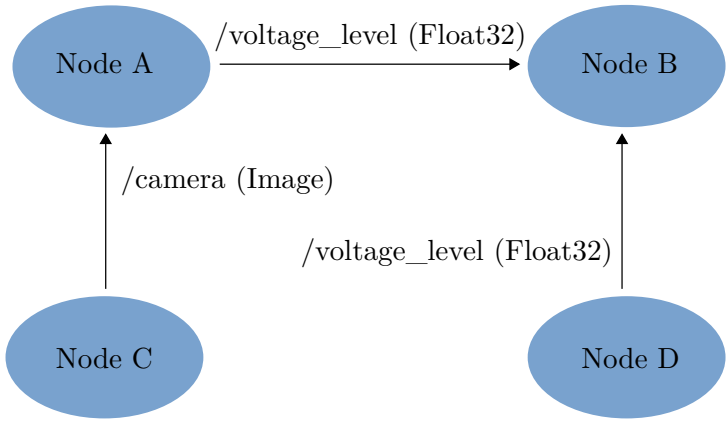


Figure 4.2.: Illustration of ROS publishers and subscribers.

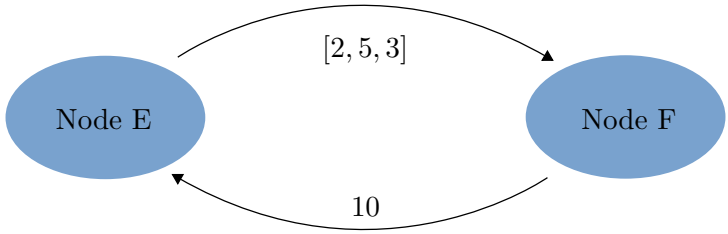


Figure 4.3.: Illustration of a ROS service where the client node E request a service from the server node F.

More information about ROS concepts like **parameters** and **bags** can be found at the ROS Wiki [12]. Figure 4.2 illustrates the publisher-subscriber communication in ROS. Node C publishes to the `/camera` topic and node A subscribes to it. The message type sent over this topic is of type `Image`. Node A and node D publishes to the `voltage_level` topic and node B subscribes to it. Node B will receive messages from both node A and node D of type `Float32`. A simple service is illustrated in Figure 4.3. Here, node E request node F to add a list of numbers and node F replies with the sum. The node handling the request and responding is typically called the server and the one requesting the service is called the client. In the figure, node E is the client and F the server.

4.3. Gazebo

Gazebo is a robot simulator. This open-source software provides a physics engine, 3D graphics and easy communication with ROS through a dedicated node. The

ROS-integration means that objects inside the simulation can be manipulated by sending messages from other ROS nodes to the Gazebo node. A typical example is that Gazebo subscribes to the `/cmd_vel` topic and robots inside the simulation can be controlled by specifying the wanted linear and angular velocity. Gazebo can also publish messages to other ROS nodes. The TurtleBot3 model that can be downloaded in Gazebo publishes LiDAR data, a camera stream and encoder data. This is the same data that would be published from the Raspberry Pi of a real TurtleBot. Because messages must follow a well-defined data structure, the data can be treated the same way no matter if it comes from Gazebo or from the actual TurtleBot.

The Gazebo simulator allows for a "digital twin". A virtual model of the real environment can be simulated in Gazebo. One can either import CAD models or build the world inside Gazebo. This allows for rapid testing during development and can be used for monitoring when the system is up and running. New tasks can first be simulated before they are initiated in the real world.

4.4. ROS Packages

Several useful ROS packages for mobile robots already exist. Here, some of the packages that lay the foundation for the functionality developed in this project are mentioned.

tf is a package that eases the handling of systems with many coordinate frames. Nodes can set up a listener that listens to transforms sent on the tf-topic. A transform describes the pose of a coordinate frame relative to another frame. The most typical example is the pose of the robot relative to the map's coordinate frame.

Navigation is a stack of 2D navigation packages for ROS. By combining information from odometry and other sensors, it sends velocity commands that aim to move the robot towards a specified goal pose. For localization it uses a Monte Carlo approach that is based on a particle filter, as described in section 3.5. The map is provided as an occupancy grid map, section 3.9. A path from the current pose to the goal pose is calculated by a global stochastic planner, section 3.6. The local planner keeps the robot from collisions with obstacles and uses the Dynamic Window Approach, section 3.7. Additionally, the Navigation stack implements functions for recovery when the robot is stuck or other errors happen. Planning and execution of navigation to a given pose is controlled by the `move_base`

action-server.

roslibjs is a package for interaction with ROS through the web browser using JavaScript. The package initializes a node and allows the browser to subscribe to topics, publish messages, call services and more through this node.

actionlib provides a standardized way of working with preemptable tasks. Among many others, the Navigation stack relies on actionlib. Contrary to the standard server-client setup, the action server-client setup allows for cancellation of services and feedback on the service progress. Thus, when a robot goal pose is requested, the server will continuously provide the client with the robot's current pose and the client can at any time cancel the navigation, effectively making the robot stop moving. Because of the feedback, action servers are especially useful for tasks that can take a long time to finish. A standard server can achieve the same results, but the client will not know the progress before the service is completed and it receives the response.

4.5. Other Software Used

Leaflet is a JavaScript library for interactive maps. Maps can be imported as image files and the programmer can define the wanted coordinate frame. The library lets you know where the map was clicked, place markers, trigger pop-ups and more. Leaflet.heat is a plugin for Leaflet that can be used to visualize a heatmap on top of a map.

OpenCV is an image processing library with a Python interface. Images can be loaded and represented as pixel intensities in a matrix. The library is typically used for computer-vision tasks and many popular image processing algorithms are implemented. Thresholding, edge detection and even feature detection are all available.

4.6. Important Messages and Actions

This section briefly lists and explains some of the ROS messages and actions that the work in this project has been built upon.

`TFMessage` expresses the pose of a child frame inside a parent frame. The trans-

form consists of a translation along a 3D vector and rotation described by a quaternion.

`Twist` is used to control the velocity of a robot. It consists of two 3D vectors. One vector for the linear velocity and one vector for the angular velocity.

`Odometry` contains both a pose, like `TFMessage`, and a twist, like `Twist`. Additionally, both of the two parts have an associated array of 36 numbers representing the covariance of the variables.

`LaserScan` holds the information from a LiDAR scan. It specifies the starting angle, the angle increment and the end angle. The distance before collision in these directions are specified in an array. Also, information about the lower and upper distance bound of a single reading is specified.

`Image` is a message that stores information about a RGB-image. The height and width of the image is stored together with the pixel intensities in a range from 0 to 255, listed in an array.

`OccupancyGrid` represents an occupancy grid map. It is similar to `Image` and contains the width and height of the map. The occupancy probability of a tile is stored in an array like the pixels of an image. Additionally, the resolution of the map is provided, which is the width of a tile in meters.

`MoveBaseAction` is an action that is used to set navigation goals for the robot to follow. The goal is specified as a pose. The feedback is the current pose of the robot. The result is an empty message that is sent when the robot has successfully reached the requested goal.

4.7. Implemented Messages, Services and Actions

Messages, services and actions that have been implemented in this project are explained in the following.

`ScanMismatches` is a message containing the resulting binary image when the original map is subtracted from the new map. It will have 1-pixels at the positions where it is believed that there is an obstacle.

`Obstacles` is a message that specifies the number of found obstacles and their

estimated centroids.

`DriveAround` is a service where the request is to move around an obstacle in a circle. The request specifies the position of the obstacle and the radius for the robot to keep from the obstacle. The response is a list of points. These points make up the estimated contour of the obstacle.

`TakePicture` is a service that requests the robot to take a picture of an obstacle at a given position.

`FollowRoute` is an action that specifies an ordered list of waypoints that the robot should visit in order. The goal is the list of waypoints, the feedback is which waypoint the robot is currently chasing and the result is a boolean value indicating if the route was completed successfully or not.

4.8. Implemented Nodes

This section briefly lists the nodes that have been implemented in this project. The intention is to give an overview of the nodes' responsibilities and how they communicate. Implementation details will be left to later chapters. Figure 4.4 shows a simplified graph of how the nodes communicate over topics. The use-case in the graph is that the robot should follow an inspection route and visualize found obstacles in the browser. Green nodes are pre-existing nodes of ROS and the blue ones are implemented in this project. However, note that in the full graph there are more nodes and topics than what is being shown in the figure.

`robot_pose_publisher` is responsible of updating the pose of the robot inside the Gazebo simulation. It subscribes to the `tf` topic and publishes a `LinkState` message over the `gazebo/set_link_state` topic.

`follow_route_server` handles a request to visit a list of checkpoints given by a `FollowRoute` action. For every checkpoint it sends a `MoveBaseAction` request to the `move_base` node. `move_base` takes care of planning the route, avoiding obstacles and sending motion commands to the robot's motors.

`gas_level_publisher` calculates the distance from the robot to the gas leakage and outputs a value based on this distance. The robot's position is found from a `tf` message and the sensor reading is published on the `eva/gas_level` topic as a `Float32` message.

`gas_sensor_monitoring` subscribes to the gas levels published by the robot and a wall-mounted gas detector. When the gas level reported by the mounted sensor exceeds a given threshold, the node commands `move_base` to make the robot navigate to a position close to the mounted sensor. This way, the robot can perform its own independent measurement.

`thermal_center` is a node that aims to find the position of a gas leakage in an image. It subscribes to the `Image` messages sent over the `turtlebot3_camera` topic. Further, it finds the pixels with a blue hue, calculates the centroid of these pixels and draws a marker. The result is published over `eva/gas_image` as an `Image`.

`scan_mismatches` perform a new mapping. By subscribing to `LaserScan` messages over the `scan` topic and receiving the robot's pose with `tf`, it performs mapping using ray tracing and an occupancy grid map. It also subscribes to `map` and is able to find occupied cells in the new mapping that are not occupied in the original map. The result is published as a `ScanMismatches` message on the `eva/scan_mismatches` topic.

`obstacle_publisher` analyzes the possible obstacles found during the new mapping and finds their centroids. The node subscribes to `eva/scan_mismatches` and performs morphological operations to remove noise. The centroids of the found obstacles are published as an `Obstacles` message over the `eva/obstacles` topic.

`take_picture_server` is responsible for finding a position to take a picture of an obstacle from given the position of the obstacle. The service is called with the `TakePicture` service. To find a position that does not cause collision with walls or similar, the node subscribes to the `map` topic and receives an `OccupancyGrid` message. When a suitable position is found, `move_base` is ordered to move the robot to this position before the node stores the latest `Image` message published over the `turtlebot3_camera` topic.

`drive_around_server` will let the robot drive around an obstacle given the obstacles position and the wanted distance between the obstacle and the robot. It finds a starting pose in a similar way as `take_picture_server`. Then it calculates how it should move to complete a circling of the obstacle and publishes this as a `Twist` message on the `cmd_vel` topic. As it moves around the obstacle, mapping is performed and the contour of the obstacle is returned when it has moved all the way around, as specified in the `DriveAround` service. The result of the mapping is received as a `ScanMismatches` message over the `eva/scan_mismatches` topic.

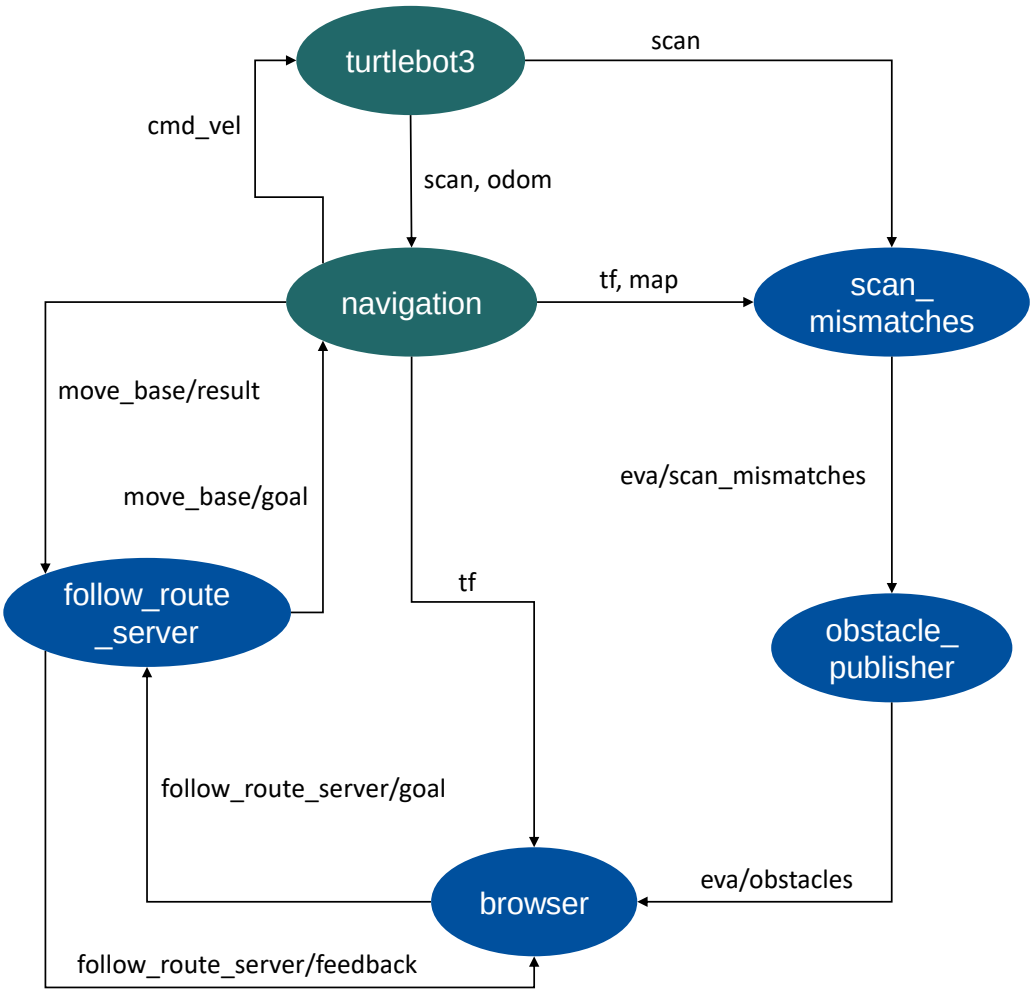


Figure 4.4.: Simplified graph of how the nodes communicate over topics during obstacle detection.

Chapter 5.

Design of Testing and Simulation Environment

A description of the preparations that were carried through before the testing is found in this chapter. The chapter also explains how the functionality of the robot was divided into five modes to showcase five different use-cases.

5.1. The World

A room of about 6 by 3.5 meters served as the testing environment, Figure 5.1. To replicate some of a platform's features, the room was furnished with a rectangular object, a circular object and additional walls. Also, it housed a set of thin cylinders to replicate a fence. Being able to detect thin objects, like the cylinders, can be important to avoid falling off a platform. Additionally, some areas of the room were designed to be narrow and allowed for testing the robot's path planning and maneuverability in tight spaces. Compared to a real unmanned offshore platform, the world that the robot was tested in is less challenging. A real platform will have a dynamic environment where equipment may move around. Opposite to this, the testing environment was static, as defined by Norvig and Russel [30]. Another difference to note is that on a platform there might be a multi-agent environment where different robots co-operate, while in the testing the TurtleBot was on its own.

Figure 5.1.: The testing environment.

5.2. Mapping

The mapping of the world was performed by the TurtleBot3. By teleoperating the robot in the environment, the occupancy grid map was generated using the built-in SLAM functionality of ROS. Also, a simplified map was drawn in Inkscape for illustration purposes. However, it is the raw map directly from the SLAM mapping that is used for navigation. A screenshot from the mapping process can be seen in Figure 5.2 and the final occupancy grid map is shown in Figure 5.3.

5.3. Modelling

The simplified map was imported in Siemens NX and a CAD model of the room was created based on the map. This model was exported to the STL file format and converted to the DAE mesh file format in the Blender software. This file is loaded into Gazebo, Figure 5.4. For the model of the robot, the same steps were performed. In the end, the digital environment consisted of three parts. These were a ground plane, a model of the room and the robot.

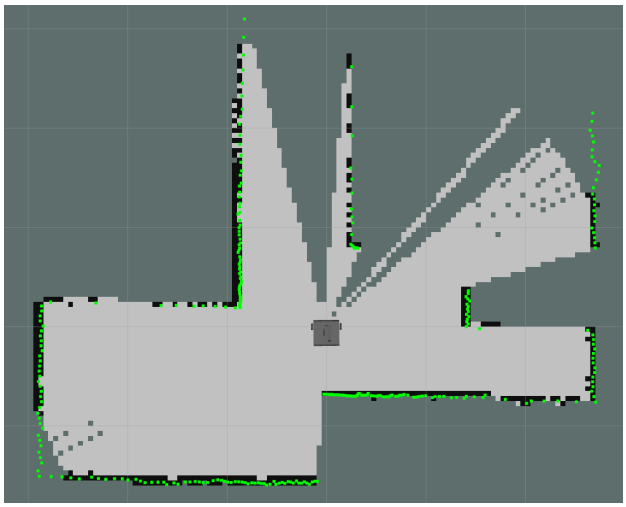


Figure 5.2.: SLAM mapping of the room.

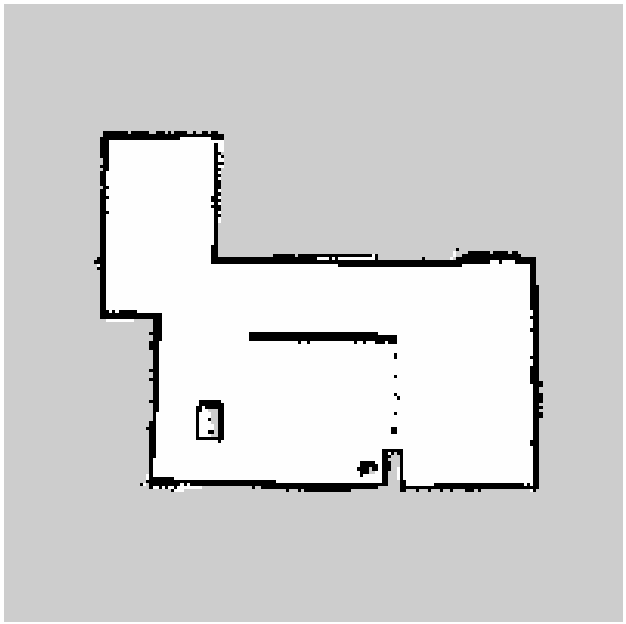


Figure 5.3.: Occupancy grid map of the room.

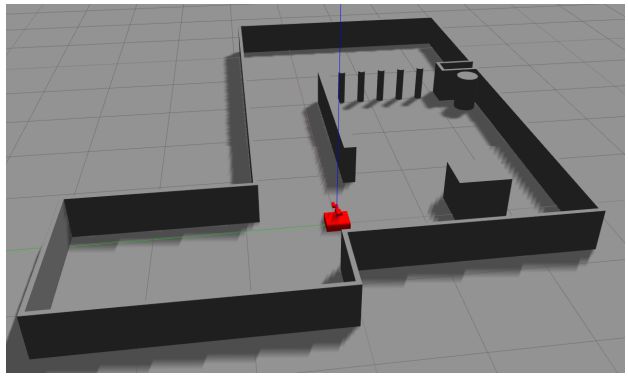


Figure 5.4.: CAD model of the world imported into Gazebo.

5.4. Operational Modes and User Interface

This project showcases several different responsibilities that an AGV can have on an unmanned O&G platform. Not all the implemented ROS nodes listed in section 4.8 need to be active to fulfill each responsibility. Thus, to simplify the system, five different modes were defined. Each of the modes have their own ROS launch file that initializes their needed nodes. Further ease of usability was gained by developing a simple web page for each mode. Such, all necessary interactions with the system could be controlled from the web browser. The next sections describe which functionality that is showcased in each of the five modes. Note that the robot used in the experiment does not have an actual gas sensor or IR camera. How these were simulated is explained in later chapters. Videos demonstrating each of the modes are attached in the delivery of this thesis.

Mode 1 is simple teleoperation. The user can control the robot's linear and angular velocity using the arrow keys on the keyboard. Real-time video from the robot is shown in the browser and the operator can utilize this mode to visually inspect wanted areas of the platform.

Mode 2 focuses on inspection of gas concentration. The gas level read by the robot's gas sensor is displayed to the user and the collected data is displayed on the map as a heatmap layer. Over time, areas of the platform with high gas concentration will become visible as the robot has visited those areas. Additionally, the robot's camera searches for areas within its frame with a low temperature and plots a marker where it believes there might be a gas leakage. In this mode, navigation is achieved by clicking the map and letting the robot autonomously navigate to the specified position.

Mode 3 implements a scenario where the robot is used in interaction with gas sensors mounted on the platform. When one of the mounted sensors trigger an alarm the robot autonomously moves to a location close to the sensor. Then, the robot performs its own independent measurement using the on-board gas sensor.

Mode 4 demonstrates the implemented obstacle detection in this project. The robot is instructed to follow a pre-defined route. Mapping has been implemented in this project and is used to map the environment during the inspection route. By comparing the new mapping to the original map, the robot is able to detect new obstacles that have been added to the environment. The estimated position of these obstacles are displayed in the browser. When the inspection route is finished, the operator can order the robot to take a photo of an obstacle by clicking on it. The picture will be displayed in the browser when ready. Additionally, the operator can order the robot to move in a circle around an obstacle to gather more information about its shape. A refined contour of the obstacle is then displayed on the map.

Mode 5 was implemented to demonstrate that the operator can cancel an inspection route at all times. If something of interest is picked up by the operator during an inspection, the operator can pause the inspection, teleoperate the robot, have a closer look at the point of interest and later resume the autonomous inspection.

Chapter 6.

Gas Leakages

The first methods developed in this project are presented in this chapter. A method requiring a mobile robot with a gas detector is explained first. Then, a possible application of a mobile robot with an IR-camera is outlined. Lastly, the experiment is explained and the results are presented and discussed.

6.1. Modelling a Gas Leakage

A simple way of simulating a gas sensor is to output a value based on the distance from the robot to the leakage. Assume that there is a gas leakage with a center at $\boldsymbol{r}_g^o$ and that the robot is currently at the position $\boldsymbol{r}_r^o$. The model used to simulate a gas sensor reading in this thesis is,

$$y = \begin{cases} 100\frac{s-||\boldsymbol{r}_g^o-\boldsymbol{r}_r^o||}{s} + v, & \text{if } ||\boldsymbol{r}_g^o - \boldsymbol{r}_r^o|| \leq s \\ 0, & \text{otherwise} \end{cases}, \tag{6.1}$$

where s is how far the gas spreads and v is zero-mean Gaussian distributed noise. Thus, the gas sensor readings depend linearly on the distance to the source.

In addition to gas sensors that analyze the composition of the air, IR-cameras can be used to detect gas leakages. When gas expands, the pressure drops and the temperature sinks. This can be seen as a distinct colour in an image from an infrared camera. In the experiments and simulations, gas leakages are modelled as blue spheres. Because the TurtleBot is delivered with a RGB-camera, but not an IR-camera, blue colours are here used to represent cold regions.

Figure 6.1.: Heatmap of the gas concentration reported by the robot.

The HSV colour space is well suited for detection of image regions of a specific colour. Instead of representing the light emitted from a pixel as a combination of red, green and blue light, the colours are determined by their hue, saturation and value. While the saturation and value will depend on the brightness in the environment, the hue should be independent of this. Therefore, we can filter out pixels that have a hue value outside of the desired region.

6.2. Heatmap

A heatmap is used to visualize the measured gas levels. Assume that the gas sensor reads a level of y' when it is at position (x_r, y_r). Then, the heatmap is updated at this position according to the intensity of the reading. As can be seen in Figure 6.1, saturated red colours are used to illustrate areas of high gas concentration. The functionality is implemented using the Leaflet.heat plugin for the map application Leaflet.

6.3. Gas Leakage Detection with IR Camera

After having established a way of determining which pixels of a camera image that are likely to be a part of a gas leakage, this information can be further processed. From the filtering we have a binary image with 1-pixels at the positions where it

is most likely that there is a gas leakage. A logical extension is to calculate the position of the centroid in the image. Equation (3.20) allows us to do this,

$$\bar{x} = \frac{M_{10}}{M_{00}} \qquad \bar{y} = \frac{M_{01}}{M_{00}}.$$

If the prior steps of filtering out the regions of the image where there are gas leakages have been successful, this centroid will be the at the center of the gas leakage. Figure 6.2 shows the required steps to visualize the position of the centroid.

6.4. Experiment Setup

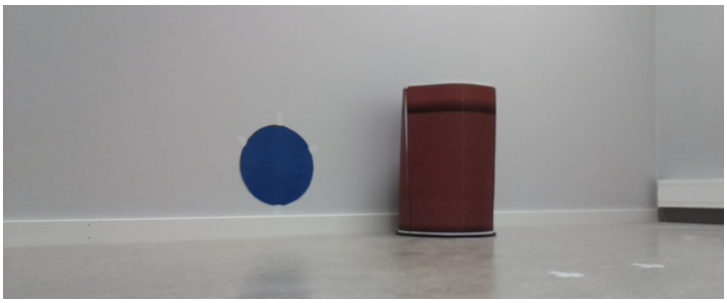
A blue disc of diameter 15 cm was attached on the wall at the position $(-1.5, 1)$, in the object frame (map), 15 cm of the ground. The HSV ranges to filter out the blue colour were set to $H \in [100, 140]$, $S \in [50, 255]$ and $V \in [50, 255]$. Note that OpenCV discretizes the hue into the range $[0, 179]$ while the saturation and value are discretized into the range $[0, 255]$. Testing of the robot's ability to locate the blue disc was conducted by moving the robot around and noting if the marker was in the correct position.

6.5. Results

The center of the blue disc was easily located in the video stream from the robot's camera. However, it did happen that pixels in the correct range were found also when the disc was not within the image frame. This could be other blue objects or simply noise. Visualization of the gas concentration as a heatmap also functioned as planned. Red regions appeared on the map close to where the center of the gas leakage was set.

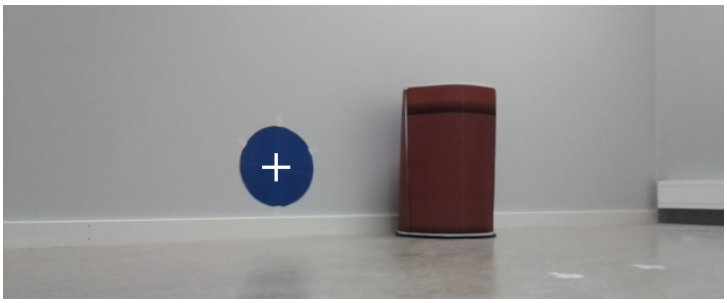
6.6. Discussion

A possible use of an infrared camera has been presented in this chapter. Assuming that it is possible to set a temperature threshold that separates the expanding gas from the environment, in the image, the work shows that it is possible to



(a) Image from camera stream.

(b) Result after HSV filtering Figure 6.2a.



(c) Centroid of Figure 6.2b projected onto Figure 6.2a.

Figure 6.2.: Steps of finding the center of a gas cloud.

estimate the position of the gas leakage in a real-time video. However, this is only the position on the image plane. A more useful feature would be to estimate the position of the gas leakage in the object (map) frame. Then, it would be possible to plot the leakage on a map. One way of doing this could be to take pictures from different angles and perform triangulation to calculate the depth. A stereo-vision setup with two infrared cameras is another approach that could be worth exploring. Note that all this depends on the assumption of being able to set a temperature threshold for what is considered a possible gas leakage. Platforms located in arctic environments could potentially challenge this assumption since the surrounding temperature can be as cold as the expanding gas.

The second feature demonstrated in this chapter is the generation of a heatmap. Even though the setup was not tested with a real gas sensor, the experiments show that it is possible to generate a visualization of the gas concentration by letting the robot visit different parts of the platform. Whether or not a visualization with a heatmap is meaningful must be tested in a scenario with a real gas leakage. Maybe a simpler presentation where one only distinguishes between over or under a threshold value is better. However, the ability of the robot to perform measurements that are independent of the measurements taken by the sensors mounted on the platform is valuable. If a mounted sensor measures a value above the threshold, the robot can be summoned to a position close to where the measurement was taken. Now, if the robot also measures a dangerous gas concentration, the production on the platform could have to be paused. On the other hand, if the robot does not measure a dangerous concentration, the production can continue until human operators can come to the platform.

6.7. Further Work

Testing with a real infrared camera must be performed before one can draw a conclusion on how useful such a camera would be for the purpose of gas leakage detection. Ideally, a 3D vision system should be used. This is because it would allow the robot to estimate the position of the leakage in the frame of the map, not only in the image plane. As mentioned, a stereo-vision setup could be one solution. Another solution could be to make the robot take pictures of the same leakage from different positions and calculate the depth based on two images and the two corresponding robot poses. A different kind of sensor that one could experiment with is a microphone. Leakages produce sound and this information can be used either separately or in combination with gas concentration sensors or infrared cameras.

Chapter 7.

Obstacle detection

This chapter is structured in the same way as the previous chapter. The docus is now on detection of obstacles on the platform using a mobile robot with a LiDAR. An iterative process of trial and error was carried through before the final method was established. The chapter will also explain some of the discarded methods.

7.1. Map Transformation

To relate a point in space to a pixel on the map, a coordinate transform from the world to the pixel coordinates of the occupancy grid map is needed. With the settings used in this project, a square meter of the world is divided into a grid of 20 by 20 pixels. Hence, the resolution, ρ, is 0.05 meters per pixel. The origin of the object frame is placed at $(200, 184)$ in the coordinates of the occupancy grid map image, as seen in Figure 7.1. If we apply the camera model on this problem, the camera matrix becomes,

$$K = \begin{bmatrix} \frac{f}{\rho} & 0 & u_0 \\ 0 & \frac{f}{\rho} & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 20 & 0 & 200 \\ 0 & 20 & 184 \\ 0 & 0 & 1 \end{bmatrix}.$$

As there is not a real camera capturing the image, we cannot think of the focal length f in terms of a physical distance and it is set to a unit value of 1. Because the z axis of the camera model should point away from the camera, but the z axis in the ROS Navigation stack is defined upwards, a rotation of π around the x axis

is needed from the object frame to the camera frame,

$$T_o^c = \begin{bmatrix} R_o^c & \boldsymbol{t}_{co}^c \\ \mathbf{0}^T & 1 \end{bmatrix},$$

where the rotation matrix is,

$$R_o^c = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\pi) & -\sin(\pi) \\ 0 & \sin(\pi) & \cos(\pi) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix},$$

and $\boldsymbol{t}_{co}^c = [0, 0, 1]^T$. The translation from the camera frame to the object frame along the z axis is set to match the fictional focal length, $f = 1$. Then, the position of the point in the camera frame is,

$$\tilde{\boldsymbol{r}}_{cp}^c = T_o^c \tilde{\boldsymbol{r}}_{op}^o.$$

The normalized image coordinates $\tilde{\boldsymbol{s}}$ are equal to $\boldsymbol{r}_{cp}^c$ because of the unit distance from the camera frame to the object frame. At last, the pixel coordinates are given as,

$$\tilde{\boldsymbol{p}} = K\tilde{\boldsymbol{s}} = K\boldsymbol{r}_{cp}^c.$$

Combining all the operations, we end up with

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 20 & 0 & 200 \\ 0 & -20 & 184 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x^o \\ y^o \\ 1 \end{bmatrix}. \tag{7.1}$$

7.2. Ray Tracing

The purpose of the ray tracing is to determine which grid cells of the occupancy map a given laser ray passes through and at which grid cell it hits a surface. For this calculation, the pose of the robot in the object frame, the angle of the beam in the robot frame and the distance from the robot to the closest object along the ray are given. Note that the pose of the robot is estimated and that this can contribute to mismatches between the real path of the laser ray and the traced ray.

Let $\boldsymbol{t}_{or}^o$ be the position of the robot frame relative to the object frame and let R_r^o be the rotation from o to r. The rotation matrix describes a rotation of θ_r

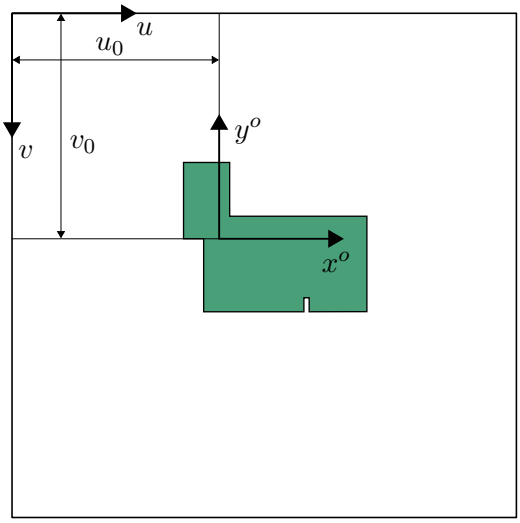


Figure 7.1.: Map plane and object frame relationship.

around the z axis. This transformation is given by the navigation package and is published as a `tf` message. Furthermore, let the angle of the laser beam relative to the robot frame be $\theta_{l,i}$. Then, the transformation from the object frame to the frame of laser beam number i is,

$$T^o_{l,i} = \begin{bmatrix} R^o_r & \boldsymbol{t}^o_{or} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} R^r_{l,i} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} R_z(\theta_r) & \boldsymbol{t}^o_{or} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} R_z(\theta_{li}) & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} R_z(\theta_r + \theta_{l,i}) & \boldsymbol{t}^o_{or} \\ \mathbf{0}^T & 1 \end{bmatrix}.$$

Bresenham's line algorithm requires the starting grid cell and the angle of the ray to be projected, as presented in section 3.8. $\boldsymbol{t}^o_{or}$ is transformed into u and v coordinates by equation (7.1). As the z axis of the object frame and the camera frame point in opposite directions, the angle of the line in the grid map becomes $-(\theta_r + \theta_{l,i})$. Let the laser reading for the given ray be a distance of y. This corresponds to $y' = \frac{y}{\rho}$ cells. All cells encountered during the ray tracing before the line is of length y' are cells that the laser passed through. The cell where the length of the line becomes y' is the cell where the LiDAR believes there is an object. The occupancy grid map is then updated according to these results.

7.3. Related Work

Obstacle avoidance is a well known topic in the literature. The Dynamic Window Approach, section 3.7, will let the robot deviate from the planned route if obstacles show up. However, the task of mapping the obstacles does not seem to be an equally well-documented topic. This section presents the prior work that was found to be most relevant for the wanted obstacle detection functionality of this thesis.

Dogan [8] introduces "a probabilistic approach to the problem of mission planning for UAVs (Unmanned Aerial Vehicles) flying through an area of multiple sources of threat". Every threat is defined by its expected position $\boldsymbol{\mu}_i$ and its covariance matrix K_i. Given N threat sources, the total threat at a position $\boldsymbol{r}$ is then calculated as the sum of all the multivariate normal distributions at $\boldsymbol{r}$,

$$f(\boldsymbol{r}) = \sum_{i=1}^{N} \frac{1}{2\pi\sqrt{\det(K_i)}} \exp[-\frac{1}{2}(\boldsymbol{r} - \boldsymbol{\mu}_i)^T K_i^{-1}(\boldsymbol{r} - \boldsymbol{\mu}_i)]. \tag{7.2}$$

The probabilistic map given by the probability density function is known as a Probabilistic Threat Exposure Map (PTEM). Instead of modelling threat sources, Desai [7] utilised the PTEM method to map cylindrical obstacles with an AGV using a LiDAR. $\boldsymbol{\mu}_i$ is then the mean position of the obstacle and K_i is the covariance that depends on the size of the obstacle. Equation (7.2) is used to calculate the probability of there being an obstacle at position $\boldsymbol{r}$.

The solution proposed by Desai is not directly applicable in this project. Desai found obstacles in an open space, but for a robot navigating on an offshore platform one has to take permanent structures like walls, pipelines and more into account. These structures should not be considered as obstacles. Obstacles are objects that are not present in the specifications of the platform. These can be parts that have fallen down, bird nests, tools left by operators and similar. In the following, let m be a map of the platform that was generated when no obstacles were present. A first attempt at mapping obstacles, in this project, was by extending the occupancy grid mapping to differentiate between obstacles and permanent structures.

7.4. Obstacle Grid Mapping

Let $z_{x,y}$ be the event that the LiDAR scan reports that cell (x, y) of a grid map is occupied. Let $m_{x,y}$ be the event that cell (x, y) is occupied in the original map. The aim of the algorithm discussed here is to find obstacles. These can be expected to be found in cells that are not occupied in the original map, but now appear to be occupied. The conditional probability of a cell containing an obstacle can be expressed as

$$P(o_{x,y} = 1 | z_{x,y}, m_{x,y})$$

With the two binary variables $z_{x,y}$ and $m_{x,y}$, there are four possible conditions.

- $z_{x,y} = 0, m_{x,y} = 0$
- $z_{x,y} = 1, m_{x,y} = 0$
- $z_{x,y} = 0, m_{x,y} = 1$
- $z_{x,y} = 1, m_{x,y} = 1$

$P(o_{x,y} = 1 | z_{x,y}, m_{x,y})$ will be greatest in the second case where $(z_{x,y} = 1, m_{x,y} = 0)$. Here, the LiDAR reports that there is something blocking the way, but the cell is not occupied on the original map. However, the measurement of the LiDAR is not error-proof and also the original map can contain errors. Hence, the probability is not 100%. For the other three conditions, the probability should be low. The equation for updating the conditional probability of an obstacle occupying a cell becomes,

$$\begin{aligned} \log\Big(\frac{P(o_{x,y} = 1 | z_{x,y}, m_{x,y})}{P(o_{x,y} = 0 | z_{x,y}, m_{x,y})}\Big) = \log\Big(\frac{P(z_{x,y}, m_{x,y} | o_{x,y} = 1)}{P(z_{x,y}, m_{x,y} | o_{x,y} = 0)}\Big) \\ + \log\Big(\frac{P(o_{x,y} = 1)}{P(o_{x,y} = 0)}\Big). \end{aligned} \tag{7.3}$$

This is a development of equation (3.18).

As mentioned, this method is an effort to extend occupancy grid mapping. Unfortunately, testing proved that uncertainties in both the pose of the robot and the LiDAR range readings resulted in poor performance. If a laser beam reflected off a wall, but inaccuracies caused the reading to be a little too short, the algorithm would read $m_{x,y}$ in a grid cell right before the wall. When this cell is not occupied

on the original map, the algorithm would increase the likelihood of this cell containing an obstacle. All in all, the result was that false obstacles were detected just in front of most walls. An attempt was made to improve the algorithm by checking not only the exact cell where the laser beam was reported to reflected off, but also the neighbourhood for occupancy. This gave a better result, but not as good as wished for. Also, it introduced complexity to the method and defining an appropriate size of the neighbourhood proved difficult. Because of the poor results, a new method was tried out, one that utilized morphological image processing.

7.5. Obstacle Mapping using Morphology

The obstacles will occupy the cells which are now occupied, but were not occupied during the original mapping. Let S be the set of occupied cells from the current mapping and let M be the set of occupied cells from the original mapping. The set of cells that are occupied by obstacles can be expected to be found as

$$O = S \cap \bar{M} = S - M.$$

Walls and other permanent structures will not necessarily be mapped in exactly the same grid cells in the new scan, S, as in the original scan, M, because of noise. To ensure that all these structures are removed properly, they are enlarged with a dilation by a structuring element B_0,

$$M' = M \oplus B_0.$$

Instead of subtracting the original map, this dilated map is subtracted,

$$O_1 = S - M'. \tag{7.4}$$

An obstacles can appear in the binary image as a set of structures located close to each other. To connect these structures, a closing operation is performed,

$$O_2 = O_1 \bullet B_1.$$

To further improve the result, an opening operation is utilized to remove noise,

$$O_3 = O_2 \circ B_2.$$

Due to a combination of inaccuracies in the pose estimation and the sensor readings, the new mapping will sometimes map pixels far outside the map. Let B_3 be a mask with ones inside the borders of the map and zeros outside. To remove all incorrectly detected points outside the map, the intersection between the occupancy map and the mask is calculated,

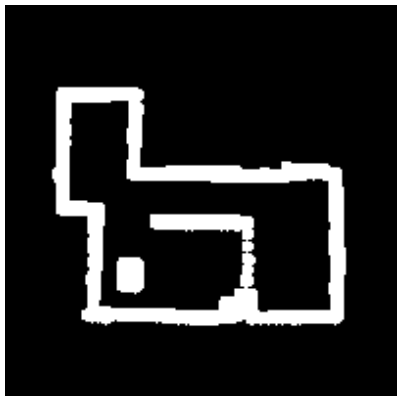
$$O_4 = O_3 \cap B_3.$$

The last step in locating the obstacles is to convert the binary image O_4 to a set of contours, one for each coherent group of 1-pixels. This is achieved with the `findContours()` function in OpenCV, as described in section 3.11. Each contour is defined by an ordered list of points and the contour can be drawn with straight lines between each successive point. For each of the contours, the centroid of the contour is calculated with equation (3.20). Centroid $(\bar{x}_i, \bar{y}_i)$ is interpreted as the center of obstacle i. Figure 7.2 explains the steps from a new scan to a contour for each obstacle.

7.6. Alternative Obstacle Mapping with DBSCAN

As an alternative approach to the many morphological operations presented in section 7.5, clustering using DBSCAN (explained in the appendix, section B.7) was explored. However, the method still depends on morphology in the initial step to subtract the original map from the new mapping. Therefore, the input to the method is the binary image O_1 of equation (7.4). Then, the binary image is converted into a list of pixel positions of the 1-pixels. These points are clustered using the DBSCAN algorithm. The result is a set of labeled points, as seen in Figure 7.3. Points with matching labels are assumed to belong to the same obstacle.

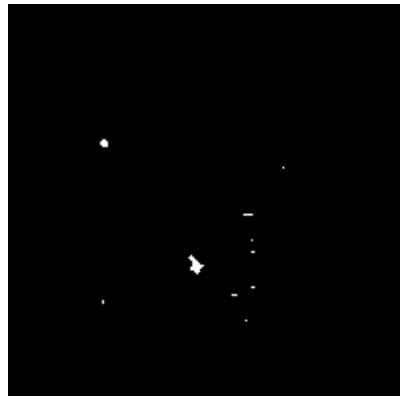
Unfortunately, early tests of the DBSCAN method indicated that the morphology approach achieved superior results. Simply subtracting the map and applying DBSCAN resulted in the detection of numerous false obstacles. The results improved if more morphological pre-processing was performed, but due to the excessive complexity introduced by both morphological pre-processing and the following clustering, it was deemed that an approach solely based on morphology was better.



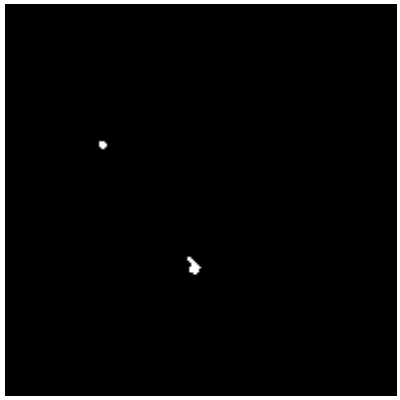
(a) Dilated map.

(b) New mapping.

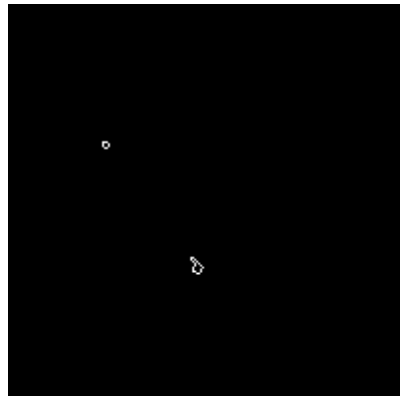
(c) Figure 7.2a subtracted from Figure 7.2b.



(d) Closing of Figure 7.2c.



(e) Opening of Figure 7.2d.



(f) Contours of Figure 7.2e.

Figure 7.2.: Steps of the morphology method.

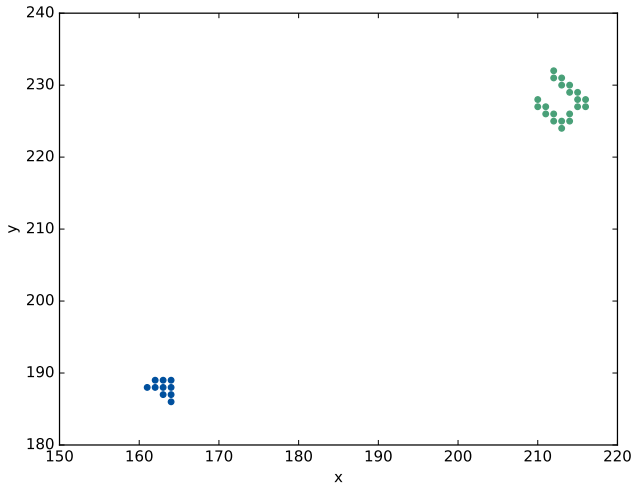


Figure 7.3.: Result of DBSCAN applied on Figure 7.2c.

7.7. Photograph Obstacle

When an obstacle is found, many possibilities open up. To photograph an obstacle the robot needs to be facing directly towards the center of the obstacle from a distance. Let (x_c, y_c) be the coordinates of the center of the obstacle and let R be the wanted distance from where the robot captures the image to the obstacle's center. Possible capturing positions are given by,

$$(x, y) = (x_c + R\cos(\theta), y_c + R\sin(\theta)), \theta \in [0, 2\pi).$$

For a given θ the corresponding angle of the robot is $\theta' = \theta - \pi$ such that the robot faces towards the obstacle.

A set of possible capturing poses are calculated by sampling θ. Because some of the positions might already be occupied by walls or other objects, a filtering is needed. For all poses, a check is performed to ensure that there is sufficient space for the robot by inspecting the occupancy grid map. If there is not space for the robot at the position, the pose is discarded. Lastly, the position closest to the current position of the robot is picked. Figure 7.4 illustrates eight possible capturing poses. When a pose is chosen, an action client is initialized. The client requests `move_base` to plan and execute a path to obtain this pose. Upon successful execution, the latest image published by the camera node is forwarded to the web browser and is displayed in a pop-up.

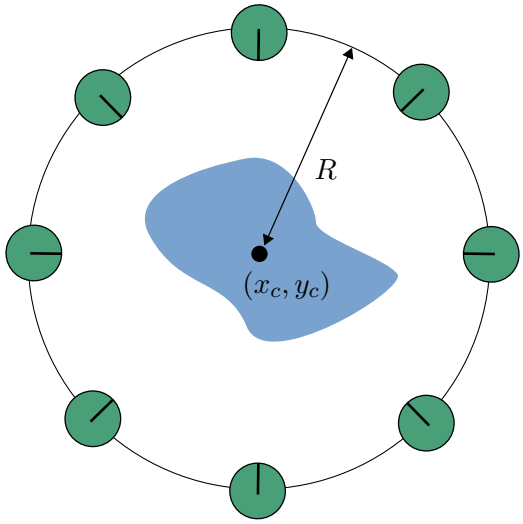


Figure 7.4.: Eight possible poses for photographing of the obstacle.

7.8. Inspection of a Single Obstacle

Moving around an obstacle will let the robot sense the obstacle from every angle. This can give a more complete picture of the obstacle than what has been sensed during an inspection route. Possible starting poses can be found in the same way as the capturing poses, but with an angle of $\theta' = \theta + \frac{\pi}{2}$ instead, normal to the radial axis. Let R be the desired distance to keep from the center of the obstacle, (x_c, y_c). The distance the robot will have to travel to complete a whole circling is,

$$d = 2\pi R.$$

If the robot moves with a linear velocity of v, the operation will have a duration of,

$$t = \frac{d}{v} = \frac{2\pi R}{v}.$$

Also, during the operation the robot will need to rotate an angle of 2π. To do this in a time of t requires an angular velocity,

$$\omega = \frac{\Delta\theta}{t} = \frac{2\pi}{\frac{2\pi R}{v}} = \frac{v}{R}.$$

When the starting pose is reached, a `Twist` message is published on the `cmd_vel` topic with a linear velocity of v and an angular velocity of $\frac{v}{R}$. A stop signal is sent after a time interval of $\frac{2\pi R}{v}$.

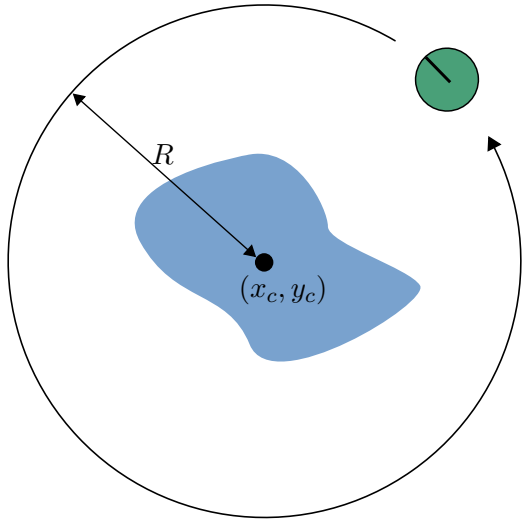


Figure 7.5.: Moving around the obstacle at a distance of R.

When the robot has finished the tour around the obstacle, Figure 7.5, the area can be inspected further. The obstacle must be within a square defined by the top left corner $(x_c - \frac{R}{2}, y_c + \frac{R}{2})$, and the bottom right corner $(x_c + \frac{R}{2}, y_c - \frac{R}{2})$. This square is transformed into pixel coordinates using the camera model, section 7.1. Information about the shape of the obstacle can be extracted from this square region of the occupancy grid map. A morphological closing operation is performed in the extracted region to connect the 1-pixels making up the obstacle. Then the OpenCV function `findContours()` is used to obtain an ordered list of vertices making up the boundary of the obstacle in the map image. This list is sent to the browser and the obstacle is visualized on the map as a Leaflet `polygon` object.

7.9. Specification of Inspection Route

In this thesis, an inspection route is defined as an ordered list of checkpoints that the robot should visit in order. Every checkpoint p_i consists of a Cartesian (x, y) coordinate in the plane. To initialize the inspection route, an implemented action-client sends the list of checkpoint to an implemented action-server. The server goes through the checkpoints one by one and communicates with the `move_base` action server. `move_base` is requested to plan and execute a path to the checkpoint before the server moves on to the next checkpoint and repeats the process until the last checkpoint is reached. During the execution of an inspection route, the action-server sends feedback to the action-client in form of the current checkpoint

i that the robot is chasing. Also, the action-client can at any time cancel the action and the robot will stop when it reaches the next checkpoint.

An angle θ is calculated for every checkpoint to increase the flow of the inspection route. The angle is calculated such that the robot is facing towards the next checkpoint when the current checkpoint is reached. Let (x_i, y_i) be the coordinates of the current goal checkpoint and (x_{i+1}, y_{i+1}) be the coordinates of the next. Then, the current checkpoint angle θ_i is calculated as,

$$\theta_i = \text{atan2}(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}).$$

This gives the checkpoint a more meaningful goal direction than for example a fixed direction of $\theta_i = 0$.

7.10. Experiment Setup

The structuring element used in the dilation operation of the map (B_0) was a circle of radius six pixels. B_1, the structuring element in the closing operation, was defined as a square with a side length of two pixels. B_2 in the opening operation was set equal to B_1.

Three different obstacles were used in the experiments. Their dimensions and shapes are described in the following.

1. Cardboard box with dimensions 0.31 m $\times$ 0.4 m
2. Cardboard box with dimensions 0.19 m $\times$ 0.25 m
3. Plastic cylinder with a diameter of 0.2 m

The number of obstacles placed out before every test varied. Sometimes all the three obstacles were out in the room, sometimes none. The positions at which the obstacles could be placed are shown in Figure 7.6. These positions were measured by hand, marked on the floor and their coordinates were known.

When the obstacles were in place, the robot was initialized and it was commanded to move to the checkpoints, marked in Figure 7.6, in order. During the inspection route, the robot performed occupancy grid mapping. When the last checkpoint was reached, the system performed the morphological operations specified in sec-

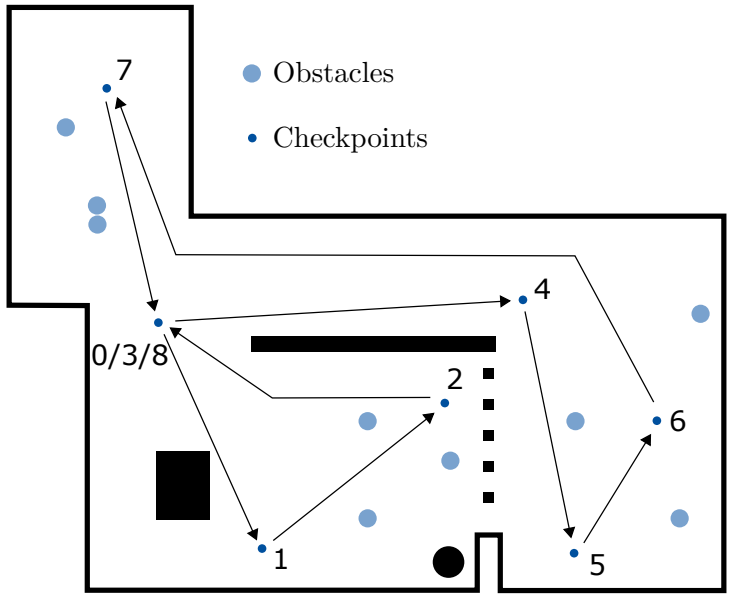


Figure 7.6.: Checkpoints of the inspection route and the possible obstacle locations used in the experiment.

tion 7.5 on the newly created occupancy grid map and the centroids of the found obstacles were published. These estimated centroids were compared to the actual obstacle positions of the particular test. To evaluate the accuracy of the results, root mean square error (RMSE) is used,

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}((\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2)}, \tag{7.5}$$

where $(\hat{x}_i, \hat{y}_i)$ is the estimated position of obstacle i's centroid and (x_i, y_i) is the actual position.

7.11. Results

In total, the robot successfully located 17 out of 19 obstacles across the 10 iterations of the test. However, the accuracy of the estimated positions varied and the RMSE of the 17 detected obstacles was 0.190 m, calculated from equation (7.5). The greatest euclidean distance between an obstacle and its estimated position that was seen in the experiments was 0.283 m. The complete results can be found in Table A.1.

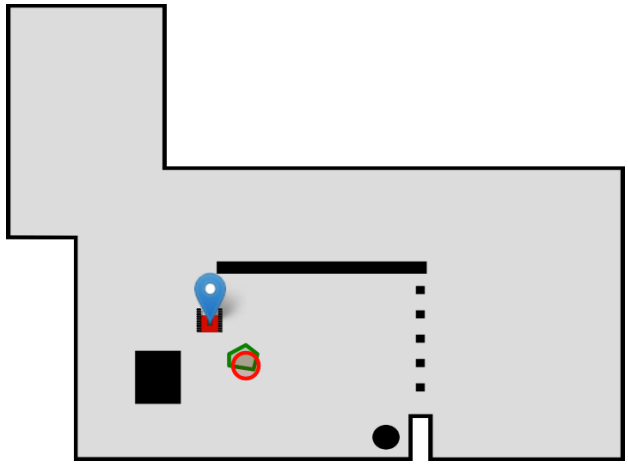


Figure 7.7.: Estimated position of the 0.2 m diameter cylinder in red and the refined shape in green.

Of the 19 obstacles, two obstacles were not found at all. The first one was detected during the inspection route, but was later considered as noise and discarded before the robot reached the end of the inspection. The second one was not found because it was placed close to another obstacle. By inspecting the intermediate steps of the morphological operations, one could see how the obstacle was considered as part of the nearby obstacle and that they merged into one larger obstacle.

The robot did not report any false obstacles during the experiments. As already stated, the greatest euclidean distance from a true obstacle position to a reported obstacle position was 0.283 m. In other words, it did not occur that noise caused the robot to believe there were more obstacles present than in reality.

Photographing of detected obstacle worked as expected during the testing. The robot was always able to find a suitable pose to capture the image from and the images popped up in the web browser window. When it comes to the more exhaustive inspection of a single obstacle, like explained in section 7.8, the results were somewhat more varying. For the largest of the obstacles, the polygon added on top of the map in the web browser did add some information compared to just visualizing the estimated centroid. However, for the smallest obstacle, the added polygon was too inaccurate to reveal information about its shape. The cylindrical shape of the smallest obstacle was not clearly displayed, as shown by the green contour in Figure 7.7. It is also worth noting that, at some occasions, the robot was not able to move all the way around the obstacle because of collisions.

7.12. Discussion

The process of developing a method for obstacle detection went through several iterations of trial and error. All of the methods showed some promise, but the quality of the estimations varied greatly. The first effort extended the occupancy grid mapping by updating the log-odds of a grid cell containing an obstacle. The difference from the occupancy grid mapping is that the method distinguishes between obstacles and permanent structures of the environment. This was first combined with a Hough circle transform (explained in the appendix, section B.6) to try and find cylindrical obstacles. Obviously, this method had its limitations because it could only locate cylindrical obstacles, but this was not the greatest problem. The "obstacle grid mapping" was simply not robust enough against noise. It is likely that a higher accuracy LiDAR could have improved the performance, but it is uncertain to what extent. The next method was based on the original occupancy grid mapping, as explained in section 3.9. Because the obstacles are those structures in the environment that are present now, but was not present during the original mapping, the idea was to find the obstacles by subtracting the original map from the new map. It was quickly seen that noise was still a problem, but by utilizing morphological operations the noise was reduced. Lastly the unsupervised clustering algorithm DBSCAN was tried as an alternative to the many morphological operations, but the results showed little improvement and the method was not looked further into.

Encouraging results were seen during the testing. Most importantly, the robot was able to estimate the approximate position of the majority of the obstacles. As a tool for getting an overview of potential obstacles on the platform, the demonstrated accuracy can be sufficient. Comparing the results to the work of Desai [7], the performance is a little poorer. His results equate to a RMSE of 0.067 m while a RMSE of 0.190 m was experienced in this thesis. Yet, it should be noted that in his experiments the robot was standing still at a known position, while in this project the robot is moving and its pose is estimated. Thus, it is hard to compare the two. Significant effort was devoted in this project to find more research concerned with the problem of estimating obstacle positions, but unfortunately without success.

The developed obstacle detection relies on a large number of parameters. How these are set can greatly influence the performance of the method. Let

$$P_{\text{tp}} = P(z_{x,y} = 1 | m_{x,y} = 1) \text{ and } P_{\text{fp}} = P(z_{x,y} = 1 | m_{x,y} = 0),$$

be the probabilities of a true positive reading and a false positive reading. Firstly,

the occupancy grid mapping requires one to set P_{tp} and P_{fp}. These are the probabilities that the laser reports a cell to be occupied given that it is occupied and the probability that the laser reports a cell to be occupied given that it is not occupied. The more accurate the sensor is, the higher P_{tp} can be set. Also, the more accurate the sensor is, the lower P_{fp} can be set. On one hand, by setting P_{tp} high and P_{fp} low, noise can cause grid cells to be falsely classified as occupied. On the other hand, by setting P_{tp} and P_{fp} both closer to 0.5, many of the cells that are actually occupied will probably not be classified as containing obstacles. Because the results from the mapping are utilized in the later steps of the obstacle detection, these two probabilities will influence the final result. Assuming that the sensor is highly accurate can cause the method to state that there are obstacles at positions were there are not obstacles. In the opposite case, the method can miss the detection of obstacles.

The morphological operations in the obstacle detection rely on additional parameters. The first one is the structuring element B_0 used in the dilation of the original map. A large structuring element will grow the original map more and when this map is subtracted from the new map, a larger area will be subtracted. If B_0 is set too high, this can cause obstacles located close to the walls to be removed. However, if set too low, the noise in the new map close to the walls will not be removed properly and this can cause false detection of obstacles close to the walls.

Two more parameters are important to the morphological operations. These are the structuring element (B_1) of the closing operation and the structuring element (B_2) of the opening operation. The closing is meant to connect pixels that belong to the same obstacle, but have been split by one or more 0-pixels due to inaccuracies in the mapping. If the size of B_1 is small, one runs the risk of the closing not being able to connect structures that belong to the same obstacle. However, a large size can make the closing connect two or more obstacles located close to one another. While the closing operation is meant to tie together the pixels, the opening operation is utilized to remove noise. All structures smaller than the structuring element (B_2) will be eliminated by this operation. Therefore, it is important that B_2 is not too large as this can result in the elimination of obstacles in the binary image. On the other hand, a too small structuring element will not be able to remove all the noise if there are specific areas with a lot of noise in the image.

As seen in the results, the system did not report any false obstacles. This suggests that some more adjustments of the parameters could be beneficial. The event experienced during the testing when an obstacle was detected at an early state, but later discarded, suggests the same. Both these results indicate that the parameters could be set less conservatively. Especially, increasing P_{tf} and decreasing P_{fp} could

prevent already detected obstacles from being discarded. The second obstacle that was not detected, because it was merged with another nearby obstacle, indicates that the structuring element of the closing operation (B_1) could be made smaller. However, this could again lead to the morphological operations not being able to tie together all the pixels belonging to the same obstacle. In short, there are trade-offs to be made for every parameter. Let a type 1 error be that the system reports an obstacle when the obstacle is not actually present and let a type 2 error be that the system fails to report an obstacle that is actually there. Then, the type of application should determine which of the two errors that are most crucial to avoid. In the setting of an unmanned offshore platform, one can imagine that a type 2 error is most crucial as it is important to not miss any obstacles. Probably it is better to report that a part has fallen down one time too many than to not detect that a part has fallen down.

Estimation of obstacle positions can be a useful feature, but needs to be combined with other features to become truly powerful. The onshore operator would like to know what type of obstacles that are on the platform, not only their positions. This is where the photography feature enters. During the testing, this feature showed promising results and the robot was able to capture the whole obstacle within the frame every time. It is possible that a real-world scenario would be more challenging than what has been demonstrated in the experiments. For instance, the algorithm does not take into account if there is something in-between the robot and the obstacle when taking the picture. It only makes sure that there is sufficient space for the robot and that it is facing in the correct direction. Other small bugs like this might exist, but should be relatively easy to uncover and fix with some more testing.

The other feature that combines with the obstacle position estimation is the extended inspection. The algorithm aims to provide the operator with a contour of the obstacle based on the LiDAR measurements. Testing proves that the method works, but also that the performance is not the best. For obstacles of 0.3 m by 0.3 m or more, the method can give the operator some insight about the shape and size of the obstacle, but for smaller obstacles it does not provide any more meaningful information than what an estimation of the centroid's position provides. On the whole, it remains a little unclear whether this extended inspection will be useful on an O&G platform. As the situation stands now, the photography function is more valuable than the visualization of the shape. It should also be noted that the planing of the route around an obstacle needs more work. Collisions did happen during the testing because the robot moves around the obstacle in a circular path without considering walls or other structures that might intersect with the circle.

7.13. Further Work

A lot can be done to increase the performance of the obstacle detection. The parameters can definitely be optimized through further testing. If one wishes to experiment with methods that are not based on morphology, this could also be interesting. DBSCAN was tried here, but several other clustering algorithms exist and might work better. Nevertheless, for all of these methods to work, a reliable mapping is needed. A phenomenon experienced during testing was that the ray tracing implemented in this project would sometimes perform poorly. This was especially apparent when the robot was rotating. It is reason to believe that the pose estimation is less accurate while the robot is rotating and that substantial errors in the angle estimation caused a mismatch between the simulated rays and the actual LiDAR rays. An idea for fixing this is to only perform the ray tracing when the uncertainty in the robot's pose is below some threshold.

Automatic photographing of obstacles was demonstrated in this project and there are many possible extensions of this work. An image classifier can be trained to further process the image and try to predict the type of obstacle seen. Furthermore, the robot can be commanded to take pictures from different angles to reveal more information about the obstacle. Object detection systems like YOLO [26] can also be worth looking into. For all of this to work it is important that the robot is capable of taking photos with the obstacle well within the frame. The method should be implemented such that it ensures that nothing stands between the robot and the obstacle when the image is captured. Currently, the system does not take this into account. Improvements can also be made to the method were the robot navigates around the obstacle to collect additional LiDAR data. Controls should be made to ensure that collisions are avoided instead of blindly following a pre-defined circular path. However, there is probably more to gain by utilizing the camera than the LiDAR when it comes to further inspection of obstacles. The reason for this is that the accuracy of the LiDAR, as experienced in the testing, is too low to extract information about the shape of small obstacles.

The inspection route implemented in this project has some weaknesses. If an obstacle is blocking one of the waypoints, the method will fail as it needs to visit a waypoint before it heads towards the next. To better organize the inspection route, one could utilize the SMACH package [4] in ROS. SMACH is a "task-level architecture for rapidly creating complex robot behavior". With this, one can define what to do when the robot is not able to reach a waypoint, has completed the route and more. For example, the robot can be automatically ordered to take photos of all found obstacles after a successful inspection route. If a robot with a manipulator is used, many more applications open up. For instance, the robot

can be ordered to remove obstacles. As the complexity increases, alternatives like SMACH can be worth looking into.

Chapter 8.

Navigational Capabilities

This chapter presents tests that were designed to challenge the robot's ability to navigate the environment. These tests specifically challenge some of the key issues noted during the experiments in the two previous chapters. What can cause collisions? Is the robot able to navigate narrow sections?

8.1. Experiment Setup

In addition to observing the behaviour of the robot during the obstacle detection, a set of more special cases were tested. Objects were placed between checkpoint one and two of Figure 7.6 and the robot was ordered to navigate from checkpoint one to checkpoint two. The following objects were used, one at a time.

1. Three cylinders of height 11.5 cm and a 5.5 cm diameter
2. A structure featuring an overhang at a height of 15.5 cm
3. A chair with 2 cm diameter legs

The cylinders were used to demonstrate potential problems caused by the LiDAR only covering a single plane of about 13 cm of the ground. An overhang was set up to control that the robot can move underneath objects. Lastly, the chair was chosen because of its thin legs and should test the robot's ability to sense thin objects and avoid collisions.

Because probabilistic algorithms are controlling the navigation of the robot, one

can experience that the robot behaves differently given the challenge to navigate the same section with identical setup of obstacles. To affirm this, the robot was once again requested to move from checkpoint one to checkpoint two of Figure 7.6. This time, the 0.31 m by 0.4 m cardboard box was placed exactly between the two checkpoints. The test was performed four times and the path of the robot was recorded by reading of the robot's estimated position at a rate of 2 Hz.

Lastly, two more tests were performed with the path recording turned on. These were two full inspection routes with obstacles. The first one was the regular route, shown in Figure 7.6, and the second one was a modification of the first one where the robot also had to pass the narrow section in the bottom left of the map.

8.2. Experiment Results

The setups for testing the robot's ability to cope with low objects, overhangs and thin objects are shown in Figure 8.1. As seen in Figure 8.1a, the robot failed to avoid collision with the 11.5 cm tall cylinders. The data from the LiDAR shows that it was not able to detect the cylinders at all. The next challenge, the overhang of 15.5 cm, was easily passed. Once again, the LiDAR was not picking up the overhang and the route between the two checkpoints was planned as if the overhang was not there. The LiDAR's ability to detect thin objects was tested in the setup shown in Figure 8.1c. A promising behaviour was seen where the LiDAR detected the 2 cm diameter legs of the chair and the robot successfully navigated around the chair.

For the demonstration of the stochastic behaviour, recorded paths from checkpoint one to checkpoint two are shown in Figure 8.2. Note that the obstacle was not moved between the tests and that the initial pose of the robot was identical for all four runs.

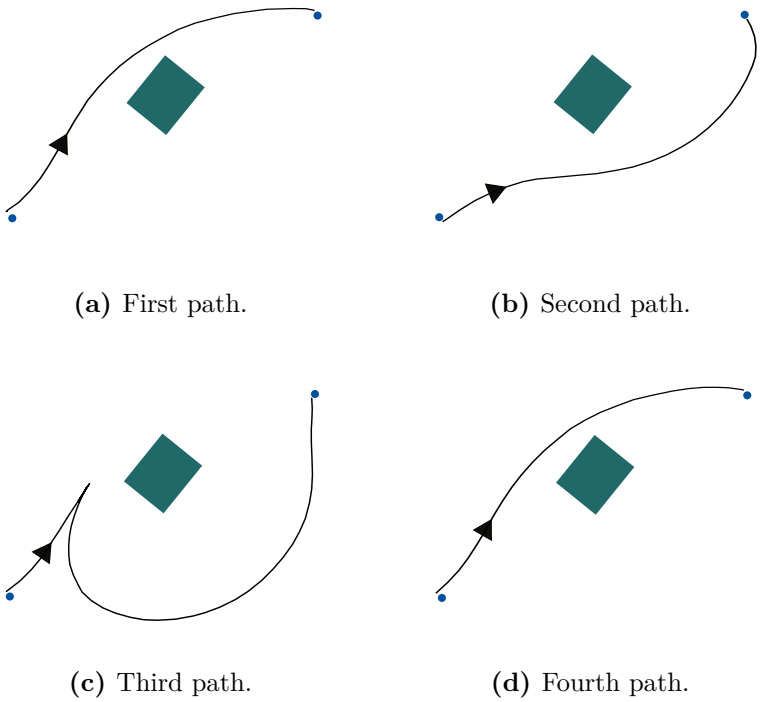
Furthermore, the behaviour of the robot was noted during the execution of the 10 obstacle detection experiments. For the most part, the robot seemed to maneuver efficiently. The problems mainly aroused when the robot was forced to navigate in narrow areas. During a similar setup of obstacles as in Figure 8.4, the same behaviour was seen where the robot moved back and forth several times before it was able to pass the obstacle. Additionally, in a test where one of the obstacles was placed close to a checkpoint, the robot ended up so close to the obstacle that it was not able to move away from it and froze before it had finished the inspection route.

(a) Collision with low obstacle.

(b) Passing under overhang.

(c) Navigating around thin objects.

Figure 8.1.: Three navigation challenges.



(a) First path. **(b)** Second path.

(c) Third path. **(d)** Fourth path.

Figure 8.2.: Recorded paths of the robot between the same checkpoints. Obstacle setup is identical for all the figures.

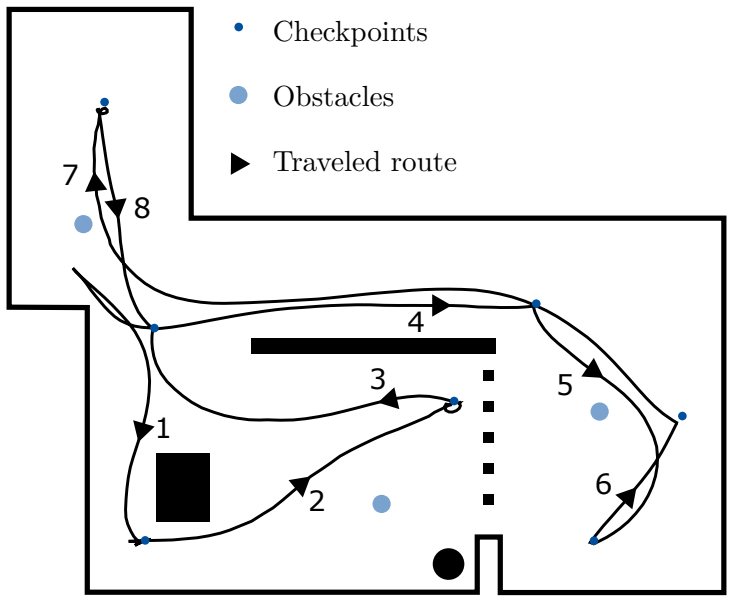


Figure 8.3.: Recorded path of the robot during an alternative inspection route.

Recorded paths of two full inspections routes are presented in the following. Figure 8.3 shows the robot navigating in a narrow area on the way to checkpoint one and two. The robot has a size of approximately 0.28 m × 0.31 m and the space between the wall and the black box is 0.6 m. However, note that the robot does not drive directly towards the first checkpoint in the beginning. Further, one can note how the robot plans and executes a smooth path around the obstacle on the way to checkpoint five. In Figure 8.4, the path to checkpoint three is the most interesting part. The robot seems to have trouble passing the obstacle. For both figures, one should take into account that the path is extracted from the robot's estimated position. Therefore, the paths are not a hundred percent accurate.

8.3. Simulation Setup

Early in testing it was found that the robot had problems passing through narrow sections. The problem seemed to arise when the passage got less 0.7 m wide. Then, the robot would hesitate to navigate through the section. It was suspected that the cause of the hesitation was uncertainties in the pose estimation because planning a collision-free route is harder when the current pose is highly uncertain. To challenge this hypothesis a simulation was set up were the robot had two different modes. One mode where the LiDAR accuracy was increased and one

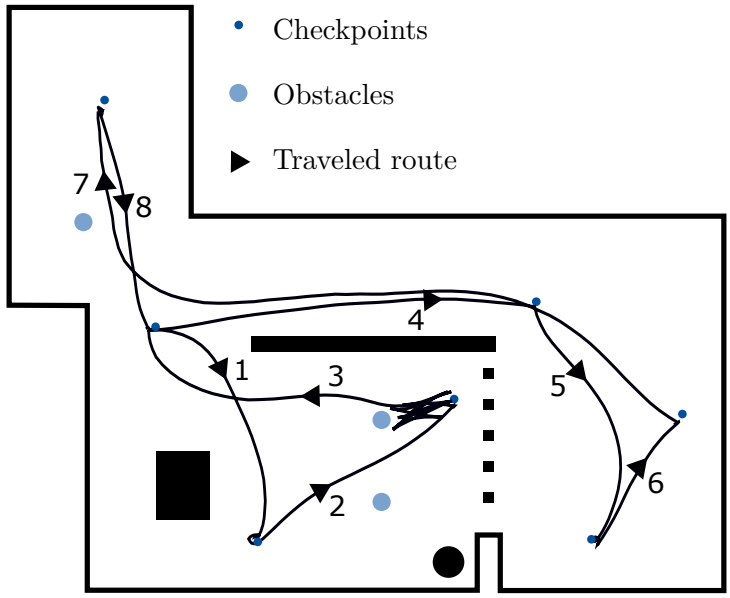


Figure 8.4.: Recorded path of the robot during an inspection route.

	High accuracy	Standard accuracy
delta [m]	0.02	0.05
resolution [m]	0.005	0.015
stdev [m]	0.004	0.01

Table 8.1.: Parameter setting in the simulation.

mode where the LiDAR had a similar accuracy as the real TurtleBot3. The parameter settings in the two modes can be seen in Table 8.1. "delta" is the grid size of the occupancy grid map. "resolution" is the resolution of the LiDAR measurement. "stdev" is the standard deviation of the zero-mean Gaussian noise added to the LiDAR measurement to simulate inaccuracies. All of the parameters were set to about a third of the original value to simulate a higher quality LiDAR than the LDS-01 which comes shipped with the TurtleBot3.

To test if a better performing LiDAR would increase the ability of the robot to navigate narrow sections, a 3D world was modelled. The shape of the world can be seen in Figure 8.5. Two versions of the world were used, one where the distance D of Figure 8.5 was set to 0.6 m and one where it was set to 0.9 m. The worlds were mapped with SLAM using both the standard accuracy and the high accuracy simulation. A total of 20 tests were performed for both worlds. 10 with the robot in standard accuracy mode and 10 with the robot in high accuracy mode. For

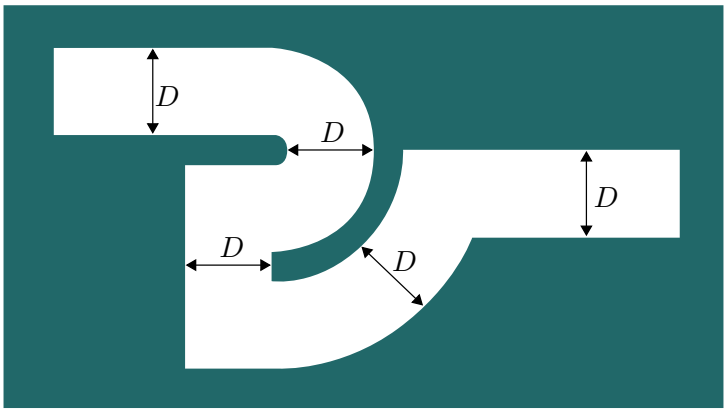


Figure 8.5.: Illustration of the passage with corridor width D.

each test, the robot was ordered to navigate from the left end to the right end of the corridor. The time usage from start to goal was noted for every test. If the robot was not able to reach the goal before it got stuck, the time spent before it got stuck was recorded. Also, it was noted where in the world the robot hesitated or got stuck.

8.4. Simulation Results

Results from the maneuverability simulation in the narrow corridor are found in the appendix, section A.2. With standard settings, the robot was not able to navigate the 0.6 m wide path any of the ten times. On average, it got stuck after 36.8 s. Typical areas where the robot got stuck can be seen in Figure 8.6. In these first 10 tests, especially area number two seemed to be problematic for the robot.

For the robot in standard settings, but in the 0.9 m wide world, the performance improved. Only once did the robot get stuck, this was at point one in Figure 8.6. The nine times the robot completed the path, the average time usage was 46.5 s. However, the variation was substantial with times ranging from 35.4 s to 104.1 s. In the 104.1 s run, the robot had problems both in area one and two as it tried to cut the corners, nearly got stuck and had to back out.

With the LiDAR accuracy increased to 0.004 m, the goal was reached six out of the ten times in the 0.6 m wide world. Average time spent to reach the goal was 64.2 s. In general, the robot hesitated much to navigate the narrow corridor, especially at point four. The robot needed to back out from this corner four out

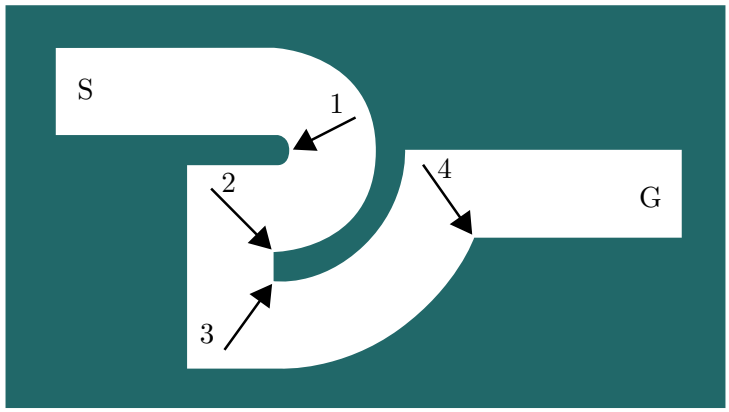


Figure 8.6.: Critical points of the narrow corridor.

of the six times it was able to complete the route.

The simulated robot completed the path ten out of ten times with high accuracy LiDAR in the 0.9 m wide corridor. On average it took 41.0 s to complete the path. In one run the robot hesitated in area one and in another it hesitated in area two. Other than this, the robot completed the route without problems.

8.5. Discussion

The first set of tests demonstrate some major issues related to the specific LiDAR used in the experiments. This LiDAR will only let the robot perceive the environment at a plane 13 cm of the ground. As experienced in the testing, objects lower than this are invisible to the robot and it will not be able to map these low objects, nor avoid collisions with them. For a robot to be useful on an O&G platform it will need to have a more sophisticated LiDAR with the ability to emit and detect laser rays at an angle, not only in a single plane. If not, the whole platform would have to be designed with this in mind. For the overhang and the thin objects, the results were satisfying. In general, the collision avoidance implemented in `move_base` coupled with the LDS-01 LiDAR is impressive and even objects as thin as 2 cm are easily avoided.

Figure 8.2 demonstrates the stochastic behaviour of the system. This does not necessarily need to be a major concern as the robot is, in general, good at finding a way from A to B as long as there exists some path with sufficient space. Even if this means going the long way around. Of the four figures, Figure 8.2c is

particularly interesting. One can see how the robot gets a little too close to the obstacle and has to back off to avoid a collision. Typically, the robot is risk-averse and will try to avoid collisions to the greatest extent. This means planning a route with a sufficient clearance to any walls. It is possible to adjust this balance by changing the parameters `lethal_cost` and `neutral_cost` in the Navigation stack of ROS. They relate to the expected immediate payoff, $r(\boldsymbol{x}_i, \boldsymbol{u})$, of equation (3.14). `lethal_cost` is the payoff of going to a cell close to an obstacle, while `neutral_cost` is the payoff of going to a cell far from any obstacle and with no risk of collisions. If faster navigation is wanted, one can experiment with bringing the values of the two parameters closer. This was not tried in the experiments, but should make the robot less risk-averse. Naturally, the downside is an increased risk of the robot getting stuck in narrow sections of the map. Also, the parameters of the DWA local path planner have an influence on how much risk the robot will take. Setting high values for α and γ relative to β (of the objective function (3.16)) should increase the speed of the robot at the cost of potentially more collisions.

The two full inspection routes shown in Figure 8.3 and Figure 8.4 show some of the same signs as what was just discussed. The robot is able to plan and execute a path from A to B as long as there is sufficient space. Note how the robot does not drive directly towards the first waypoint in Figure 8.3, but moves backwards. This is likely because the robot's uncertainty in own pose is quite high when the localization has just been initialized. Therefore, it needs to move around and receive new measurements to decrease the uncertainty before it tries navigating the narrow area around the first waypoint. In Figure 8.4, the most interesting part is the navigation towards waypoint three. At this point it is narrow on both sides of the obstacle that the robot needs to pass. A reasonable explanation of the behaviour is that the robot tries to pass the obstacle on one side, gets too close to the obstacle, plans a route around the obstacle on the other side, but ends up facing the same problem. Behaviour like this is of course unwanted and is the main concern related to the robot's navigational capabilities after having tested it for around fifteen hours.

In the 0.9 m wide corridor, the simulation results were as expected. The robot was on average 13 percent faster with the most accurate LiDAR settings. This is not a significant difference and as speed is not the most important feature of an inspection robot, the LiDAR accuracy did not seem to affect the performance to any major extent. It should be noted that the robot got stuck one out of ten times when running with the standard LiDAR settings, but there is not enough data here to conclude on whether this was because of the LiDAR or just a coincidence. When we look at the results in the 0.6 m wide corridor there is a larger difference between the standard LiDAR and the high performance one. Six out of ten successful runs versus zero out of ten indicates that a higher quality LiDAR could be beneficial for

navigation in narrow areas. Unfortunately, it is hard to find specifications on the short-range performance of popular LiDARs like the Velodyne Puck. Because the industry is largely being driven by self-driving car applications, the short-range accuracy is not stated. If no sufficiently accurate LiDARs exist, it is possible to equip the robot with an ultrasonic sensor. Cheap alternatives like the HC-SR04 are reported to have a resolution of 3 mm [11] and can be useful for navigation in narrow areas.

To sum up, the robot shows good results when navigating in corridors with a width of no less than 0.9 m. It must be noted that the quantitative results discussed here are from a simulation and not from the actual TurtleBot, but they comply with what was seen during the experiments. Why the robot had a tendency to get stuck in the areas indicated in Figure 8.6 is unclear. Different combinations of parameters in the local planner need to be tested to answer this. Bengel [1] concludes that corridors on an offshore platform should have a width of no less than 0.7 m, based on standards for transportation of an injured person. If the robot needs to navigate corridors with a width between 0.7 m and 0.9 m, a higher performing LiDAR can be beneficial. Alternatively, in combination with other range sensors, like ultrasonic sensors. Also, if a collision once in a while is acceptable, more aggressive settings of the parameters in the local planner can be beneficial, even though this was not tried during this project.

8.6. Further Work

As discussed, one of the main challenges revealed during the testing was for the robot to navigate tight spaces. A first effort to better this should be to experiment with different parameter settings for the local planner. If this is not successful, one could experiment with a higher quality LiDAR or some other range-sensor. Even physical switches like the ones found on the bumper of a vacuum cleaner robot can be used if the robot is allowed to come into contact with walls and other structures on the platform.

One last feature that has not been discussed in this thesis is a recovery mode. What happens if the robot loses the network connection? The robot should probably be able to navigate back to a docking station in case this happens. Recovery of a robot that has gotten stuck or has run out of battery should also be looked into.

Chapter 9.

Concluding Remarks

In this thesis, possible use-cases of a mobile robot on an unmanned offshore production platform have been investigated. With a TurtleBot3 Waffle Pi and the navigational capabilities of the ROS Navigation stack, new features for a mobile robot have been developed with ROS. The new features are controlled from the web browser and demonstrate how an onshore human operator can have control over important events on the platform like gas leakages and unwanted objects lying on the platform.

Methods concerned with gas leakages discussed in the thesis include the generation of a heatmap and the detection of leakages with an infrared camera. Both methods were tested, but only with a simulated gas sensor and a RGB-camera treating the colour blue as a low temperature. If applied on an offshore platform, the mobile robot can work alongside a network of mounted gas detectors to increase the reliability of the gas detection system and thus decrease the risk of explosions.

While the detection of gas leakages has been developed by combining already known methods, there is more to take away from the obstacle detection. The thesis contributes with a method for detecting obstacles using a mobile robot equipped with a LiDAR and a map of the environment. Experiments show that the method is able to estimate the centroid of obstacles with an accuracy of about 0.2 m. The developed software plots the estimates on the map and can help humans to quickly get an overview of the status of the platform. Additionally, it opens up for other exciting opportunities. An extension that was developed in this project was to automatically photograph the obstacle and display it to the user. Most definitely there are more applications of this feature left to be discovered by inventive people.

None of the methods mentioned in this thesis work if the robot is not able move around on the platform. The thesis demonstrates the excellent performance of the ROS Navigation stack in combination with a LiDAR, the only exception being navigation in areas narrower than about 0.9 m. Tests indicate that this can probably be improved by using more accurate range sensors. The theory also suggests that adjusting parameters for the local path planning should help. Without having tested the setup on an actual offshore platform, it is not possible to draw a definite conclusion, but the testing suggests that the ROS Navigation stack can be used on a platform to plan and execute navigation from A to B given a robot with a design suited for this environment.

Hopefully, an inspection robot will one day be an important piece in ensuring the safe operation of an unmanned O&G platform. Unexpected events will happen on the platform and a general purpose inspection robot can help keeping the platform as independent of human intervention as possible.

References

[1] M Bengel, K Pfeiffer, B Graf, A Bubeck, and A Verl. "Mobile robots for offshore inspection and manipulation". eng. In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2009, pp. 3317–3322. ISBN: 9781424438037.

[2] Arne Ulrik Bindingsbo, Charlotte Skourup, and John Pretlove. "The robotized field operator.(oil and gas companies of the future)(Process Automation)". English. In: *InTech* 56.10 (2009). ISSN: 0192-303X.

[3] Kumar Bipin. *Robot Operating System cookbook : over 70 recipes to help you master advanced ROS concepts*. eng. Birmingham, UK, 2018.

[4] J. Bohren. *smach*. 2019. URL: http://wiki.ros.org/smach (visited on 06/04/2019).

[5] Guilherme P.S. Carvalho, Marco F.S. Xaud, Ighor Marcovistz, Alex F. Neves, and Ramon R. Costa. "The DORIS Offshore Robot: Recent Developments and Real-World Demonstration Results". eng. In: *IFAC PapersOnLine* 50.1 (2017), pp. 11215–11220. ISSN: 2405-8963.

[6] Naim Dahnoun. "Hough transform". eng. In: *Multicore DSP*. Chichester, UK: John Wiley & Sons, Ltd, 2017, pp. 591–603. ISBN: 9781119003823.

[7] P Desai, H E Sevil, A Dogan, and B Huff. "Construction of an obstacle map and its realtime implementation on an Unmanned Ground Vehicle". eng. In: *2011 IEEE Conference on Technologies for Practical Robot Applications*. IEEE, 2011, pp. 139–144. ISBN: 9781612844824.

[8] A Dogan. "Probabilistic approach in path planning for UAVs". eng. In: *Proceedings of the 2003 IEEE International Symposium on Intelligent Control*. IEEE, 2003, pp. 608–613. ISBN: 0780378911.

[9] D.H. Douglas and T.K. Peucker. "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature". In: *Classics in Cartography: Reflections on Influential Articles from Cartographica*. wiley, 2011, pp. 15–28. ISBN: 9780470669488.

[10] Boston Dynamics. *SpotMini*. 2019. URL: https://www.bostondynamics.com/spot (visited on 04/26/2019).

[11] Elecrow. *Ultrasonic ranging module : HC-SR04*. 2019. URL: https://www.elecrow.com/download/HC_SR04%5C%20Datasheet.pdf (visited on 06/03/2019).

[12] Open Source Robotics Foundation. *Documentation - ROS Wiki*. 2018. URL: http://wiki.ros.org/ (visited on 03/20/2019).

[13] D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". eng. In: *Robotics & Automation Magazine, IEEE* 4.1 (1997), pp. 23–33. ISSN: 1070-9932.

[14] B Gerkey. *amcl - ROS Wiki*. 2018. URL: http://wiki.ros.org/amcl?distro=melodic (visited on 03/07/2019).

[15] Rafael C Gonzalez. *Digital image processing using MATLAB*. eng. S.l., 2009.

[16] N Gordon, D Salmond, and A Smith. "Novel approach to nonlinear/non-Gaussian Bayesian state estimation." eng. In: *IEE Proceedings F. Radar and Signal Processing* 140.2 (1993), P.107–P.113. ISSN: 0956-375X. URL: http://search.proquest.com/docview/26116978/.

[17] G Grisetti, C Stachniss, and W Burgard. "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters". eng. In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46. ISSN: 1552-3098.

[18] Fredrik Gustafsson. *Statistical sensor fusion*. eng. Lund, 2012.

[19] Kenneth Joy. *Bresenham's Algorithm*. 1999. URL: http://graphics.idav.ucdavis.edu/education/GraphicsNotes/Bresenhams-Algorithm.pdf (visited on 04/02/2019).

[20] Sherman Karp and Larry B. Stotts. "Light detection and ranging". In: *Fundamentals of Electro-Optic Systems Design*. Cambridge: Cambridge University Press, 2012, pp. 151–178. ISBN: 9781139108850.

[21] Eitan Marder-Eppstein. *base_local_planner*. 2018. URL: http://wiki.ros.org/base_local_planner?distro=kinetic (visited on 02/06/2019).

[22] "Mobile Robots". In: *Robotics: Modelling, Planning and Control*. London: Springer London, 2009, pp. 469–521. ISBN: 978-1-84628-642-1. DOI: 10.1007/978-1-84628-642-1_11. URL: https://doi.org/10.1007/978-1-84628-642-1_11.

[23] Asmus Nielsen. *Unmanned wellhead platforms – UWHP summary report*. eng. 2016.

[24] Nvidia. *GEFORCE® GTX 1080 Ti*. 2018. URL: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/ (visited on 02/16/2019).

[25] Oceaneering. *Remotely Operated Vehicles (ROVs)*. 2019. URL: https://www.oceaneering.com/rov-services/ (visited on 02/13/2019).

[26] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: (2018).

[27] RoboSavvy. *Turtlebot3 Waffle Pi*. 2017. URL: https://robosavvy.com/store/turtlebot3-waffle-pi.html (visited on 03/19/2019).

[28] Robotis. *TurtleBot3*. 2019. URL: http://emanual.robotis.com/docs/en/platform/turtlebot3/autonomous_driving/ (visited on 03/21/2019).

[29] Robotis. *Turtlebot3*. 2019. URL: http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (visited on 03/19/2019).

[30] Stuart J. (Stuart Jonathan) Russell. *Artificial intelligence : a modern approach*. eng. Boston, 2016.

[31] Erich Schubert, Jörg Sander, Martin Ester, Hans Kriegel, and Xiaowei Xu. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". eng. In: *ACM Transactions on Database Systems (TODS)* 42.3 (2017), pp. 1–21. ISSN: 1557-4644.

[32] Frank Shaffer, Ömer Savaş, Kenneth Lee, and Giorgio de Vera. "Determining the discharge rate from a submerged oil leak jet using ROV video". eng. In: *Flow Measurement and Instrumentation* 43 (2015), pp. 34–46. ISSN: 0955-5986.

[33] Dan Simon. *Optimal state estimation : Kalman, H [infinity] and nonlinear approaches*. eng. Hoboken, N.J., 2006.

[34] Jpt Staff. "Sensabot: A Safe and Cost-Effective Inspection Solution". eng. In: 64.10 (2012), pp. 32–34. ISSN: 0149-2136. URL: https://www.onepetro.org/journal-paper/SPE-1012-0032-JPT.

[35] Satoshi Suzuki and Keiichia Be. "Topological structural analysis of digitized binary images by border following". eng. In: *Computer Vision, Graphics and Image Processing* 30.1 (1985), pp. 32–46. ISSN: 0734-189X.

[36] Taurob. *UGV - taurob tracker*. 2019. URL: http://taurob.com/produkte/ugv-taurob-tracker/ (visited on 02/14/2019).

[37] Sebastian Thrun. *Probabilistic robotics*. eng. Cambridge, Mass, 2005.

[38] Aksel A. Transeth, Oystein Skotheim, Henrik Schumann-Olsen, Gorm Johansen, Jens Thielemann, and Erik Kyrkjebo. "A robotic concept for remote maintenance operations: A robust 3D object detection and pose estimation method and a novel robot tool". eng. In: IEEE Publishing, 2010, pp. 5099–5106. ISBN: 9781424466740.

[39] Y. Zhang, J. Wang, X. Wang, and J.M. Dolan. “Road-segmentation-based curb detection method for self-driving via a 3D-LiDAR sensor”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.12 (2018), pp. 3981–3991. ISSN: 15249050.

Appendix A.

Test Results

A.1. Obstacle Detection Experiment Results

The results from the obstacle detection experiment is given in Table A.1. A blank space in the estimated position means that the obstacle was not detected. Test nr. 8 was performed without any obstacles in the environment.

A.2. Maneuverability Simulation Results

Table A.2 to Table A.5 show the results for the navigation of the robot from start (S) to goal (G) in a simulation of the map in Figure 8.6. "Time usage" is measured in seconds from the navigation goal is sent to the robot successfully reaches the goal. If the robot is stuck before it reaches the goal, the time usage refers to the time it spent from receiving the goal to it got stuck. "Hesitate" indicates that the robot got close to the wall at the corresponding point, had to back up, but was able to carry on. "Stuck" indicates that the robot got too close to the wall and was not able to recover from the position. The critical points can be seen in Figure 8.6. The stated width of the world is the distance D of Figure 8.5.

Test nr.	Obstacle type	True pos.		Est. pos.	
		x	y	x	y
1	Large box	2.80	-1.40	2.70	-1.45
2	Cylinder	5.20	0.10	5.00	-0.10
	Large box	-0.90	2.00	-0.80	1.90
3	Small box	2.00	-2.00	1.95	-2.15
	Cylinder	4.00	-1.00		
	Large box	-0.60	1.20	-0.45	1.10
4	Small box	-0.60	1.00	-0.40	0.9
5	Large box	5.00	-2.00	4.80	-2.05
	Cylinder	2.00	-1.00	2.00	-1.20
	Small box	2.00	-2.00	1.90	-2.10
6	Cylinder	-0.60	1.00	-0.50	1.00
	Small box	-0.60	1.20		
7	Small box	-0.90	2.00	-0.70	1.90
	Large box	5.20	0.10	5.05	-0.10
8					
9	Cylinder	2.00	-1.00	1.95	-1.15
	Small box	2.80	-1.40	2.65	-1.40
10	Large box	-0.90	2.00	-0.85	1.95
	Cylinder	5.00	-2.00	4.80	-2.10
	Small box	4.00	-1.00	3.75	-1.05

Table A.1.: Obstacle detection test results.

Test nr.	Completed	Time usage	Hesitate	Stuck
1	No	36.7		2
2	No	35.0		2
3	No	24.4		1
4	No	29.6		2
5	No	33.7		3
6	No	30.0		2
7	No	30.0		2
8	No	43.3		2
9	No	32.5		3
10	No	72.8		2

Table A.2.: Results of simulation with standard precision in the 0.6 m wide world.

Test nr.	Completed	Time usage	Hesitate	Stuck
1	No	25.4		1
2	Yes	39.5	2	
3	Yes	104.1	1, 2	
4	Yes	40.2	2	
5	Yes	35.6		
6	Yes	35.5		
7	Yes	56.4	2	
8	Yes	36.0	2	
9	Yes	35.4		
10	Yes	36.2		

Table A.3.: Results of simulation with standard precision in the 0.9 m wide world.

Test nr.	Completed	Time usage	Hesitate	Stuck
1	Yes	64.0	2	
2	No	113.0		3
3	Yes	38.8	4	
4	No	41.5		1
5	Yes	86.5	4	
6	No	32.1		2
7	Yes	59.6	2, 3	
8	Yes	44.2	4	
9	No	167.2		2
10	Yes	91.9	1, 4	

Table A.4.: Results of simulation with increased precision in the 0.6 m wide world.

Test nr.	Completed	Time usage	Hesitate	Stuck
1	Yes	36.4		
2	Yes	36.1		
3	Yes	35.7		
4	Yes	35.5		
5	Yes	35.1		
6	Yes	35.5		
7	Yes	35.3		
8	Yes	35.4		
9	Yes	62.8	1	
10	Yes	61.9	2	

Table A.5.: Results of simulation with increased precision in the 0.9 m wide world.

Appendix B.

Additional Theory

B.1. Vectors

When tracking an object, a representation of the object's position is needed. The position can be described by a vector in a coordinate frame. A three-dimensional Cartesian coordinate frame is defined by three orthogonal unit vectors, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$ and $\boldsymbol{e}_3$. As a linear combination of the orthogonal unit vectors, all possible positions can be described, $\boldsymbol{v} = v_1\boldsymbol{e}_1 + v_2\boldsymbol{e}_2 + v_3\boldsymbol{e}_3$, Figure B.1.

Further, given a coordinate system, a position can be represented by a column vector,

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}.$$

B.2. Coordinate transformations

In many applications, it is necessary to be able to know how to transform a vector from one coordinate frame to another. As an example, we could think of the interaction between a camera and an industrial manipulator. If the camera has determined the position of an object that the manipulator should grab, the vector in the frame of the camera should be transformed to the frame of the manipulator. This transformation will be a function of the difference in position and orientation of the coordinate frames of the camera and the manipulator. The

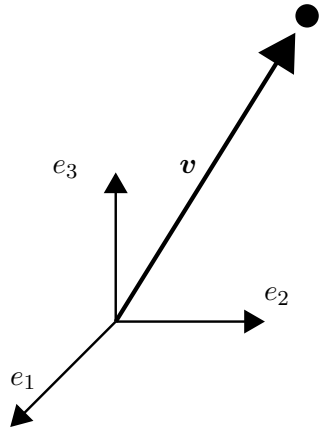


Figure B.1.: A vector in a Cartesian coordinate system.

same position can be described in two different coordinate frames, a and b, as

$$\boldsymbol{v} = v_1^a \boldsymbol{a}_1 + v_2^a \boldsymbol{a}_2 + v_3^a \boldsymbol{a}_3$$

and

$$\boldsymbol{v} = v_1^b \boldsymbol{b}_1 + v_2^b \boldsymbol{b}_2 + v_3^b \boldsymbol{b}_3.$$

If we use the column vector description, we must specify if the coordinated are expressed in frame a or b, $\boldsymbol{v}^a$ or $\boldsymbol{v}^b$.

B.3. Rotation matrices

A three-dimensional vector can be transformed from a coordinate frame to another coordinate frame with a different orientation by premultiplying it with a matrix,

$$\boldsymbol{v}^a = R_b^a \boldsymbol{v}^b,$$

where R_b^a is called a rotation matrix. This particular one is a coordinate transformation matrix from frame b to a. An example is illustrated in Figure B.2.

The transformation matrices for simple rotations of an angle about one of the axes of the coordinate frame are

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$

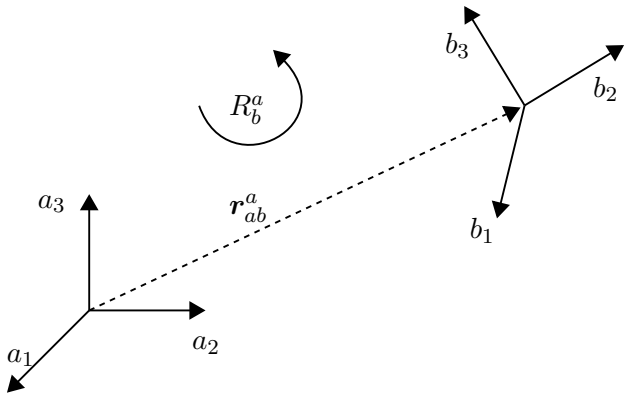


Figure B.2.: A coordinate transformation from frame b to a.

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If frame b is rotated $\frac{\pi}{4}$ around the z axis relative to frame a, the column vector in frame a can be found as $\boldsymbol{v}^a = R_z(\frac{\pi}{4})\boldsymbol{v}^b$. This could also be seen as rotating a point by $\frac{\pi}{4}$ inside the same coordinate system.

More complex rotations can be achieved by chaining rotations about the x-, y- and z-axes. A rotation matrix for a rotation about the x axis by ϕ, then around the y axis by θ and lastly around the z axis by ψ can be found as $R = R_z(\psi)R_y(\theta)R_x(\phi)$.

B.4. Homogeneous transformation matrices

Sometimes not only rotation from a coordinate frame to another is desirable, but also a translation. This is achieved through the concept of the homogeneous transformation matrix.

$$T_b^a = \begin{bmatrix} R_b^a & \boldsymbol{r}_{ab}^a \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Here, $\boldsymbol{r}_{ab}^a$ is the vector describing the position of the origin of frame b relative to the origin of frame a. R_b^a is the same rotation matrix as already seen. Homogeneous transformation matrices can be combined in the same way as rotation matrices.

T_b^a will be a 4×4 matrix and we need a four-dimensional vector to multiply with it. This is simply obtained by adding an entry of 1 at the end of the original vector,

$$\tilde{\boldsymbol{v}} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 1 \end{bmatrix}.$$

B.5. Bayesian filtering

Bayesian filtering uses a model of how the system evolves over time and incoming measurements to estimate an unknown probability density function, pdf. How the system evolves over time is described by the system equation,

$$\boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_{k+1}, \boldsymbol{w}_k). \tag{B.1}$$

The other equation needed is the measurement equation and it should describe how like it is to observe a measurement,

$$\boldsymbol{z}_k = h_k(\boldsymbol{x}_k, \boldsymbol{v}_k). \tag{B.2}$$

$\boldsymbol{x}$ is the state of the system, $\boldsymbol{z}$ is the measurement and $\boldsymbol{u}$ is the action chosen. The process noise $\boldsymbol{w}_k$ is assumed to be a white noise sequence with known pdf [33]. The same holds for the measurement noise $\boldsymbol{v}_k$. k describes which discrete time step we are at. Equation (B.1) will let us calculate $P(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k, \boldsymbol{u}_{k+1})$, how the state of the system changes over time. Equation (B.2) will let us calculate $P(\boldsymbol{z}_k|\boldsymbol{x}_k)$, how likely we are to observe a measurement given a state of the system.

The a priori distribution, $P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})$, is our best guess of the system's new state before receiving a new measurement. As described by Gustafsson [18], the goal of the Bayesian filtering is to calculate the a posterior distribution, $P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k+1})$, our updated belief of the system's new state after receiving a new measurement. For doing this, we utilize Bayes' theorem,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(A,B)}{P(B)}. \tag{B.3}$$

We have,

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k+1}) = P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}), \tag{B.4}$$

which using equation (B.3) gives us

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}) = \frac{P(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1})}{P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1})}. \tag{B.5}$$

Also, using Bayes' theorem we have

$$P(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}) = P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})P(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}), \tag{B.6}$$

and

$$P(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}) = P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}). \tag{B.7}$$

Inserting (B.7) into (B.6) and the result into (B.5), we obtain

$$\begin{aligned} &P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}) \\ &\quad = \frac{P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})}{P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1})}. \end{aligned} \tag{B.8}$$

The Markov sensor assumption,

$$P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}, \boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}) = P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1}), \tag{B.9}$$

tells us that the current measurement only depends on the current state, not the prior measurements. Equation (B.9) simplifies (B.8) into

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}) = \frac{P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1})P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})}{P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1})}. \tag{B.10}$$

Again, using Bayes' theorem we have,

$$P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}) = P(\boldsymbol{z}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})P(\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}).$$

Inserting this into (B.10) and canceling out the terms, we get

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1}) = \frac{P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1})P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})}{P(\boldsymbol{z}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})}. \tag{B.11}$$

Equation (B.11) is the key equation and allows us to recursively update the belief, $P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})$ when we obtain a new measurement $\boldsymbol{z}_{k+1}$.

The denominator of (B.11) is simply a normalization term and is found with

$$P(\boldsymbol{z}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}) = \int P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1})P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})d\boldsymbol{x}_k, \tag{B.12}$$

an integral over the numerator of equation (B.11) for all state vectors. $P(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1})$ in the equation above is available from the measurement equation (B.2).

$P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})$ in (B.11) and (B.12) is found from

$$P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}) = \int P(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k, \boldsymbol{u}_{k+1})P(\boldsymbol{x}_k|\boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k})d\boldsymbol{x}_k. \tag{B.13}$$

An intuitive explanation of this is that the probability of transitioning into a new state is the probability of being in a prior state times the probability of a transition from this prior state to the new one, given the action, summed up for all possible prior states. $P(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k, \boldsymbol{u}_{k+1})$ in the above equation is available from the system equation (B.1).

Now, when we have a transition, we calculate $P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k})$ with (B.13). If we receive a new measurement, we use (B.11) to calculate $P(\boldsymbol{x}_{k+1}|\boldsymbol{u}_{1:k+1}, \boldsymbol{z}_{1:k}, \boldsymbol{z}_{k+1})$.

B.6. Hough Transform

The Hough transform is a technique for feature extraction in digital images. With the transform, one can "detect lines, circles or any structures with a known parametric equation" [6]. By a process known as "voting", every foreground pixel in the image will vote on the combination of parameters that agrees with the position of this pixel. The foreground pixels are the selected pixels in the image we use to search for the features. Typically, The Canny edge detector is used to find these pixels from the raw image.

Circles can be described by their center and radii. All the possible combinations of x and y that solve the following equation will be on the circumference of the circle with radius r and center in (a, b),

$$(x - a)^2 + (y - b)^2 = r^2. \tag{B.14}$$

In polar coordinates, we have

$$(x, y) = (a + r\cos\theta, b + r\sin\theta).$$

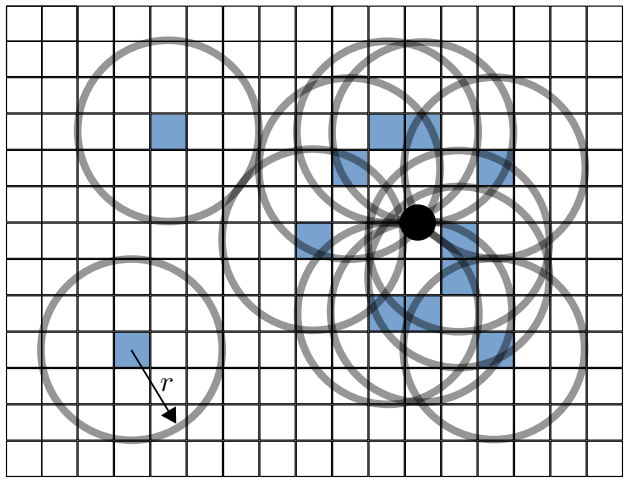


Figure B.3.: Voting process in the Circle Hough Transform (known radius).

Solving for a and b

$$(a, b) = (x - r\cos\theta, y - r\sin\theta).$$

Figure B.3 illustrates the voting process of the Hough transform for a circle with known radius. Every foreground pixel (blue) votes on the points along a circle with radius r around it. The area where there is the highest density of votes is crowned as the center of the circle we are searching for. In the figure, this center is illustrated with a black dot. Looking at the area around the black dot of Figure B.3, it can seem like the foreground pixels are indeed forming a circle here.

The votes are gathered in a discrete container known as the accumulator. For a circle with known radius, the parameters are a and b, and the accumulator will be a 2D matrix. If the radius is not known, the accumulator will be a 3D matrix and the foreground pixels will vote on (a, b, r)-combinations of parameters. Because the number of parameter combinations in the matrix is limited, the accumulator must be set up according to one's own needs. An example could be to have a from -5 to 5 in increments of 0.1. This would give 101 possible a-values. A vote will be assigned to the closest cell. The same quantization process must be performed on b and r.

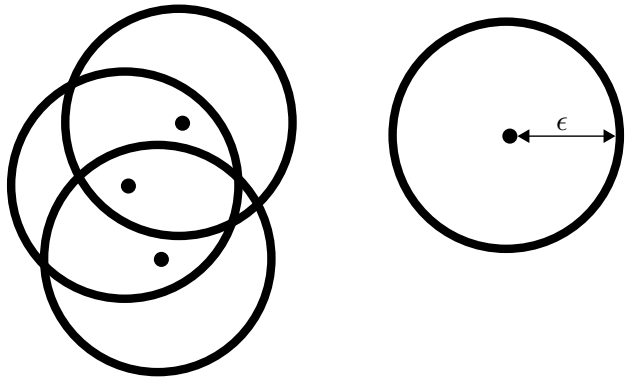


Figure B.4.: Three points in a cluster and an outlier.

B.7. DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [31] is an unsupervised machine learning algorithm. It is used to group data points into clusters. DBSCAN is versatile because it does not require the number of clusters to be pre-defined, clusters can have all sorts of shapes and it is robust to outliers.

The clustering algorithm considers a randomly chosen point and a neighbourhood. Neighbouring points are all points that are within a distance ϵ of the chosen point. Euclidean distance is typically used as the distance measure. If the number of neighbours is below some threshold `minPts`, the chosen point is considered as noise. If not, the point is considered as the first point in a new cluster. All the neighbours of the point are also assigned to this cluster. For every neighbour, if it has at least `minPts` neighbours, these neighbours are also considered as part of the same cluster. The process is repeated until no point in the cluster has any unvisited neighbours. Then, a new randomly, unvisited, point is chosen and the process is repeated from the beginning. Figure B.4 shows an example of a clustering where three points are assigned to the same cluster and the last point is classified as noise. Pseudocode and additional details can be found in the paper [31] by E. Schubert et al.

B.8. Ramer–Douglas–Peucker Algorithm

The Ramer-Douglas-Peucker Algorithm [9] is an algorithm for simplifying lines. Let a line be defined by a set of N points, $\boldsymbol{p}_i$, $i \in [0, N)$. The goal of the algorithm

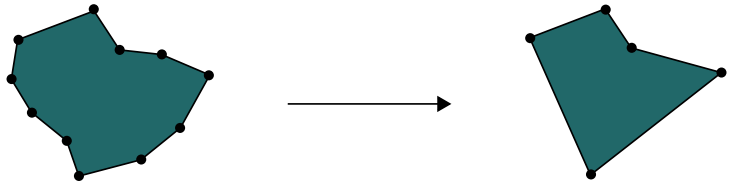


Figure B.5.: A simplification process of a boundary.

is to reduce the number of points needed to describe the "essence" of the line. As an example it can simplify a structure defined by its contour, see Figure B.5. If the original set of points were obtained from noisy measurements, the smaller set of points can provide a better and more general representation of the true structure. The algorithm works by considering a straight line between the first and the last point. Then, the distances from the line to all other points are calculated. The point furthest away from the line is selected. If this point is further than a threshold distance ϵ from the line, the algorithm is called recursively from the first point to this point and from this point to the last point. If closer than ϵ, the point is removed from the set and the algorithm call is terminated. A more detailed explanation can be found in *Classics in Cartography* [9].

Appendix C.

TurtleBot3 Waffle Pi Specifications

Item	Value
Maximum Translational Velocity	0.26 m/s
Maximum Rotational Velocity	1.82 rad/s
Maximum Payload	30 kgs
Size (L × W × H)	281 mm × 306 mm × 141 mm
Weight (+ SBC + Battery + Sensors)	1.8 kgs
Operating Time	About 2 hrs
Charging Time	About 2 hrs 30 min
Actuators	DYNAMIXEL XM430-W210-T
Single Board Computer	Raspberry Pi 3
Controller Board	OpenCR
Camera	Raspberry Pi Camera Module V2
LiDAR	LDS-01 360° angular range 1° angular resolution 120 mm to 3500 mm range ±15 mm accuracy up to 500 mm ±5.0 % accuracy over 500 mm
Additional Sensors	3-axis gyroscope 3-axis accelerometer 3-axis magnetometer

Table C.1.: Specification of the TurtleBot3 Waffle Pi [27]

Appendix D.

Eva Repository Description

All the code written for the purpose of this thesis can be found in the GitHub repository[1]. Also, instructions on how to run the code can be found there. All the most important files contain comments that explain how they work. To quickly sum up the different parts that make up the ROS package, the contents of the different folders will be briefly explained in the following.

action, msg and srv include the definition of the actionlib actions, the messages and the services.

css, js and pages include the files necessary CSS, JavaScript and HTML files for the web browser interface.

img includes image files used in the web browser interface and to store the images of Mode 4.

map includes the occupancy grid map used for navigation and Leaflet files.

models includes the CAD model of the room and the robot. The robot model is only loaded when not running a simulation. This is because the simulation includes the model from a TurtleBot3 package.

scripts include the Python files that run in the nodes.

worlds includes the setup of the Gazebo world and controls the import of models.

[1] https://github.com/krNesland/eva

Kristoffer Nesland

AGV on Unmanned Offshore Platform

NTNU
Kunnskap for en bedre verden