Nobleo Technology,
Eindhoven University of Technology

Internship Thesis

A progressive approach to dynamic obstacle avoidance using LiDAR based obstacle location prediction.

Author:
Josja Geijsberts

Supervisors:
Frank Sperling (Nobleo Technology)
René van de Molengraft (TU/e)

Student number:
0896965

CST reference number:
CST2019.042

June 21, 2019

Abstract

Dory is an autonomous version of the Ranmarine Technology WasteShark that Nobleo Technology is working on. Navigation needs to be improved in order to enable this floating drone to operate in busy waters. Layered costmaps using a standard sensor-based obstacle layer are commonly used to handle navigation for ROS based robots. This approach is conservative and does not allow for future representations of the environment to be used. Therefore, a progressive approach to dynamic obstacle avoidance using obstacle location prediction is developed during this thesis. The simulation results of this approach are very promising and show that it could act as an enhancement to the standard obstacle layer. The real life tests show that further improvements are required for this introduced method to be sufficiently robust to qualify as an implementation for a commercially available product. This thesis concludes with the next steps required for obstacle location prediction based navigation to enhance the performance of the WasteShark.

Keywords— ROS, Costmap Layers, Obstacle Location Prediction, WasteShark

Contents

Chapter 1

Introduction

Nobleo Technology is a branch of *Nobleo* that specializes in intelligent systems where controlling motion, practical engineering and expertise in algorithms for sensing and signal processing are the main pillars of development [1].

RanMarine Technology is a Dutch company that has developed a floating drone called the WasteShark. The purpose of this drone is to fare in harbours, rivers and canals, and collect waste that has been dropped into the water. The WasteShark developed by RanMarine Technology was missing one vital component; autonomy. That is where Nobleo Technology comes in [2].

Dory, the WasteShark Nobleo Technology is working on, is an autonomous version of the WasteShark equipped with, amongst other sensors, a LiDAR. This LiDAR is used during navigation to detect obstacles and thereby avoid collisions. These collisions are avoided by coming to a standstill and waiting for the obstacle to be removed or for a new command to be received. This can result in Dory being dormant for long periods of time. This issue also occurs when Dory is to operate in busy harbours and canals. The continuous movement of ships around Dory and her to be followed path, will result in safe movement to be increasingly complex. Another issue is that Dory can not come to a standstill instantly, so once a dynamic obstacle is detected, she might not be able to stop in time. An implementation of dynamic obstacle avoidance, combined with obstacle tracking and location prediction will solve these issues.

1.1 The proposed solution

This thesis introduces a new method for dynamic obstacle avoidance that uses obstacle tracking and location prediction to generate safe paths for Dory to follow. Predicting the location of dynamic obstacles allows for planning based on future representations of the environment and will results in Dory having sufficient time to respond to the situations that will arise. By providing Dory with more time to respond, collisions are less likely to occur. As the obstacle avoidance method developed during this thesis is made to be an addition to the existing framework, it should not interfere with the existing software. Another important requirement to be met is the cost of Dory. RanMarine Technology wants to keep the cost of the WasteShark low, to keep it a commercially attractive product. Therefore the method developed during this thesis should rely only on the existing hardware.

Based on the described goal and requirements, the following research question can be formulated:
What approach for dynamic obstacle avoidance based on the hardware available on Dory can decrease the amount of required replanning and increase the knowledge of future representations of the environment compared to standard methods whilst still meeting safety requirements?

1.2 Outline

Firstly, an in-depth investigation into the various methods available for robot navigation is described in Chapter 2. This chapter also provides insight into the framework the dynamic obstacle avoidance method developed during this thesis should fit into. Secondly, Chapter 3 elaborates upon the requirements to be met by the developed software. This includes a more detailed description of the action plan for the dynamic obstacle avoidance method as well as a description of how this software will be tested. The various platforms the software developed during this thesis will be tested on will also be described in this chapter. In Chapter 4, the developed obstacle avoidance method is described. Chapter 5 elaborates upon the results found with this method as well as provide a discussion on these results. Finally, Chapter 6 describes the conclusion and recommendation as well as various next steps that could result in interesting developments.

Chapter 2

Related work

This chapter provides background knowledge about Robot Operating System (ROS); the framework the dynamic obstacle avoidance method should fit into. It also describes the basics of robot navigation within ROS, such as global planners and costmap based navigation. This chapter also describes various existing dynamic obstacle avoidance methods, on which the method developed during this thesis is based or to which it will eventually be compared.

2.1 ROS and global planners

ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. At the lowest level, ROS offers a message passing interface that provides inter-process communication [3]. This inter-process communication relies on a system made up of nodes and topics. A node is an executable that uses ROS to communicate with other nodes. This communication is executed through the use of topics, nodes can publish messages to a topic as well as subscribe to a topic to receive messages. Messages are simply data types used when subscribing or publishing to a topic, and can contain any combination of data types [4].

Amongst many other packages, ROS also has a robot navigation package; the Navigation Stack [5]. The Navigation Stack takes information from sensor streams and provided knowledge of the environment and outputs velocity commands. The conversion from information to command velocities is commonly done by the global and local planners. The global planner generates a plan to get from the current position to a goal position, and the local planner uses the direct environment to ensure this plan is followed by the robot without encountering problems. Therefore, usually, the local planner is responsible for the dynamic obstacle avoidance [6]. These planners are based on costs that are registered in a costmap. A costmap is a grid-based map where each cell is described by a cost.

Global planners are conventionally fast interpolated navigation functions that can be used to create plans for mobile robots. These planners operate on the previously introduced costmap to find a minimum cost plan from a start point to an end point in a grid. This cost minimization is often based on existing algorithms such as Dijkstra's or A*. Within ROS, many different global planners exist along with many different navigation interfaces. The most commonly used interface is `move_base` [7], [8], [9]. There are various different global planners to interact with this navigation interface, three of which will be further investigated here. Firstly, the global

planner currently implemented on Dory; the `navfn` planner [10]. Secondly, the most commonly used global planner and official successor of `navfn`; the `global_planner` [11]. Finally the newest and most state of the art global planner; the `dlux_global_planner` [12].

2.1.1 The `navfn` planner

The `navfn` planner was first introduced in 1999 [13]. It is the first global planning package that adheres to the `BaseGlobalPlanner` interface allowing it to be easily implemented as a plugin to work together with the navigation packages. The planner is based on a global dynamic window approach as a generalization of the dynamic window approach, which was introduced two years earlier and rested on the principle of implementing the constraints imposed by limited velocities and accelerations of the robot [14]. This, if all parameters are tuned correctly, leads to a path that is optimal for the robot it was tuned for, since it rests on the specific robot dynamics. The cost minimization is done using Dijkstra's algorithm. The main issue with this planner is that it is not very flexible and tuning and optimizing are not as easy as one would like it to be. Therefore, a successor was created; the `global_planner`.

2.1.2 The `global_planner`

The `global_planner` was built as a more flexible replacement to `navfn`. The basic functionalities are equal to those of `navfn`. Tuning, however, has been expanded with many different parameters to further tune the planner to match the desired behavior of the robot. This includes the choice of implementing a different cost minimization algorithm, A*, and various filters that can change the planners outcome. This planner is meant to be used with the standard Navigation Stack described earlier in this chapter. This Navigation Stack has been the main navigation basis in ROS for a long time, and in the middle of 2018 a successor was first introduced; the `robot_navigation` package including the new `dlux_global_planner`.

2.1.3 The `dlux_global_planner`

The `robot_navigation` package is being developed to be the spiritual successor of the Navigation Stack. Along with it came the new `dlux_global_planner`. This planner is again mostly based on the `global_planner` and thereby the `navfn` planner. The main difference is its compatibility with the `robot_navigation` package. It can, however, also be implemented to work with the original Navigation Stack to investigate potential improvements. Two main differences in implementation are the addition of the PotentialCalculator and the Traceback. The PotentialCalculator calculates a numerical score for some subset of the cells in the costmap. The potential should be zero at the goal and grow higher from there. The Traceback uses that potential to trace a path from the start position back to the goal position. This allows for further customizability and could improve planning behaviour.

Based on the information found on these planners, the following can be concluded:
the `dlux_global_planner` is a refactoring for the `robot_navigation` package of the `global_planner` which in turn was a refactoring of the `navfn` planner. Much of the core implementation is based on design decisions of the authors of `navfn`, and according to the authors themselves, for better or for worse. Therefore, the plans generated by these planners are highly likely to resemble each other.

2.2 Costmaps

The aforementioned global planners are based on costs that are registered in a costmap. This costmap takes in sensor data from the world, and uses that information to build an occupancy grid of the data. Then, the costs are combined to form a 2D costmap, where the costs are often inflated based on a user specified inflation radius. The cost can be described as three different values; lethal space, known space and unknown space. Lethal space means that there is an obstacle at that position and the robot will definitely be in collision if it is at that position. Known space has a cost varying from 0 to 253, and the cost is used to steer the robot in desired directions. The interpretation of unknown space differs for each global planner, but means that there has been no sensor feedback from that position [15], [16]. The costmap used to be a single layer where all costs are combined, however, in 2014 a new concept was introduced; layered costmaps [17]. This concept is visualized in Figure 2.1:

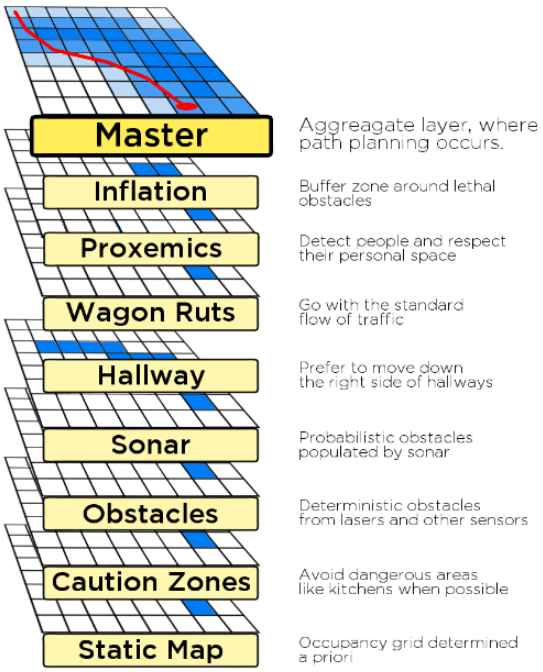


Figure 2.1: A stack of costmap layers, showcasing the different contextual behaviors achievable with the layered costmap approach [17].

Using this approach, each layer can have its own behaviours and role in the master costmap. This also allows for more customization within the Navigation Stack, as a user can build and add custom layers.

The costmap used by the global planner, the global costmap, is usually a larger map with a lower resolution compared to the costmap used by the local planner, since to global planner has to take a larger area into account. Therefore, generally, the global planner is not used for replanning, since the computational power required for generating a new global plan is much higher than letting the local planner diverge from the global plan and locally solve encountered issues [18]. The low resolution of the global costmap is an additional reason to let the local planner solve encountered issues.

2.3 Social navigation layers

Since the layered costmap structure allows for more customization, various new layer plugins have come available as ROS packages. One of these plugins is the `social_navigation_layers` plugin. It contains two different layers users can add to their costmap layer stack, both of which are designed to implement various social navigation constraints, such as proxemic distance. They both share functionality in that they both subscribe to where people are and alter the costmaps with a Gaussian distribution around those people [19]. The tracking of people allows for the prediction of their movement. Here, this is used to move slower when people are moving in the direction of the robot and allowing the robot to move faster when people move away from it. This prediction is based on a Kalman filter acting on the position of a person for a specified number of updates [20].

2.4 The standard approach

Dynamic obstacle avoidance is usually performed by implementing an obstacle layer to the costmap layer stack. This obstacle layer takes various sensor inputs and writes all hits as lethal space in the costmap. This prevents the robot from moving into these areas. By combining this with an inflation layer that inflates all obstacles based on the dimensions of the robot, this prevents the robot from colliding with obstacles. This sensor input is generally implemented within the local costmap and the local planner ensures that these obstacles are avoided. A static map is also often implemented, this is a map that is commonly user specified and provides information about the environment whilst not requiring the robot to have previously been there. This static map is commonly used by the global planner [15], [8].

Chapter 3

Experimental setup

This chapter elaborates on the setup used during this thesis to test the developed dynamic obstacle avoidance method. This includes an explanation of the requirements to be met by the developed dynamic obstacle avoidance method, a description of the action plan and a description of the hardware platforms the tests will be performed with.

3.1 Requirements

The dynamic obstacle avoidance method has to fit within the existing framework used for Dory, and should leave the main functionalities intact. Dory uses `move_base` as navigation interface, and, as described in Chapter 1, the method developed during this thesis should not interfere with the existing software. Therefore, the navigation interface is maintained at `move_base`. The current navigation method used with Dory relies mostly on the local planner. Usually, as mentioned in Section 2.4, the local planner is responsible for dynamic obstacle avoidance. Only when the local planner can not find a new plan, the global planner is used. In this case, however, the local planner is responsible for ensuring Dory is correctly aligned with the water currents, meaning the local planner is too busy to also handle dynamic obstacle avoidance. Therefore, to ensure the existing framework remains intact, the global planner is made responsible for the dynamic obstacle avoidance.

As mentioned in Section 2.2, the global costmap is usually a large map with a low resolution, which is not an issue since the local planner compensates for this by using the local costmap which is much smaller and of higher resolution. During this thesis, however, the global planner is responsible for obstacle avoidance, therefore requiring the global costmap to have a sufficiently high resolution as well as maintain the size. This impacts the achievable update rate of the costmap. For safe navigation and taking into account that the obstacles are dynamic and can be moving towards Dory, requiring fast replanning, an update frequency of the costmap of 2.0 Hz is required. This frequency should be met both in simulation as well as on the robot itself.

Dory should be localized properly, with little disturbances and variations. This is required for reliable obstacle detection and prediction. Collision prevention should be the highest priority of the obstacle avoidance algorithm, meaning collision avoidance surpasses the importance continuous operation. As also mentioned in Chapter 1, the cost of the WasteShark must stay low, therefore the method developed during this thesis should rely only on the existing hardware.

Dory should also behave differently towards dynamic obstacles compared to static obstacles. As the behaviour of static obstacles is known, Dory can move around these obstacles closely

and efficiently. The behaviour of dynamic obstacles, however, can be predicted but may never assumed to be known. Therefore Dory should avoid these obstacles in a more conservative way.

3.2 Setup

Chapter 1 already mentions that the new method for dynamic obstacle avoidance developed during this thesis uses obstacle tracking and location prediction to generate safe paths for Dory to follow. The paths will be generated by the global planner and based on the global costmap to minimize interference with the current framework. To implement the distinction between behaviour of static obstacles and dynamic obstacles, two custom costmap layers will be developed; a static obstacle layer and a dynamic obstacle layer. Note that here the static and dynamic adjective relates to the obstacle and not the costmap layer. The costmap layers will be based on a custom ROS message type, as described in Section 2.1, which describes obstacles by their size, position and velocity. Within the costmap layers, these obstacle messages will be correctly transformed into costs displayed and used in the global costmap. A custom node will be developed that fills this message based on sensor data. Then, the performance of the resulting global costmap will be tested together with the three different global planners described in Section 2.1.

The performance of these tests will initially be investigated in simulation. Once sufficient performance is achieved in simulation, testing will proceed with a real robot. As the proposed solution is generic, this can also be tested on different robots. Since Dory is a popular robot and testing with Dory in water could be a risky start, initial testing will be done with Clara. Clara is Nobleo Technology's AGV platform, equipped with, amongst other sensors, a LiDAR. Once testing with Clara provides sufficient confidence in the developed software, testing will proceed with Dory. Dory is located at the Ijzeren Man in Eindhoven. Initial implementation will be performed at Nobleo Headquarters after which Dory will return to the Ijzeren Man where the final tests can be performed. As there are no ships to test with at the Ijzeren Man, pedalos (also known as paddle boats or "waterfietsen") will be used to imitate the behavior of ships. A picture of the Ijzeren Man is shown in Figure 3.1.

Figure 3.1: The Ijzeren man, testsite of Dory.

These tests will also be performed with the traditional implementation of obstacle avoidance as explained in Section 2.4. Since the implementation developed during this thesis is designed to be a significant improvement in busy waters, the benefits of this method as opposed to the

traditional method might be difficult to observe. The results will therefore mostly be used as a proof of principle.

3.3 Dory

The autonomous version of the WasteShark, the one that Nobleo Technology is working on, is called Dory. Dory is a rectangular drone of approximately 1.5 by 1 meter and 0.5 meters high. Dory is open at the front so by faring in the direction of the waste, the waste is collected in the robot. Dory can "eat" up to 200 liters of waste per run. When Dory has collected enough waste, which she can measure by comparing her thrust to her velocity, she can fare to her docking station where she can drop the collected waste and continue to clean the water.

As mentioned in Section 3.1, Dory should have reliable localization. This should not be a problem since Dory is equipped with an RTK-GPS system. This provides Dory with the GPS signal as well as the position error of the GPS signal through a calibrated fixed base station, the SharkPod. Therefore, Dory can correct for the position error, which results in a very reliable and steady GPS location. Dory is equipped with two computers; an odroid board as well as a small pcb. Dory is also equipped with a LiDAR, the RPLiDAR A2, which provides laser data at a 360° circular range and 12 m. distance range. Figure 3.2 shows Dory.

Figure 3.2: Dory, Nobleo's testbot of the RanMarine WasteShark.

As described in previous chapters, there exists an elaborate software framework for Dory already, which is developed using ROS. Within this framework, various functionalities have been implemented, such as autonomous docking, basic path following navigation and a simulator.

3.4 AGV platform Clara

Nobleo Technology also has her own AGV platform called Clara. The hardware and software framework of Clara are surprisingly similar to that of Dory, which ensures that the results found with the initial testing with Clara also provide a representation of the performance on Dory. Clara is also equipped with an RPLiDAR A2, also uses two PC's; an Odroid and pcb, and also has reliable and steady localization. The localization of Clara is not performed through GPS, but through a specialized sensor; the Accerion sensor.

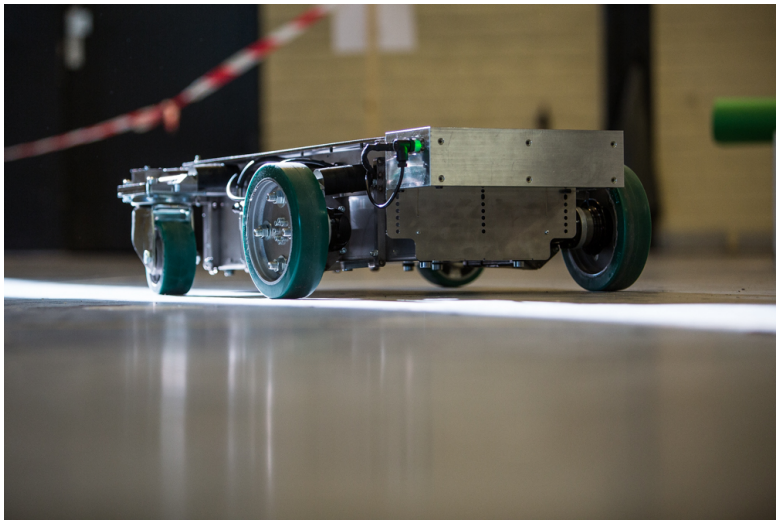


Figure 3.3: Clara, Nobleo's AGV platform.

3.5 Hypothesis

In Chapter 1, the following research question was formulated:
What approach for dynamic obstacle avoidance based on the hardware available on Dory can decrease the amount of required replanning and increase the knowledge of future representations of the environment compared to standard methods whilst still meeting safety requirements?

Based on the described requirements and the provided setup, the following hypothesis can be formulated:
Navigation based on layered costmaps using LiDAR based obstacle detection and prediction will result in an increase in planning options and a decrease in required replanning whilst still meeting safety requirements.

Chapter 4

Methods

As described in Section 3.2, a ROS message type will be developed that describes obstacles by their size, position and velocity. A node will be developed that fills this message based on sensor data and two custom costmap layers will be developed; a static obstacle layer and a dynamic obstacle layer. The general setup of these developments will be described in this chapter.

4.1 Sensor integration

As mentioned in Section 3.2, all obstacles have to be published to a topic that the custom costmap layers can listen to. On this topic, a list of obstacles is published and obstacles are described as follows:

$$\texttt{obstacle} = \begin{cases} \texttt{id} & \text{string} \\ \texttt{lidar\_index} & \text{integer} \\ \texttt{pose} & \begin{cases} \texttt{position} & \text{point} \\ \texttt{orientation} & \text{point} \end{cases} \\ \texttt{velocity} & \text{point} \\ \texttt{size} & \text{point} \end{cases}$$

where a point is described as three doubles. The `lidar_index` represents the LiDAR data index of the center point of an obstacle and is required for the clearing of previous obstacles.

Detecting and publishing these obstacles is done in two steps:

1. Based on the LiDAR data, obstacles are registered and updated with `id`'s, `lidar_index`es, `pose`s and `size`s.
2. Based on the `id` and `pose`, a Kalman filter determines the velocity of the obstacle.

The second step is taken from one of the dependencies of the `social_navigation_layers`, the `people_velocity_tracker`. This is a node that determines the velocity of people, and with various small alterations, can also be used to determine the velocity of obstacles. The first step, however, requires a new node.

The node performs the following steps:

1. Loop through the data of the LiDAR and segment obstacles.
2. Remove the obstacles that are within unknown or lethal space based on the static map.
3. Check if the `lidar_index` is sufficiently close to that of previously detected obstacles; if so, update the older obstacle, else, create a new one.
4. Combine the valid obstacles in a list and publish to a topic for the velocity node for further processing.

The segmentation of obstacles based on LiDAR data is done through L-shape fitting. This means each obstacle is described by three points; the start point (e.g. the first point of the obstacle the LiDAR encounters), the closest point and the end point. This has one main limitation, namely that all obstacles will be represented as rectangles. Another problem is that, from the side, the LiDAR data of a ship can be presented as a line. In that case, attempting to fit an L-shape will not yield realistic representations. Therefore, an additional check is integrated in the node; a check to see if the three found points represent a line or a triangle. The pseudocode for finding the three previously mentioned points based on LiDAR data is described in Algorithm 1.

Algorithm 1: Processing the LiDAR data to segment obstacles into the form of three points

Input : Laser data
Output: A list of obstacles, represented as three points

```
/* temp_obstacle is a data type described by three points and the lidar_index */
data[] = LiDAR.data                                   // array filled with lidar data
in_obstacle = data[0] < LiDAR.max_range     // boolean represents state within loop
min_jump = config.min_jump     // user specified minimal distance between obstacles
closest_entity = LiDAR.max_range
for i ← 0 to data.size() by 1 do
    if in_obstacle then
        if (data[i] < closest_entity) then
            /* transform lidar point to position w.r.t.  the global frame */
            temp_obstacle.closest_point = found point
            closest_entity = data[i]
        else if (data[i + 1] − data[i] > min_jump) then
            /* transform lidar point to position w.r.t.  the global frame */
            temp_obstacle.max_point = found point
            /* add temp_obstacle to obstacle list */
            in_obstacle = false temp_obstacle =  closest_entity = LiDAR.max_range
    else if (data[i − 1] − data[i] > min_jump) then
        /* transform lidar point to position w.r.t.  the global frame */
        temp_obstacle.min_point = found point
        temp_obstacle.lidar_index = i
    /* proceed to next step */
end
```

The remaining steps performed in this node are straightforward and not further discussed here. After this node, the list of obstacles is sent to the `velocity_filter` node, which concludes the sensor integration.

4.2 Custom static layer

As explained in Section 3.1, Dory should behave differently towards dynamic obstacles compared to static obstacles. Static obstacles will behave predictably allowing for the safety margins to be smaller. Another reason to handle static obstacles differently is to allow Dory to "remember" them. Ships are often anchored for longer periods of time, so only registering them when they are within the range of the LiDAR could lead to inefficient replanning. Therefore static obstacles that Dory can no longer see will be kept in a list of logged obstacles, and will remain in the costmap for a used specified amount of time. This also requires a clearing function that inspects whether logged static obstacles are still at their specified location. This function will investigate, whenever logged obstacles are within Dory's LiDAR range, whether that specific logged obstacle is still there. If not, the obstacle will be removed from the `logged_obstacles` list.

The custom static obstacle layer is set up as follows; it subscribes to the topic the obstacles are published on and disregards the obstacles with a sufficiently high velocity. Obstacles with a very low velocity are also regarded as static obstacles to compensate for irregularities of the Kalman filter. Then, all logged obstacles are compared to the user specified logging period. If any logged obstacles have not been updated for a period longer than the logging period, they are removed from the list. The new obstacles are then checked with the logged obstacle list to see if any logged obstacles should be updated. If the obstacles are not in the logged list, they are added to the list after which all logged obstacles are used by the plotting function to be added to the costmap.

4.2.1 Plotting in the costmap

The plotting of obstacles in the costmap is similar for both the static and dynamic obstacle layer, and is based on the method for plotting people as used in the `social_navigation_layers` package. The function is structured as follows:

1. Loop through the to be plotted obstacles.
2. For each obstacle, determine the rectangle around the obstacle in which the cost should be updated. This is done based on the position and size of the obstacle.
3. For each obstacle, transform the position of the obstacle and the the rectangle found in step 2 to the gridmap coordinates.
4. Loop through the cells that fall into the gridmap rectangle from step 3.
5. For each cell, investigate whether the cell is within the obstacle, if so, set the cost of that cell to lethal.
   For the static layer, the remaining cells are set to the value they had before entering this layer. Within the dynamic layer, the function proceeds as follows:
6. For each cell not in the obstacle, investigate whether the cell is within the Gaussian distribution around the obstacle based on its velocity. If so, set the value of the cell to the value of the Gaussian distribution at that cell. This Gaussian distribution is further explained in Section 4.3.
7. The remaining cells are set to the value they had before entering this layer.

Investigating whether a cell is within an obstacle is done by converting the obstacle to a rectangle based on the position and size. Then, the cell is projected onto two perpendicular edges of the rectangle and this projection is used to verify whether the cell falls within the rectangle [21].

4.3 Custom dynamic layer

Since, as explained in Section 3.1, the behaviour of dynamic obstacles may never assumed to be known. Therefore, an added safety margin should be applied to dynamic obstacles. Since Dory can not come to a standstill instantly, providing an estimation of the future representation of the environment will prevent collisions. Chapter 1 mentions that in busy waters, Dory might have trouble navigating through all dynamic obstacles when using traditional obstacle avoidance methods. This could result in Dory being dormant for longer periods of time. Since traditional obstacle avoidance methods transform all encountered obstacles to lethal space on the costmap, the plan will never go through an obstacle, even if that obstacle will have moved away by the time the robot would be there. The custom dynamic obstacle layer developed during this thesis predicts the location of obstacles based on their velocity and the state of Dory, which allows the planner to plan through the current location of an obstacle and provides Dory with more time to react to obstacles.

The prediction of obstacles is done based on two different states; Dory has a goal position and a plan to get there, and Dory does not yet have a goal to move to or a plan to follow. To be able to differentiate between these two states, the dynamic obstacle layer also subscribes to the topic to which the status of `move_base` and to which the plan of the global planner are published. In the case that Dory has already received a path, the structure of the prediction function is as follows:

1. Loop through each point in the received path, and for each point in the path, determine how long it would take Dory to get there.
2. Loop though the list of received obstacles, and for each obstacle predict their location based on the time found in step 1 and the velocity of the obstacle.
3. Check if the predicted obstacle location is within a specified range of the point in the path. The specified range is based on the size of the obstacle multiplied with a safety margin factor.
4. If the obstacle is within that range, add it to a temporary array of obstacles and their predicted location.
5. Update the temporary array based on the closest prediction per obstacle to the path, provided that the obstacle is within the range.
6. Once all points within the path have been investigated, overwrite the to be plotted list of obstacles with the temporary array.

In the case that Dory does not have a goal or plan these steps are taken:

1. Loop through the list of received obstacles.
2. For each obstacle, determine the time required by Dory to get to the position of that obstacle.
3. Predict the location of the obstacle based on that time and the velocity of the obstacle.
4. Overwrite the to be plotted list of obstacles with the list of predicted obstacles.

Similar to the static layer, the dynamic obstacle layer subscribes to the topic the obstacles are published on. Here, however, the obstacles with a sufficiently low velocity are disregarded. Then, the location of these obstacles is predicted based on the methods described earlier in this section. The list of predicted obstacles is then used by the plotting function to be added to the costmap.

As mentioned in Section 4.2.1, the dynamic obstacle layer implements a Gaussian distribution

based on the direction and magnitude of the velocity of the obstacle. The implementation of this Gaussian distribution is taken from the `social_navigation_layers` package. It is uses a two dimensional elliptical Gaussian function as described in Equation (4.1):

$$
\begin{aligned}
f(x,y) &= A\,\exp\left(-\left(\mathrm{a}(x-x_0)^2 + 2\mathrm{b}(x-x_0)(y-y_0) + \mathrm{c}(y-y_0)^2\right)\right) \qquad \text{where} \\
a &= \frac{\cos^2(\theta)}{2\sigma_x^2} + \frac{\sin^2(\theta)}{2\sigma_y^2}, \\
b &= -\frac{\sin(2\theta)}{4\sigma_x^2} + \frac{\sin(2\theta)}{4\sigma_y^2}, \\
c &= \frac{\sin^2(\theta)}{2\sigma_x^2} + \frac{\cos^2(\theta)}{2\sigma_y^2}
\end{aligned}
\tag{4.1}
$$

where A equals the amplitude, which equals a user specified value. x_0 and y_0 represent the center of the Gaussian distribution which equals the predicted position of the obstacle. σ_x and σ_y are the x and y spreads, which are defined as a combination of the velocity, obstacle size, user specified covariance and a user specified factor with which to amplify the velocity contribution. θ represents the orientation of the velocity vector.

4.4 Planners

Since the costmap is updated at a frequency of 2 Hz, as required per Section 3.1, the global planner should also replan for each newly received costmap. The `planner_frequency` of the global planner is therefore set to 2 Hz. The influence of using different global planners is also investigated. The three global planners mentioned in Section 2.1 are implemented and compared. The implementation of these planners only requires the launch file of the `move_base` node to use a different global planner parameter. Since the `dlux_global_planner` is developed for a different navigation package, a `GlobalPlannerAdapter` is used to enable this planner to work with `move_base`. Within the parameter file of the global costmap, various parameters of the global planner can be modified. A multitude settings for these planners will be tested such as the use of a different cost optimization algorithm, A*.

Chapter 5

Results

Chapter 4 thoroughly describes the implementation of the obstacle avoidance method developed during this thesis. This chapter shows the results of this implementation as well as analyze the implications of these results.

5.1 Results

The results of the costmap can be investigated using RViz, a 3D visualization tool developed for ROS. Costmaps in RViz show costs though colors, pink represents lethal space and blue through red represent other non-zero costs.

5.1.1 Simulation

As mentioned in Section 3.2, all initial tests are performed in simulation. In simulation, a static map is also implemented. This static map is user specified and is not dependent on the LiDAR data. This static map is therefore not visible in Gazebo, the used simulation GUI. The result of the dynamic costmap layer can be seen in Figure 5.1.

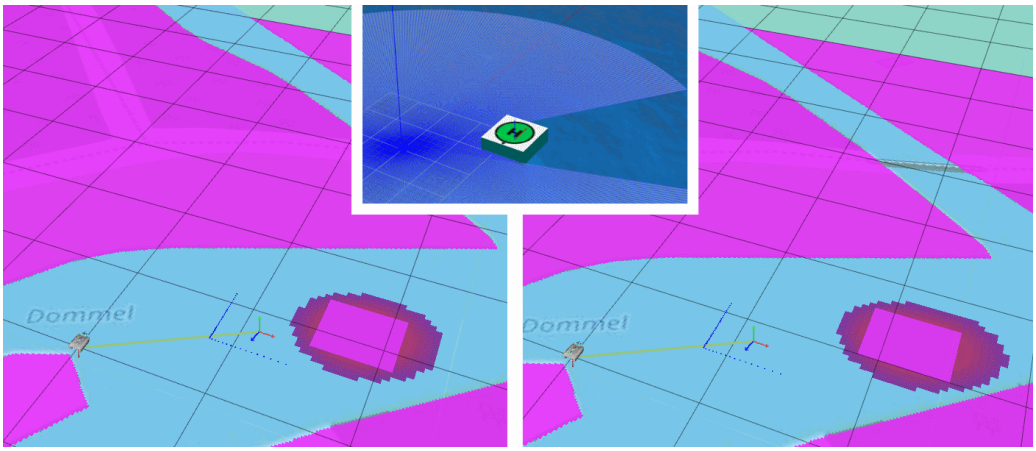
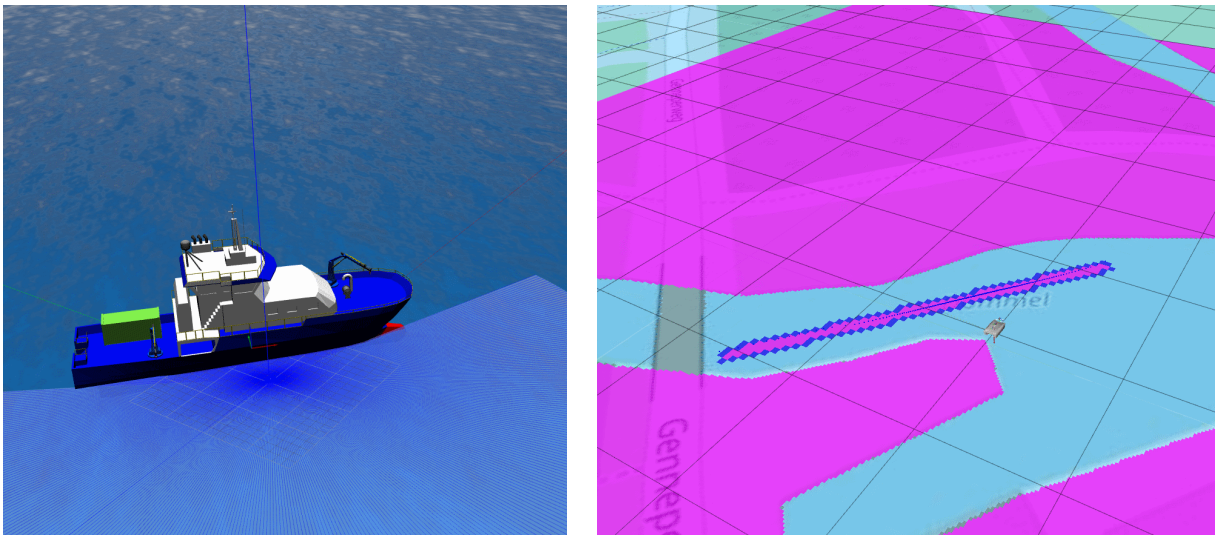


Figure 5.1: At the top the Gazebo simulation view is visible, the blue lines represent the LiDAR rays, and Dory is at the center of them. The two RViz images depict the resulting costmap for two different velocities. The velocities are simulated by moving the block in Gazebo, and the difference can be seen by the size of the Gaussian distribution and the location of the plotted obstacles. The blue dots are the LiDAR data, and the frame representing the current position of the obstacle is visible at the center of these points. The static costmap is also shown.

The custom static costmap layer is also tested in simulation. The results of this test can be seen below:



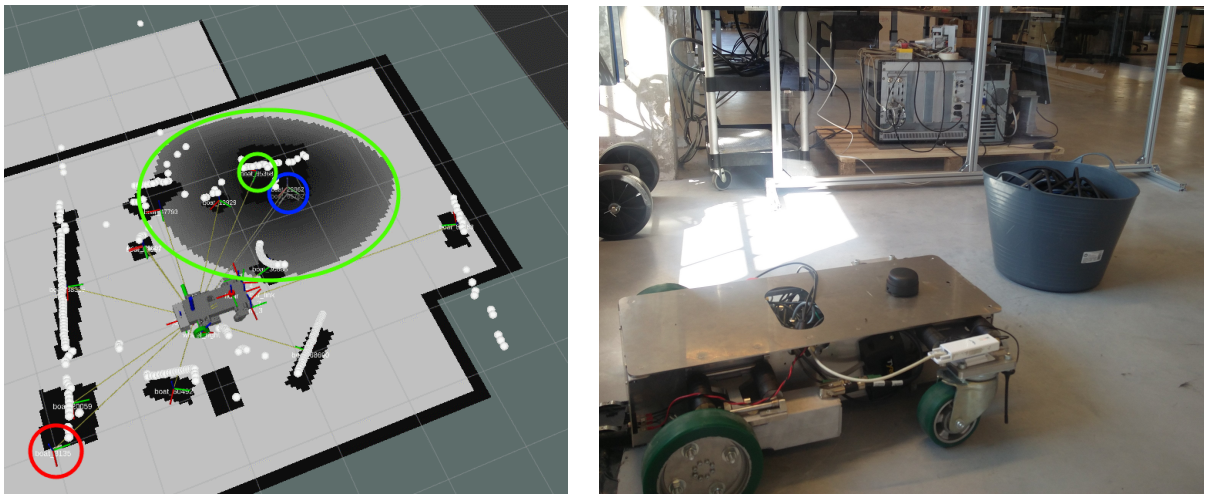
(a) Gazebo view of simulation.

(b) RViz showing the resulting costmap.

Figure 5.2: Figure 5.2a shows the simulation situation in Gazebo. The ship represents a static obstacle. As explained in Section 4.2, from the side, the LiDAR data of an obstacle can in some cases be presented as a line. This is visible in this simulation scenario as well. Therefore, there is no way for Dory to know the width if the ship. The width then resolves to a user specified default value. The ship is plotted in the costmap as static obstacle visible at the center of Figure 5.2b. Similar to Figure 5.2b, the static layer is also shown.

5.1.2 Real life tests

After testing in simulation, the software is integrated into Clara. Various boxes and buckets are placed to act as obstacles. By moving these obstacles, the custom static layer as well as the dynamic layer are tested.



(a) Costmap found with testing on Clara.

(b) Test environment with Clara.

Figure 5.3: The costmap depicted in Figure 5.3a shows various detected and plotted obstacles. The environment used to test the costmap layers is shown in Figure 5.3b.Figure 5.3a shows a ghost frame in red (e.g. frames with no matching plotted obstacle). The Gaussian distribution is clearly visible for one obstacle with a low velocity (green), the low velocity is visible since the predicted obstacle is not plotted far away from the frame. There are also a few dying frames, with a grey color (blue), which implies false detections were present and are now being cleared. It can also be seen that the LiDAR data points that are in known lethal space or in unknown space are correctly not converted to obstacles.

The initial parameter configuration used during testing proved incapable of achieving the required update frequency of 2.0 Hz. This is remedied by decreasing the costmap resolution and size to more relevant levels. Additionally, the clearing function of the static costmap layer was called based on a timer. By calling this function based on environmental conditions, the runtime of the costmap layers is improved to meet the 2.0 Hz requirement.

To test the accuracy of the velocity estimation performed by the Kalman filter, a bucket is placed on a dolly. This bucket is attached to a cord, and by steadily pulling the cord, the bucket is moved along a distance of 2 meters. This movement is timed and used to determine the velocity of the object. This is then compared to the velocity reported by the Kalman filter, the data from the travelled distance of 0.5 meters to 1.5 meters is taken to neglect acceleration and deceleration results. This is done for two different velocities en is repeated 5 times per velocity. The results are shown below:

Average performed velocity	Average measured velocity
0.180 m/s	0.214 m/s
0.320 m/s	0.334 m/s

Table 5.1: Results of velocity test with Clara. This table shows that the average velocity determined by the Kalman filter provides a decent estimation.

Once testing with Clara provided sufficient confidence in the developed software, testing is proceeded with Dory. The results of testing with Clara did, however, provide relevant insights into the importance of a high-end LiDAR. The LiDAR Clara and Dory are equipped with are noisy, which leads to false detections and incorrect velocity measurements. Therefore, in order to properly test the costmap layers using Dory, a different LiDAR is mounted on Dory; the Hokuyo UXM-30LXH-EHA. This improved LiDAR has a 190° circular range and 80 m. distance range, and is mounted at the front of Dory which means Dory will not have LiDAR data behind her. The testing at the Ijzeren Man did not run smoothly, as there was a problem with the GPS localization. The GPS localization is, as explained in 3.1, crucial for the developed method to function correctly. The problem with the GPS signal caused Dory to not register translational movements which prevented the testing of path following. Since Dory was fixed at the global frame of the map, the distance to obstacles varied when Dory was moving, which, in the costmap lead to the obstacles changing position instead of Dory. This also meant the velocity determination of obstacles did not function as desired. The costmap layers, however, could be tested by maintaining Dory at a specific location, since in that case only the movement of the obstacles is registered. The resulting costmap with the riverbank as detected obstacle shown in Figure 5.4.

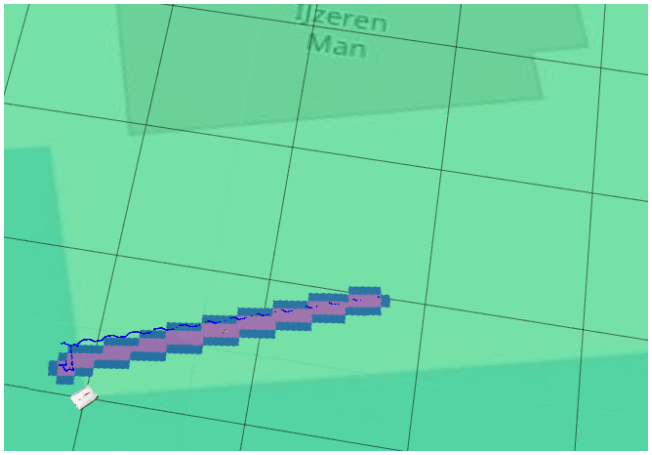


Figure 5.4: Costmap result from testing with Dory at the Ijzeren Man.

The three different global planners are also compared in simulation. For three different test situations, all three global planners are tested. Since the testing of these different global planners is not the main focus of this thesis, these results can be found in Appendix A.

5.2 Discussion

The tests performed in simulation showed very promising results, obstacles are correctly detected and converted to objects to be handled and plotted by the costmap layers. The obstacle prediction works reliably and provides confidence in the use of obstacle prediction as a more progressive approach to dynamic obstacle avoidance.

The real-life tests, however, highlighted a few issues with the suggested approach. First of all, the data obtained with both used sensors was noisy (the RPLiDAR significantly more than the Hokuyo LiDAR), which led to false detections and incorrect velocity measurements. This could be solved by using a filter that takes multiple measurements and only passes on the data with sufficient overlap between the measurements. Another issue is that the limits in

the costmap layers, such as from what velocities on an obstacle is regarded as dynamic, are very strict and therefore sensitive. This leads to obstacles suddenly changing position in the costmap, which makes the costmap more chaotic. Implementing a middle range, or implementing a form of gradual transitioning in the dynamic layer could improve this behaviour.

The test results obtained with Clara showed that the method developed during this thesis is not suited for small environments. The high quantity of small obstacles leads to obstacle detection to be increasingly complex, especially the separation of obstacles is more challenging. In a small environment, the proposed method of this thesis does not have any clear benefits compared to the standard approach mentioned in Section 2.4. Small obstacles, such as people, do not behave predictably compared to obstacles Dory will encounter, such as ships. Therefore, obstacle prediction provides untrustworthy results and a standard obstacle layer will be more reliable and safe. Testing with Dory was troublesome, and in order to provide reliable conclusions, more tests have to be performed.

Two requirements described in Section 3.1 are not met, a different LiDAR is required and Dory was not localized correctly. The second unmet requirement was a singularity and should not be a reoccurring problem. The first unmet requirement, however, can not easily be fixed. Adding a filter as mentioned earlier in this section should help, but it will not completely resolve the problem. A suggestion for RanMarine is to offer clients different versions of the WasteShark; a more expensive one that can operate in busy waters, and a standard one that can only operate safely in waters with little marine traffic. Initially, the required update frequency of 2.0 Hz was also not met. However, by adjusting the costmap parameters and by calling various functions at lower frequencies, the requirement is met.

In general, the newly developed method performed very well in simulation, but in real life it showed chaotic and unstable results. Therefore, based on the current status of the developed method, it would be unwise to implement this in a commercially available product. Using the standard approach is expected to be more reliable. By addressing the mentioned issues with the developed method, it could be an improvement if the standard approach leads to inefficient replanning, no plan can be generated or if collisions with dynamic obstacles occur regularly and a less conservative approach is desired.

Chapter 6

Conclusion

The goal of this thesis is to develop a dynamic obstacle avoidance method based on LiDAR based obstacle detection and prediction and investigate whether this method can be used as a more progressive but still safe obstacle avoidance method. Based on related work described in Chapter 2 and the requirements described in Section 3.1, an action plan is described in Section 3.2. This action plan combined with the following hypothesis:
Navigation based on layered costmaps using LiDAR based obstacle detection and prediction will result in more planning options and a decrease in required replanning whilst still meeting safety requirements.
leads to an implementation of two custom costmap layers as well as an obstacle detection node that are described in Chapter 4. The results of the tests of this implementation are discussed in Chapter 5.

The simulation results are very promising and show that the suggested approach could act as a replacement for the standard obstacle layer in environments with predictable dynamic obstacles. The real life tests performed with Clara and Dory show that the implementation of the introduced method works and that the velocity determination and obstacle location prediction functions, but should be improved. This improvement will require an enhancement to the performance of the LiDAR for reliable obstacle detection. The tests with Clara reveal that the introduced system is only an improved method compared to the standard obstacle layer when obstacles behave predictably. Various improvements, discussed in Section 5.2, are required for this introduced method to be sufficiently robust to qualify as a replacement to the traditional obstacle layer.

The recommendation for Nobleo Technology and RanMarine Technology is therefore to implement the standard approach described in Chapter 2 along with increasing the replanning frequency of the global planner. If, in the future, a more progressive approach is required since Dory is dormant often or for long periods of time, or if collisions with dynamic obstacles occur regularly the approach developed during this thesis could be further improved, using the suggestions explained in Section 6.1, and then implemented.

6.1 Future work

For the approach developed during this thesis to function sufficiently robust, various improvements are required. First of all, a LiDAR filter should be implemented that takes multiple measurements and only passes on the data with sufficient overlap between the measurements.

Secondly, the Kalman filter should be improved by integrating uncertainty. This can be done by using multiple sensors or by improving the use of previous measurements. Thirdly, the method developed during this thesis relies on various variables that have not been tuned optimally. This should be done before continuing with this method. Fourthly, a form of gradual transitioning in the dynamic layer should be implemented to decrease the amount of clutter and jumps in the costmap. Finally, the runtime could be improved further by calling more functions based on environmental conditions as opposed to timers.

Aside from required improvements, some additional functionalities and tools could be implemented to enhance the performance of the dynamic obstacle avoidance method. A different fitting algorithm, one that allows for the fitting of a variety of shapes, could be used to fit the shapes of obstacles based on LiDAR data. This will complicate the plotting of obstacles, increasing the runtime, but would improve the accuracy of the costmap. An Automatic Identification System (AIS), an automatic tracking system that uses transponders on ships, could be integrated to obtain more accurate data and to enable the avoidance of dynamic obstacles outside of the range of the LiDAR. Finally, a 3D LiDAR could be implemented to improve the accuracy of the costmap and aid with the rocking of Dory on rough waters. This will, however, increase the cost of Dory.

The concept behind layered costmaps could also be used for dynamic goal determination to directionally collect waste. Information such as the wind, the water currents, and history of waste locations could be combined in costmap layers, which could then be used to determine the highest likelihood of waste locations. This would make the waste collection of Dory more efficient and rational.

Bibliography

[1] Nobleo Technology. *About us*. https://nobleo-technology.nl/about-us. Accessed: 05-04-2019.

[2] Nobleo Technology. *Fully Autonomous Wasteshark*. https://nobleo-technology.nl/project/fully-autonomous-wasteshark. Accessed: 05-04-2019.

[3] ROS. *About ROS*. http://www.ros.org/about-ros/. Accessed: 08-04-2019.

[4] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[5] Matthias Gruhler. *The Navigation Stack*. http://wiki.ros.org/navigations. Accessed: 08-04-2019.

[6] Yoshimasa Goto, , Yoshimasa Goto, and Anthony Stentz. Mobile robot navigation: The cmu system. *IEEE Expert*, 2:44–54, 1987.

[7] Safdar Zaman, Wolfgang Slany, and Gerald Steinbauer. Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues. In *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, pages 1–5. IEEE, 2011.

[8] Arbnor Pajaziti. Slam–map building and navigation via ros. *International Journal of Intelligent Systems and Applications in Engineering*, 2(4):71–75, 2014.

[9] David C Conner and Justin Willis. Flexible navigation: Finite state machine-based integrated navigation and control for ros enabled robots. In *SoutheastCon 2017*, pages 1–8. IEEE, 2017.

[10] Kurt Konolige and Eitan Marder-Eppstein. Navfn. http://wiki.ros.org/navfn?distro=kinetic, 2014.

[11] Aaron Hoy David V. Lu!!, Michael Ferguson. *global_planner*. http://wiki.ros.org/global_planner. Accessed: 08-04-2019.

[12] David V. Lu!! *dlux_global_planner*. https://github.com/locusrobotics/robot_navigation/tree/master/dlux_global_planner. Accessed: 08-04-2019.

[13] Oliver Brock and Oussama Khatib. High-speed navigation using the global dynamic window approach. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 1, pages 341–346. IEEE, 1999.

[14] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[15] David V. Lu!! Shedding light on the ros navstack. In *ROSCON*, 2014.

[16] Hans P. Moravec. Sensor fusion in certainty grids for mobile robots. In *Sensor devices and systems for robotics*, pages 253–276. Springer, 1989.

[17] David V Lu, Dave Hershberger, and William D Smart. Layered costmaps for context-sensitive navigation. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 709–715. IEEE, 2014.

[18] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. The office marathon: Robust navigation in an indoor office environment. In *2010 IEEE international conference on robotics and automation*, pages 300–307. IEEE, 2010.

[19] David V. Lu!! *social_navigation_layers*. `http://wiki.ros.org/social_navigation_layers`. Accessed: 08-04-2019.

[20] David V. Lu, Daniel B. Allan, and William D. Smart. Tuning cost functions for social navigation. In *ICSR*, 2013.

[21] Eric Bainville. *Navfn*. `https://stackoverflow.com/questions/2752725`, 2010. Accessed: 21-03-2019.

Appendix A

Global planner results

All three global planners described in Section 2.1 are implemented via the `move_base` plugin, and using the same test case, these planners are compared. Dory uses the `navfn` planner, the result of this planner in the first test case can be seen below:

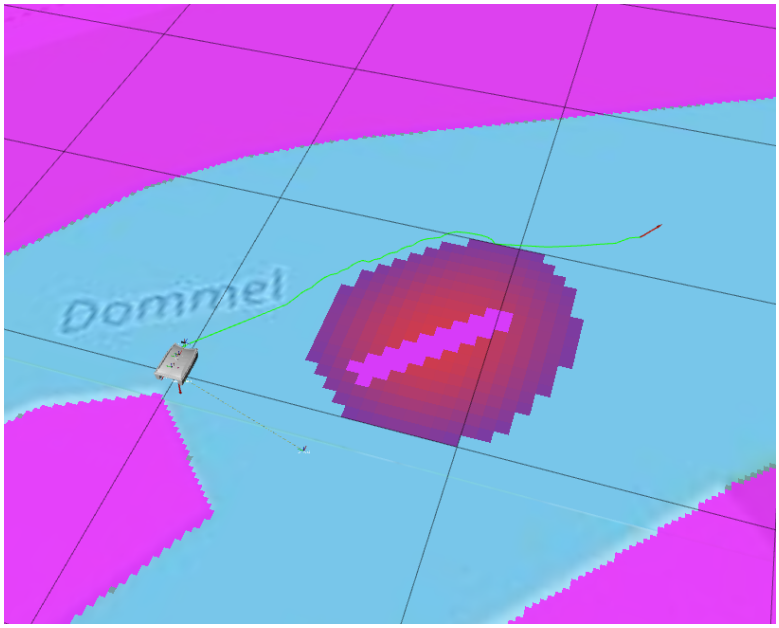
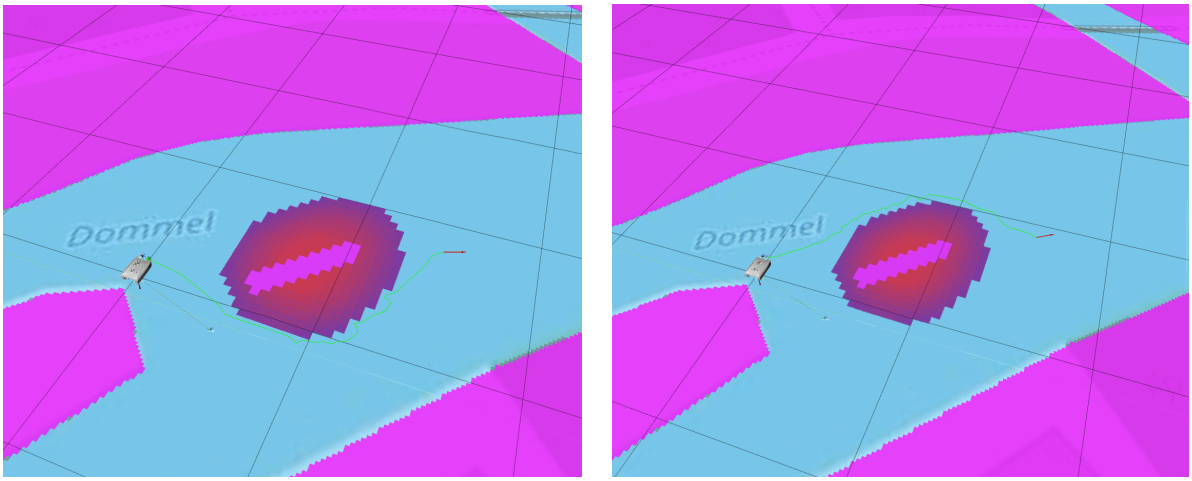


Figure A.1: Resulting path of the `navfn` planner. The resulting path avoids cells with a cost higher than zero. The path is smooth, except for a small part near the goal. The path dips near to the obstacle and the moves away again. This dip was visible with other test cases as well.

The `global_planner` was built as a more flexible replacement to `navfn`. The basic functionalities are equal to those of `navfn`. Tuning, however, has been expanded with many different parameters to further tune the planner to match the desired behavior of the robot. The result of this planner in the first test case can be seen below:



(a) Result using the standard settings.

(b) Result using altered settings.

Figure A.2: Results for using the `global_planner`. The path shown in Figure A.2a depicts the same behaviour as the path visualized in Figure A.1: a few small dips towards the obstacle but generally a smooth plan. Figure A.2b shows that path generated by the `global_planner` using A* and a quadratic potential calculation. The path shown in Figure A.2b does not contain the same dips, but the general path is less smooth.

Some further testing with the `global_planner` using A* and a quadratic potential calculation showed some strange behaviour visible below:

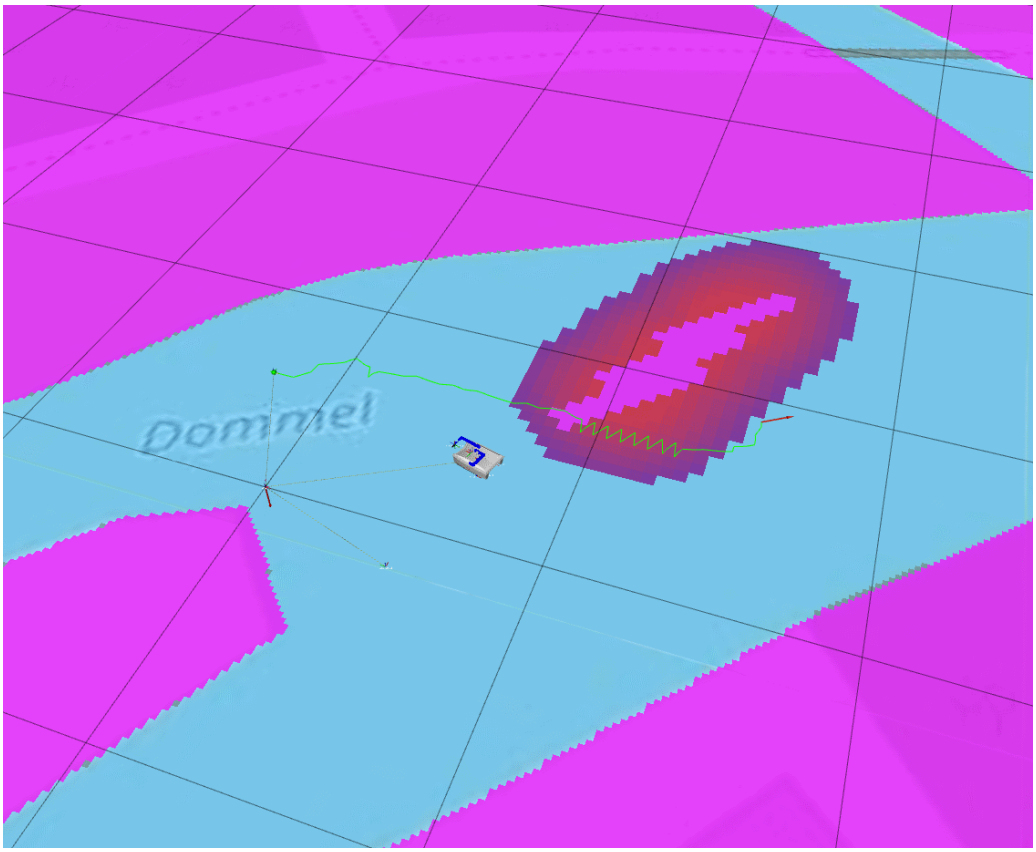
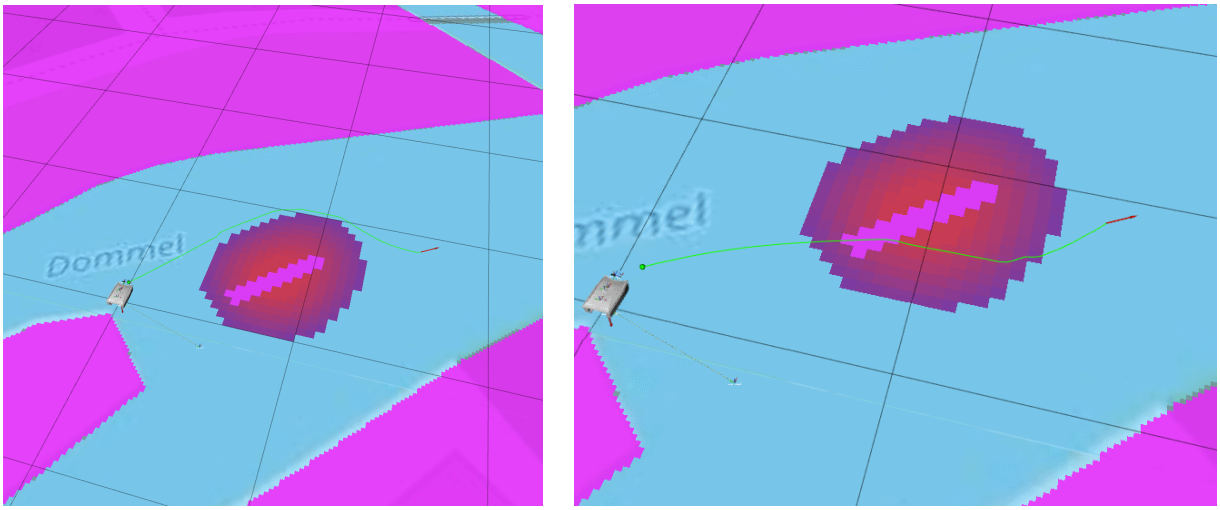


Figure A.3: The fact that the path passes through the obstacle has to do with the timing of the picture, so that is not an issue. The zig-zag behaviour, however, is a reoccurring issue, and not efficient.

The new `dlux_global_planner` is mostly based on the `global_planner` and thereby the `navfn` planner. It does, however, implement a few new functionalities. The result can be seen below:



(a) Result of the `dlux_global_planner`. (b) Fault within the `dlux_global_planner`.

Figure A.4: The path generated by the `dlux_global_planner` shown in Figure A.4a is the smoothest path shown in this chapter. There is, however, one big flaw which is clearly visible in Figure A.4b. The planner initially plans correctly, but then gradually starts moving through high costs and even lethal space. This planner is therefore not safe and will not be used.

The `global_planner` using A* and a quadratic potential calculation showed undesirable behaviour and will therefore not be used. Between the `navfn` planner and the standard `global_planner`, the standard `global_planner` on average showed the smoothest paths and will therefore be used.