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 with 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, poses and sizes.
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 sufficienly 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 is 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 */
1  data[] = LiDAR.data                         // array filled with lidar data
2  in_obstacle = data[0] < LiDAR.max_range     // boolean represents state within loop
3  min_jump = config.min_jump     // user specified minimal distance between obstacles
4  closest_entity = LiDAR.max_range
5  for $i \leftarrow 0$ to $data.size()$ by 1 do
6      if $in\_obstacle$ then
7          if $(data[i] < closest\_entity)$ then
               /* transform lidar point to position w.r.t. the global frame */
8              temp_obstacle.closest_point = found point
9              closest_entity = data[i]
10         else if $(data[i+1] - data[i] > min\_jump)$ then
               /* transform lidar point to position w.r.t. the global frame */
11             temp_obstacle.max_point = found point
               /* add temp_obstacle to obstacle list */
12             in_obstacle = false temp_obstacle =  closest_entity = LiDAR.max_range
13     else if $(data[i-1] - data[i] > min\_jump)$ then
           /* transform lidar point to position w.r.t. the global frame */
14         temp_obstacle.min_point = found point
15         temp_obstacle.lidar_index = i
       /* proceed to next step */
16 end
```

The remaining steps performed in this node are straightforward and not further discussed here. After this node, the list of obstacle is sent through 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 static obstacle layer 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. If the projection is bigger than zero and smaller than the projection of the edge on itself, the cell lies within the obstacle [15].

4.3 Custom dynamic layer

Since, as explained in Section 3.1, the behaviour of dynamic obstacles may never be assumed to be known. Therefore, an added safety margin should be applied to dynamic obstacles. 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, allowing the planner to plan through the current location of an obstacle.

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` is published and the topic to which the plan of the global planner is published. In the case that Dory has already received a path t, 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 Dory would need to get there and the velocity of the obstacle.
3. Check if the predicted boat 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 that position.
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 obstacle is predicted based on the methods described earlier in this section. The list of predicted boats are 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:

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

where A equals the amplitude is 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.

4.4 Planners

Since the costmap is updated at a frequency of 2 Hz as required in 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 planner 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 varied. Various settings for these planners will be tested such as the use of a different cost optimization algorithm, A*.