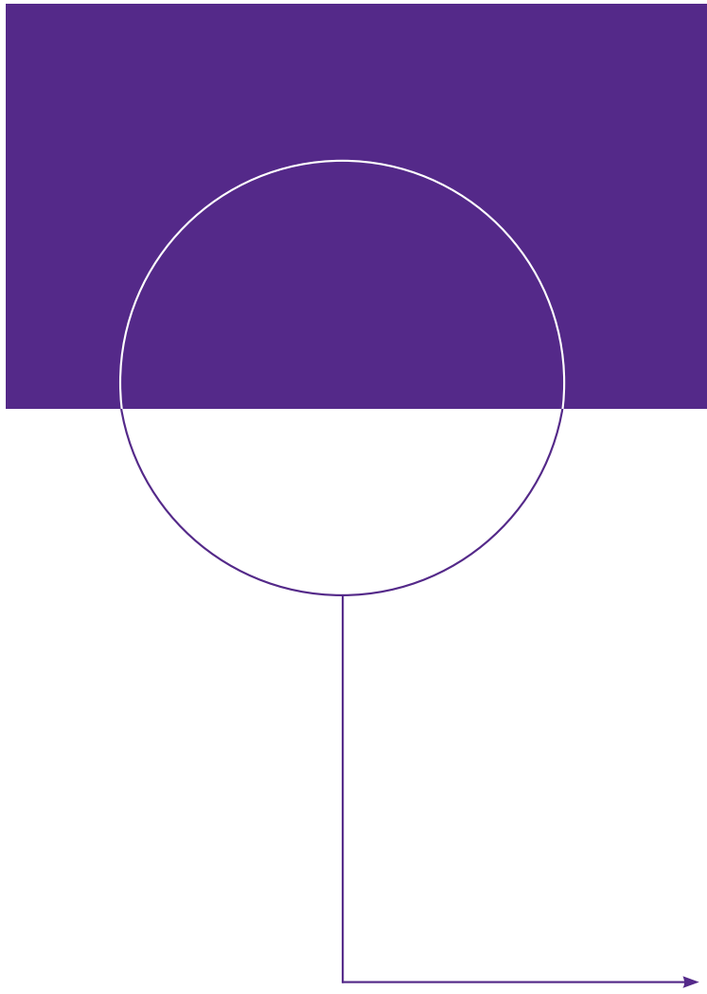


Master's thesis

2019

Simen Rogn Aune

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

Simen Rogn Aune

Development and Simulation of an Autonomous Docking System for Unmanned Surface Vehicles (USV)

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Development and Simulation of an Autonomous Docking System for Unmanned Surface Vehicles (USV)

Simen Rogn Aune

Cybernetics and Robotics
Submission date: June 2019
Supervisor: Kristin Ytterstad Pettersen
Co-supervisor: Aleksander Simonsen

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This master's thesis is given by Norwegian Defence Research Establishment (FFI) through Norwegian University of Science and Technology (NTNU) as a part of the 2 years masters degree program Cybernetics and Robotics education course. The master's thesis amount to 30 credits, and marks the end of the study program.

It requires that the solution formed is this master's thesis is developed in ROS. The solutions must be able to be integrated with FFI's current system. At the beginning of the task, two sets of data were received from FFI, which forms the basis leading up to the solution presented in the thesis. The data sets contain recorded navigational data of the USV docking a pier under normal circumstances, with information such as global position and velocity estimation included. They do not include data to create a map of the surrounding area. A simulator that creates 2D-maps of the surrounding area were created by myself. The project builds on a specialization project in which the concepts of an autonomous docking and departure problem is discussed. Chapter 1.4 Literature Review has been integrated to this report under *Previous Work*.

I would like to thank my supervisor Aleksander Simonsen from FFI for guidance throughout the semester. We had regularly talks where problems around the task were discussed. I am also thankful for the interesting task I had in hand. I have learned a lot this last semester, as well as being able to work on a larger project to create something. Also, thanks are rightfully given to the guys I had the ability to share office space with this semester. It has truly been fun working alongside them. When the motivation was low, we knew how to get going again.

Abstract

The interest in autonomous systems at sea has grown large in the last years. It is an exciting field of work, and the potential is huge. An Unmanned Surface Vehicle (USV) used as a minesweeper has potential. An important factor autonomy gives is removing any crew onboard. For a dangerous mission, autonomy can prevent humans from being injured. In the case of autonomous minesweeping, there is no risk of personnel onboard being injured, if a mine might be detonated.

The focus of this task lies in developing elements to such a vessel. A mission of a minesweeper may be divided into three separate actions: departure, out at sea, and docking. It is the departure and docking actions that will be worked with.

The task is concerning the USV that FFI has at its disposal. It is presented packages created in ROS that explains two separate problems that need to be addressed in order to one day be able to use the USV on a mission. The first problem is to detect when the USV has reached it's desired docking position along a pier. The second problem consists of creating a path for the USV from a starting position to the desired docking position. This constitutes to the docking action, which also can be used for departure.

Sammendrag

Interessen for autonome systemer til sjøs har vokst stort de siste årene. Det er et spennende arbeidsområde, og potensialet er stort. En USV brukt som minesveiper har potensial. En viktig faktor autonomi gir i dette tilfellet er å fjerne mennesker fra fartøyet. For et farlig oppdrag kan autonomi hindre at mennesker blir skadet. Ved autonom minesveiping er det ingen risiko for at personell ombord blir skadet, hvis en mine skulle bli detonert.

Fokuset på denne oppgaven ligger i å utvikle elementer til et slikt fartøy. Et oppdrag med en minesveiper kan deles inn i tre separate operasjoner: avgang, ute på sjøen og tilrettelegging inn til bryggen. Det er avgangs- og tilretteleggingsoperasjonen som vil bli jobbet med.

Oppgaven handler om en USV som FFI har til sin disposisjon. Det presenteres pakker opprettet i ROS som forklarer to separate problemer som må løses for en dag å kunne bruke USVen på et oppdrag. Det første problemet anngår å kunne oppdage når USVen har nådd den ønskede plassen langs en brygge. Det andre problemet består i å lage en bane for USVen fra en startposisjon til en ønsket posisjon langs bryggen. Dette utgjør tilretteleggingsproblemet, som også kan brukes til avgang.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Formulation

The title of this master's thesis is *Development and Simulation of an Autonomous Docking System for Unmanned Surface Vehicles*. The overall problem is self-explanatory from the title, but there are a couple of different tasks that lies within. FFI has a vessel called *Odin* which is used for research on autonomous operation of USVs. Currently, the USV is only autonomous at open waters, a distance away from land. Their vision is to also facilitate autonomous docking, the last event of an operation. An operation can generally be divided into three phases. Departure, out at sea, and docking.

Within the problem of this master's thesis, two subproblems appear. The first problem consists of retrieving information from the USV when it has reached it's desired docking position. Once a signal containing information about our arrival at the desired destination, *Odin* supports a "station-keeping-mode" which can be enabled.

The "station-keeping-mode" ensures that the vessel stays in its current geographical location. The second problem consists of planning a path from a starting position to the desired docking position. The path must be created such that is feasible for the vessel to follow it. The path has to be easy to follow, and not particularly demanding with respect to sharp turns, that causes unnecessary motions for the USV.

Figure 1.1 shows a pier located in Horten where FFI also has one of its offices located (building at the bottom is the office). USV *Odin* is located here. A solution should be designed such that it firstly works for that specific pier. Later, the possibilities of expanding the solution to work at any pier can be looked at.

Figure 1.1: Satellite Image over the Pier in Horten

1.1.1 USV Odin

Odin is a USV that FFI has created. Figure 1.2 displays the USV spoken about. The USV is equipped with a rotating LiDAR, which captures the distance to any object and land area in the vicinity of the vessel, precisely. It is also equipped with two HamiltonJet actuators at the stern of the vessel [1]. The actuators can rotate 27 degrees both ways in the way they have been mounted on the USV, with the initial position being parallel to the surge axis. Above a HamiltonJet actuator, there is a bucket mounted. The bucket can be lowered from totally above ground to under water. The buckets will reflect the water stream when the USV is in motion and can be used to extort forces in other direction that the rotation of the actuator can produce. The direction of the force depends on how much the bucket is lowered by. Using both the actuators, it is possible to combine the two forces created to manoeuvre the USV along the sway direction.

Figure 1.2: USV Odin

The USV is equipped with an Inertial Measurement Unit (IMU). Due to various circumstances, the acceleration is not available for readout by external components in the

present system that FFI has implemented. It is also equipped with a GPS, radar and the previously mentioned LiDAR. The components combined produces an estimate of the position and velocity, as well as a local map with obstacles in the surroundings of the USV.

1.2 Background

The author of this master's thesis originates from the two years master's degree program *Cybernetics and Robotics* at NTNU. The field of study gives a range of lessons from control systems theory to various courses in modelling, simulation, optimization and algorithm- and data-structures. The chosen specialization of the author is *Robot Engineering and Vessel Control Systems*, which was chosen immediately at the start of the two years master's.

1.2.1 Previous work

Research done prior in the field of autonomous vessels is located all around the world. Especially at NTNU there has been a lot of work on the subject. Thor I. Fossen has written a book entirely based on motion control of marine craft [2], and various other master theses have been written about autonomous vessels. `The Department of Engineering Cybernetics` and Norway as a country is a large partaker of the work done on the subject from a global perspective. The first electric and autonomous ship is to set sail in 2020 (Yara Birkeland), is developed in Norway by Yara and Kongsberg [3]. Bertram Volker presents a survey of USV with a historical perspective that tracks back to 1944 [4]. Most of the USV's are from the USA. In the 1950s the interest for the use of an autonomous vessel as a minesweeper or to any other dangerous mission grew, for "obvious reason" he wrote in the report.

Previous master theses from NTNU with the likes of *Autonomous Docking for Marine Vessels Using a Lidar and Proximity Sensors* [5] and *Guidance System for Autonomous Surface Vehicles* [6] have been of good help getting a grasp of various problems within autonomy at sea.

A variety of textbook has also been useful in understanding and presenting the theory needed.

1.3 Motivation

The report is written on behalf of FFI. FFI sent their problem description to NTNU, where it appeared on the list of master thesis projects. The USV is created with the hope that it one day can operate as a fully autonomous minesweeper. Having an USV that can operate autonomously, without the need of crew onboard to manoeuvre the vessel. With autonomy implemented, there is no risk of injuries among the crew onboard, should a mine suddenly be detonated. Safety is a huge factor. Currently, autonomous operations of *Odin* has been demonstrated where every aspect of the mission was handled by the vessel itself, except departure and docking from the pier.

1.4 Code

The code following this master's thesis can be found at `https://github.com/simenrau/master-project`

1.5 Outline

This report uses the *NTNU ITK thesis template* formed by Morten Fyhn Amundsen [7]. The outline of the master's thesis is as follows.

Chapter 2 - Theoretical Background explains the theories found useful in solving the problem. The explained theories should facilitate an easier understanding of the developed solution. When the actual work is presented, it should be possible to understand what has been done.

Chapter 3 - Detection Filter consists of the work leading up to the solution of the detection of docking. The chapter explains the solution in the sequence of events that are required to detect the docking. It corresponds to the first problem, as mentioned in the problem description.

Chapter 4 - Path Planner consists of the work leading up to the solution of planning a path from an initial starting point to the desired destination. Similarly to Chapter 3, the various events that lead up to the solution of the specific problem is explained. It corresponds to the second problem.

Chapter 5 - Discussion explains the solution from both of the previous two chapters, and the project in it's entirety. The solution is discussed, and any drawbacks to the solutions will be presented.

Chapter 6 - Conclusions includes a short summary of project. It also includes a section with future work.

Appendix A includes information about the software used, and how to install some of them.

Appendix B and C includes a structure tree of the packages created.

Chapter 2

Theoretical Background

The following chapter will contain an explanation of various topics and theories needed to understand in order to be able to follow the solutions presented in this Master thesis. They are listed in a manner such that theories presented later in the chapter can incorporate elements already presented.

2.1 Robot Operating System (ROS)

FFI utilizes the Robot Operating System (ROS) on it's USV. ROS is a tool that provides libraries to help developers to create robot applications. The robot, in this case, would be the USV.

Within the ROS environment it is possible to send messages from one node to another as topics. Any node can have any number of inputs and outputs, which another node

can subscribe or publish to. The subscribing node would then attach itself with a topic that the other node is publishing. A topic can contain any type of message. Figure 2.1 visualizes the process of publishing a topic to a subscribing node. '/Node_subscriber' can also publish a topic to any following node, or to any node other nodes available.

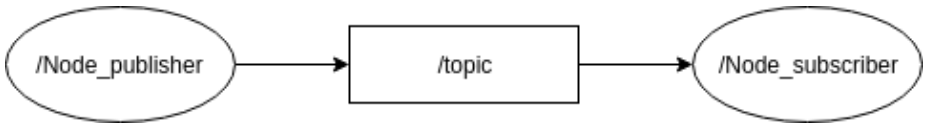


Figure 2.1: ROS Communication

To be able to use ROS, it has to be installed on a Linux computer. The preffered Operating System (OS) when using ROS is the Ubunutu distribution, where the official support lies. Ubuntu is a free and open-source Linux distribution. The process of how install Ubuntu with ROS is explained in Appendix A - Software. ROS can be integrated with both `Python` and `C++`.

2.1.1 ROS bags

There is an internal ROS file format for storing messages, which is called a *bag* file. The extension for this format is `.bag`. With a tool called `rosbag` it is possible to store, process, analyze and visualize a ROS bag. It can store multiple different messages, which a subscribing node can pick up. Similarly to Figure 2.1, `/Node_publisher` can in this case be replaced with a `/ROSbag`-node that in the same manner publishes topics to a subscribing node. Figure 2.2 visualizes this. It is a very neat tool, in which cases someone might want to simulate events based on the prerecorded data stored in a ROS bag.

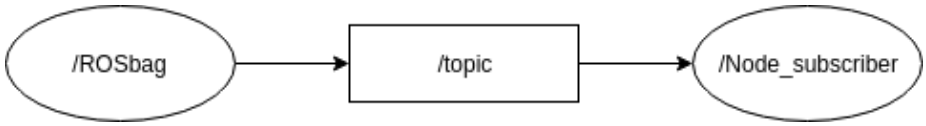


Figure 2.2: Subscribing to ROS bag

2.2 Moving Average Filtering

A moving average filter computes the average value over a specified number of points. Whilst the time-series of data is created, the moving average gets updated as fast as a new value is present. Take a look at Figure 2.3 to see how it turns out, visualized, for a random set of numbers. We observe the sliding window, which retrieves the newest value, and throws the oldest.

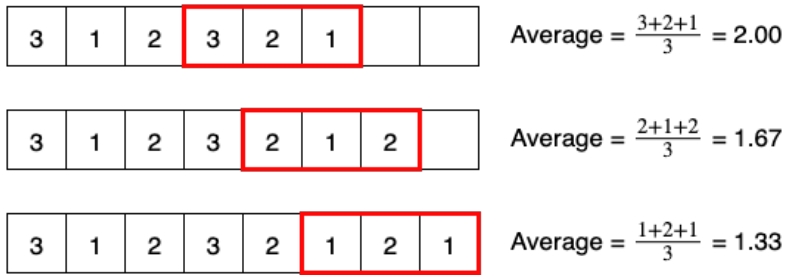


Figure 2.3: Moving Average Calculation

The example above counts for a simple moving average filter, as the average is not weighted in any way. The numbers in the current window are simply added together, and divided by the window size number (which in this small example is set to three).

Moving average is the most common digital signal processing filter because it is easy to understand and implement. The equation of the moving average is written as

$$\bar{y}_{MA}[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i-j] \tag{2.1}$$

where an arbitrary output in the case of $M = 3$ with the moving average would look like

$$\bar{y}_{MA}[5] = \frac{x[5] + x[4] + x[3]}{3} \tag{2.2}$$

The moving average filter can be used to smooth out any signal with short-term fluctuations, and to highlight longer-term trends.

2.2.1 Circular Array

A circular array can be used to store a specified number of values for any signal. The moving average filtering can continue until there are no values left, creating a very large array to store all the numbers read. The same idea can be transferred to a circular array, however, it only creates an array with the numbers of values we want to average for.

The modulus operator is used to get the next index of the array once a new value is present. The sequence number (seq) is equivalent to how many values that currently has been read, and is iterated by one every time a new value appears. After setting the array size of how many numbers we want to average for, the following formula is used to get the index for the next value.

$$Index = \text{Sequence Number} \,\%\, \text{Array Size} \tag{2.3}$$

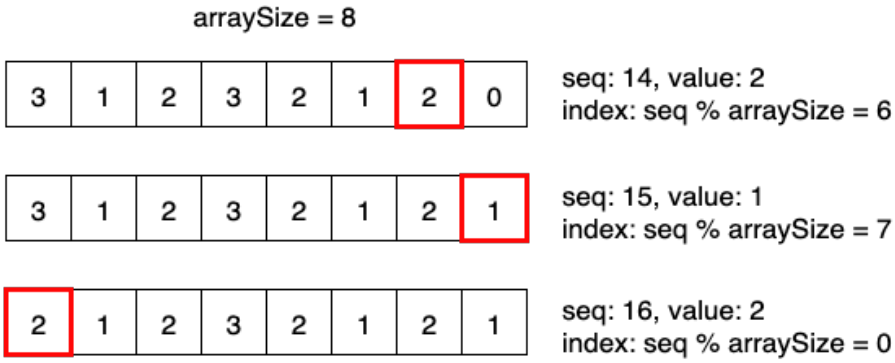


Figure 2.4: Circular Array

After a circular array has been made, the average value of the array can easily be calculated, as with (2.4) if the array is full, or with (2.5) if it is not full. Assuming the first value having a sequence number equal to zero, both equations are true.

$$\bar{y}_{CA}[i] = \frac{1}{M} \sum_{i=0}^{M-1} x_c[i] \tag{2.4}$$

$$\bar{y}_{CA}[seq] = \frac{1}{seq+1} \sum_{j=0}^{seq} x_c[j] \tag{2.5}$$

where avg_{CA} describes the average value for a circular array and x_c is a circular array.

2.3 A* Search Algorithm

The A* (read as A star) search is an algorithm that finds the minimum cost path from a starting point to a goal point by examining the nodes in between [8]. To explain the algorithm, the terms node and cost are used. A node can be explained as the index of a cell if a 2D array is used, while the distance travelled between two nodes is referred to as the cost travelling between the two nodes. The A* algorithm aims to create a path with the lowest cost (i.e. the shortest distance travelled). Eq. (2.6) displays the value of cost, denoted by $f(n)$ (n indicates the next node on the path). It is calculated by summing the two terms $g(n)$ (the cost from start to n) and $h(n)$ (heuristic function). The heuristic function $h(n)$ is the cost that is estimated from n to the goal node. A heuristic function is a function used to make an educated guess of which node to add next to the path, based on the estimated distance towards the goal node.

$$f(n) = g(n) + h(n) \tag{2.6}$$

For example, from the start node, the cost of each neighbour is calculated. The neighbour with the lowest cost is to be explored next. All the other nodes are added to a list, which we may call the *openList*, which includes the nodes that have been visited, but not evaluated. A *closedList* include the nodes visited and evaluated. We always look for the node with the lowest cost. For every new node evaluated, every cost of the neighbours of that node is being calculated. For every neighbour, the node evaluated is the predecessor.

The implementation of A* on a map to find the shortest path from a start- to the goal node, avoiding any obstacle presented on the map. A simplified form of a map is used in Figure 2.5. Any obstacle will be marked as an occupied cell, e.g. with a value corresponding to *1*. The free cells can be marked with *0*, and will be the cells that the algorithm can visit. The heuristic function $h(n)$ represents the straight-line distance

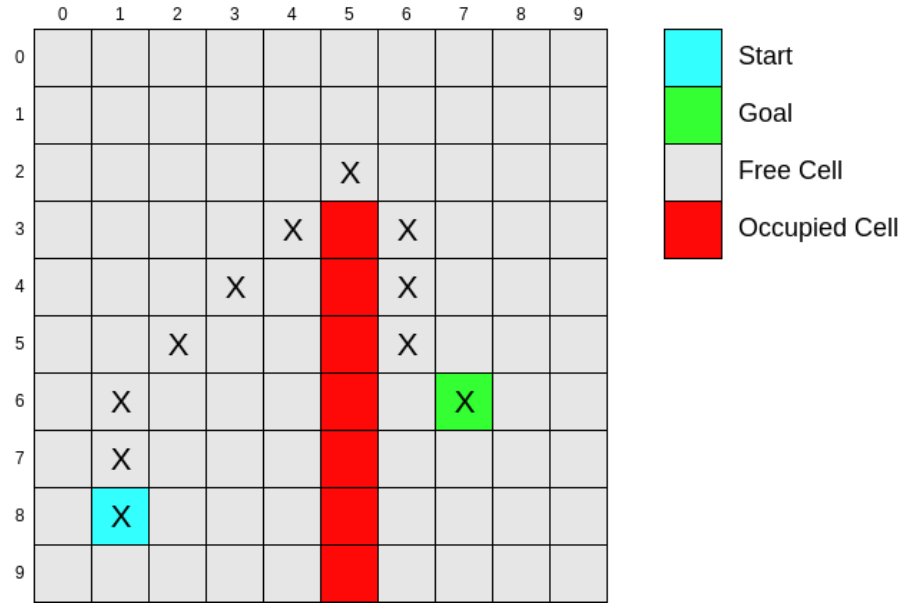


Figure 2.5: A* Search Example

from a node to the goal, while $g(n)$ equals the number of cells already included in the path before the current node. Eq. (2.7) shows the heuristic function $h(n)$, equal to the straight-line distance between node n and goal node.

$$h(n) = \sqrt{(n.x - goal.x)^2 + (n.y - goal.y)^2} \tag{2.7}$$

By examining the example from Figure 2.5, the algorithm is applied to a small case. The start cell is indicated by **(8,1)** and the goal cell is indicated by **(6,7)**. The red cells visualize occupied cells that cannot be used in our final path. The path is generated successfully, and using the A* algorithm a path is found to the goal cell while avoiding occupied cells. When arriving the at goal node, a path is being traced back to the start

following the predecessor of each node. This will produce the cells included in the path. From the figure, it is also important to note that a cell obliquely placed in relation to the current node is considered a neighbour.

Algorithm 1 - A* Search displays the pseudo-code of the A* algorithm, where n_s is the start node and n_{goal} is the goal node.

Algorithm 1 A* Search

```
1:  initialize the openList
2:  initialize the closedList
3:  n_s → openList
4:  while openList(!empty) do
5:      n ← lowest f(n) in openList
6:      if n == n_goal then
7:          terminate
8:      end if
9:      remove n from openList, and add to closedList
10:     generate each neighbour from n
11:     for each neighbour do
12:         if neighbour is in closedList then
13:             continue
14:         end if
15:         calculate g(n) and h(n)
16:         if neighbour is not in openList then
17:             add neighbour to openList
18:         else if f(n) ≥ f(neighbour) then
19:             update neighbour details
20:         end if
21:     end for
22: end while
```

2.4 Notation for Marine Vessels

Table 2.1 shows how we note motions for a marine vessel according to Society of Naval Architects and Marine Engineers (SNAME) [9]. The most important thing to remember is that we denote the x, y and z-direction with the surge, sway and heave, and the rotation about the surge, sway and heave axis with roll (ϕ), pitch (θ) and yaw (ψ). These notations hold for all maritime vessels.

DOF		Forces and moments	Linear and angular velocities	Positions and Euler angle
1	motions in the x-direction (surge)	X	u	x
2	motions in the y-direction (sway)	Y	v	y
3	motions in the z-direction (heave)	Z	w	z
4	rotation about the x-axis (roll)	K	p	ϕ
5	rotation about the y-axis (pitch)	M	q	θ
6	rotation about the z-axis (yaw)	N	r	ψ

Table 2.1: The notation of SNAME (1950) for marine vessels

2.5 Reference Frames

To be able to represent and analyze the motions of a marine craft with 6 degrees of freedom (surge, sway, heave, roll, pitch and yaw), it is convenient to define a couple of reference frame. In this task, we will use the NED (North-East-Down) reference frame, and the BODY reference frame.

2.5.1 NED-frame

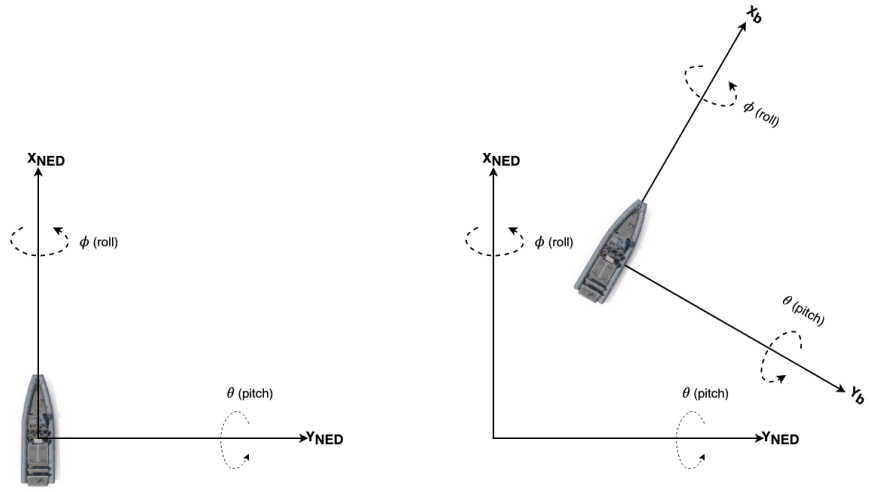
The NED-frame is a coordinate system where the origin is defined relative to the Earth's reference ellipsoid. This is the frame that is used most in our everyday life [2]. When dealing with vessels that are enclosed by travelling at small distances, the NED-frame is a fair representation, without the need for compensation of the Earth's curvature. The x-axis points to the true *North*, while the y-axis points to towards *East*. There is also an axis pointing *Down*, towards the Earth's centre. We use latitude and longitude as a measure of position. We may neglect the elevation (height), as we are dealing with marine vessels little affected by large waves. *Flat Earth Navigation* can be used for a marine craft that operates in a local area, with approximately constant global positioning. Therefore we can assume the NED-frame to be inertial, thus Newton's law will still apply.

Figure 2.6a shows a vessel located at the origin of a NED-frame. The origin we define for ourselves.

2.5.2 BODY-frame

The BODY-frame is used to represent the origin of a moving coordinate frame, where the origin is fixed to a vessel. The position and orientation of the vessel are described relative to the NED-frame. As mentioned earlier, the NED-frame operates as an inertial reference frame, since the global position of the craft is approximately constant.

Figure 2.6b visualizes the BODY-frame with respect to the NED-frame. We notice the vessel shown in the figure has a rotation of yaw, while the position of the vessel has shifted north-east.



(a) NED-frame (b) Body-frame relative to NED

Figure 2.6: 2D-representation of reference-frames for a marine vessel

2.6 Euler Angle Representation

Euler angles are used to represent a rigid body in a three-dimensional space. Given the angles roll (ϕ), pitch (θ) and yaw (ψ), a representation is obtained by the three principal rotation matrices

$$R_{x,\phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \tag{2.8}$$

$$R_{y,\theta} = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{2.9}$$

$$R_{z,\psi} = \begin{bmatrix} cos(\phi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Multiplying the three principal rotations yields the rotation matrix of the rigid body.

$$R_n^b(\Theta_{nb}) = R_{x,\phi} R_{y,\theta} R_{z,\psi} \tag{2.11a}$$

$$= \begin{bmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi c\phi s\theta \\ s\psi c\theta & c\psi c\phi + s\phi s\theta s\psi & -c\psi s\phi + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \tag{2.11b}$$

where $s \cdot = sin(\cdot)$ and $c \cdot = cos(\cdot)$, for simplification purposes.

2.7 Unit Quaternions

To avoid singularity of the Euler angles, quaternions if often used and applied to mechanics in the three-dimensional space. Quaternion is a four-component representation of a 3D rotation based on unit quaternion [10]. A quaternion $\boldsymbol{q}$ is defined as a complex number with one real part η and three imaginary parts given by the vector

$$\epsilon = [\epsilon_1, \epsilon_2, \epsilon_3]^T \tag{2.12}$$

A unit quaternion satisfies $q^T q = 1$. The set Q of an unit quaternion is therefore defined as

$$Q := \{q | q^T q = 1, q = [\eta, \epsilon^T]^T, \epsilon \in R^3 \; and \; \eta \in R\} \tag{2.13}$$

Using the following equation, as described in (Fossen, 2011) [2], the rotation matrix of the unit quaternion can be obtained

$$R_b^n(q) := I_{3x3} + 2\eta S(\epsilon) + 2S^2(\epsilon) \tag{2.14}$$

$$R_n^b(q) = \begin{bmatrix} 1 - 2(\epsilon_2^2 + \epsilon_3^2) & 2(\epsilon_1\epsilon_2 - \epsilon_3\eta) & 2(\epsilon_1\epsilon_3 + \epsilon_2\eta) \\ 2(\epsilon_1\epsilon_2 + \epsilon_3\eta) & 1 - 2(\epsilon_1^2 + \epsilon_3^2) & 2(\epsilon_2\epsilon_3 - \epsilon_1\eta) \\ 2(\epsilon_1\epsilon_3 - \epsilon_2\eta) & 2(\epsilon_2\epsilon_3 + \epsilon_1\eta) & 1 - 2(\epsilon_1^2 + \epsilon_2^2) \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{2.15}$$

2.7.1 Quaternion to Euler Angles Conversion

By comparing the rotation matrices of the two representation of a rigid body in a fixed coordinate frame, we can acquire the Euler Angles when knowing the quaternions of the system.

By the following equation,

$$R_n^b(\Theta_{nb}) := R_n^b(q) \tag{2.16}$$

we can obtain the Euler Angles by comparison,

$$\begin{bmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi c\phi s\theta \\ s\psi c\theta & c\psi c\phi + s\phi s\theta s\psi & -c\psi s\phi + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{2.17}$$

will result in

$$\phi = atan2(R_{32}, R_{33}) \tag{2.18a}$$

$$\theta = -sin^{-1}(R_{31}) = -tan^{-1}\left(\frac{R_{31}}{\sqrt{1 - R_{31}^2}}\right); \quad \theta \neq \pm 90^\circ \tag{2.18b}$$

$$\psi = atan2(R_{21}, R_{11}) \tag{2.18c}$$

where

$$R_{11} = 1 - 2(\epsilon_2^2 + \epsilon_3^2) \tag{2.19a}$$
$$R_{21} = 2(\epsilon_1\epsilon_2 + \epsilon_3\eta) \tag{2.19b}$$
$$R_{31} = 2(\epsilon_1\epsilon_3 - \epsilon_2\eta) \tag{2.19c}$$
$$R_{32} = 2(\epsilon_2\epsilon_3 + \epsilon_1\eta) \tag{2.19d}$$
$$R_{33} = 1 - 2(\epsilon_1^2 + \epsilon_2^2) \tag{2.19e}$$

2.8 Low-Pass Filter

In signal processing, a filter is a process that removes unwanted features from a signal. Filtering is a class of signal processing and is often related to removing some frequencies or frequency bands.

A low-pass filter is a filter that passes signals with a frequency lower than a given threshold and that eliminates signals of high frequencies [11]. A simple low-pass filter consists of a capacitor and a resistor. The capacitor's impedance increases with decreasing frequency. Low impedance blocks the high-frequency signals, while high impedance passes through low-frequency signals. This kind of filter is often used in practice to suppress unwanted noise from a signal.

$$y[0] = \alpha \cdot x[0] \tag{2.20a}$$
$$y[i] = \alpha \cdot x[i] + (1 - \alpha) \cdot y[i - 1] \tag{2.20b}$$

Digitally, an algorithm can be implemented to achieve the same functionality as the physical filter. Equation (2.20) can be implemented in a digital solution for any signal, where i: iteration number, x: signal to be filtered, α: discrete-time smoothing parameter

2.9 Geographical Coordinate Calculation

This section presents a variety of different calculations for latitude/longitude points. The calculations consider a spherical earth, even though the earth is slightly ellipsoidal. The radius of the earth is approximately equal to $R = 6371 \times 10^3 m$ and used in the formulas to come.

2.9.1 Calculation Distance Between Lat/Lon Points

With two points of latitude and longitude positions, we can calculate the distance in meters between them, by using the following equations. These formulas can be implemented in `C++`. The formula is also known as the *haversine* formula, used to calculate the great-circle distance between two point [12].

In the following equations; ϕ - latitude, λ - longitude, d - distance between latitude/-

longitude points.

$$\Delta\phi_{rad} = (\phi_2 - \phi_1) \cdot \frac{\pi}{180} \tag{2.21a}$$

$$\Delta\lambda_{rad} = (\lambda_2 - \lambda_1) \cdot \frac{\pi}{180} \tag{2.21b}$$

$$a = \sin^2(\frac{\Delta\phi_{rad}}{2}) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2(\frac{\Delta\lambda_{rad}}{2}) \tag{2.21c}$$

$$d = R \cdot 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a}) \tag{2.21d}$$

2.9.2 Calculation Displacement from Lat/Lon

Given an initial lat/lon-position, a bearing angle and the distance to the new position, one can calculate the position of a new lat/lon-position using the following formulas; where β - bearing angle in radians, and the rest of the variables as in the previous subsection.

$$\beta = \alpha \cdot \frac{\pi}{180} \tag{2.22a}$$

$$\phi_{1,rad} = \phi_1 \cdot \frac{\pi}{180} \tag{2.22b}$$

$$\lambda_{1,rad} = \lambda_1 \cdot \frac{\pi}{180} \tag{2.22c}$$

$$\phi_{2,rad} = \arcsin\left(\sin\left(\phi_{1,rad}\right) \cdot \cos\left(\frac{\beta}{R}\right) + \cos\left(\phi_{1,rad}\right) \cdot \sin\left(\frac{d}{R}\right) \cdot \cos\left(\beta\right)\right) \tag{2.22d}$$

$$\lambda_{2,rad} = \lambda_{1,rad} + \text{atan2}(\sin\left(\beta\right) \cdot \sin\left(\frac{d}{R}\right) \cdot \cos\left(\phi_{1,rad}\right), \tag{2.22e}$$

$$\cos\left(\frac{d}{R}\right) - \sin\left(\phi_{1,rad}\right) \cdot \phi_{2,rad}) \tag{2.22f}$$

$$\phi_2 = \phi_{2,rad} \cdot \frac{180}{\pi} \tag{2.22g}$$

$$\lambda_2 = \lambda_{2,rad} \cdot \frac{180}{\pi} \tag{2.22h}$$

2.9.3 Calculating Bearing Angle

With the knowledge of two different lat/lon points, it is possible to calculate the bearing angle of the line drawn between them. Consider 2.7. The example illustrates which angle is the bearing angle for two lines, that is drawn between two lat/lon points.

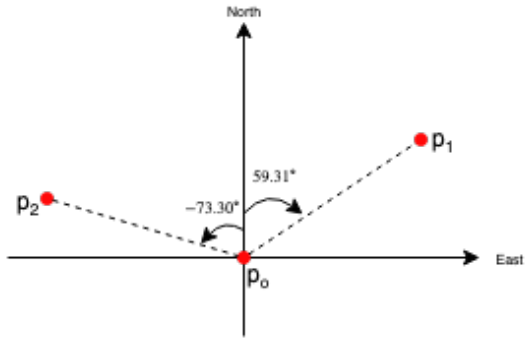


Figure 2.7: Bearing Angle

The following formulas lead to finding the bearing angle.

$$x = \cos(\phi_{1,rad}) \cdot \sin(\phi_{2,rad}) - \tag{2.23a}$$

$$\sin(\phi_{1,rad}) \cdot \cos(\phi_{2,rad}) \cdot \cos(\lambda_{2,rad} - \lambda_{1,rad}) \tag{2.23b}$$

$$y = \sin(\lambda_{2,rad} - \lambda_{1,rad}) \cdot \cos(\phi_{2,rad}) \tag{2.23c}$$

$$\beta = \text{atan2}(y, x) \cdot \frac{180}{\pi} \tag{2.23d}$$

Chapter 3

Detection Filter

3.1 Introduction

In this chapter, there will be a full description of how the `Detection Filter` package was created. The package was created in order to receive information when the USV has reached the designated docking position along the pier. Firstly, the task was all about achieving the explained information for the specific case that is the pier in Horten, where FFI has one of its offices, and also where the actual USV is located. The solution to be presented is able to produce a signal which indicates if the USV has made contact with the pier. It is important to mention that the USV will dock with an orientation parallel to the pier. It will slide into place, with velocity directly in the sway direction.

It was received two separate ROS bags which contained real-time data of the USV in motion, which FFI had captured. With the recorded data it is possible to simulate

the environment without being at the fixed location of the USV. The bags will be mentioned as *USV1.bag* and *USV2.bag* further in the report. As previously mentioned in 2.1 - Robot Operating System (ROS) and 2.1.1 - ROS bags, the tool used to implement the solution for this USV is called ROS. For developing code in ROS, `C++` was used as the programming language.

We are interested in a couple of specific signals, which are located and stored as topics in the ROS bag. We are going to create a ROS-node that can be integrated with FFI's current solution. To create a node, the identification of wanted inputs that will be required to generate the wanted output is required. The output of interest that this package will produce is a *Boolean* signal. The signal should be high (equal to *1*) when the conditions set for it to be high are fulfilled. The flag will be high when there has been detected contact with the pier.

3.1.1 Contents of ROS bags

The ROS bags contains the following information and topic of interest:

USV1.bag	
Duration:	198s
Topics:	/usv/odometry_ins_ned (19801 msgs)

USV2.bag	
Duration:	196s
Topics:	/usv/odometry_ins_ned (19679 msgs)

Within the topic of */usv/odometry/ins_ned* it exists two messages of interest, `pose` and `twist`. Both these messages are defined in the ROS environment. The `pose` message contains the position in x, y and z-direction and orientation in quaternions, while `twist` contains velocity and angular velocity (in body frame).

The two bags both includes a start at a distance away from the pier in Horten, at open-waters, as well as the USV being manoeuvred to a docking position at the pier. This means that the both of the bags contain the sequence of the vessel making contact with the pier, which we obviously need in order to create a package that can detect whether the USV has reached it's desired docking position. We are going to utilize this in the creation of the following package. In section 3.5 - Simulation and Results it is explained further how the bags were used together with the created package to simulate the process.

3.2 Idea

The idea is to use the velocity-data to generate an estimate of the acceleration, as the acceleration is the time derivative of the velocity. With the acceleration, it is possible to detect an abnormality in the signal when making contact with the pier. The acceleration prior to the contact with the pier should be relatively flat, as the USV slides to the destination.

As illustrated in figure 3.1, the input to the package will be the position, orientation and velocity of the USV. With that data, it will be possible to estimate the acceleration in the horizontal sway direction. Within the package, we will use the estimated horizontal acceleration as an element of triggering a flag high to indicate if the USV has docked. This flag will be the output.

It is also desired to manually enter the destination when wanted to dock. Whenever

the destination is set, we can calculate the distance from the current position to the set destination. The distance between the position can be calculated in meters, as explained in Section 2.9. By continuously calculating the distance between the current position of the USV and the set destination, another condition of setting the flag high can be added. For example, this condition can return true when the USV is closer than a set distance threshold. By adding the said condition, we can prevent the flag from being trigger at distances far away from the desired docking position. There could, for instance, occur events which cause the acceleration in the sway direction to behave similarly to when there has been made contact with the pier, a distance that is not the set destination.

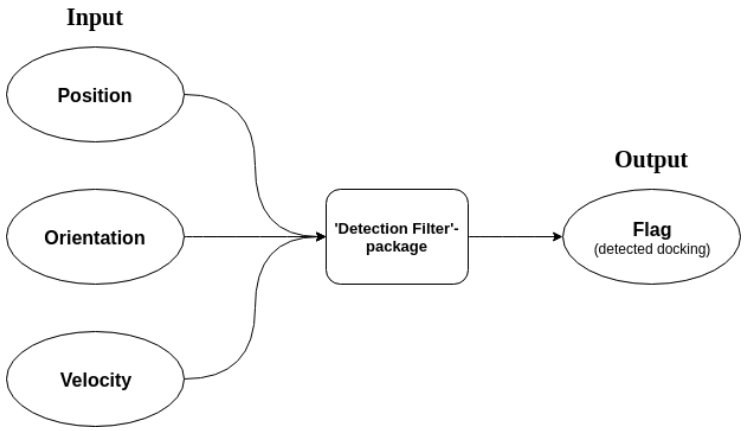


Figure 3.1: I/O of `Detection Filter`

3.3 Development

The task was to create a ROS package which handles the detection of docking, such that it can seamlessly be fitted into FFI's environment when finished. As previously mentioned, the package is written in `C++`. The data contained in the ROS bags were

collected whilst the USV was navigated around the area of the pier in Horten. Both bags include the event of docking, which is vital in understanding how to create the signal to detect docking.

3.3.1 Compensating Roll

It is normal for a ship to sway. The roll of any vessel at sea may vary a lot, heavily depending on the weather located at the scene. In our case, it will most likely not affect the USV very much. Nevertheless, it is important the compensate for it, because the result will be better. FFI are using quaternions to represent the USV in the three-dimensional space. To be able to compensate for the roll-angle, a conversion from quaternions to Euler angles was necessary. Using equation 2.18a - 2.18c it is possible to obtain the Euler angles when the quaternions are known.

Figure 3.2 simply explains which velocity vector we are interested in using. The figure is made with a large roll angle for illustration purposes only. The horizontal velocity is the acceleration vector of interest.

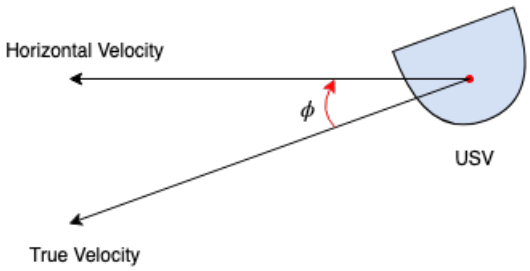


Figure 3.2: Velocity Vectors

After converting the quaternions to Euler angles, which is illustrated by figure 3.3. Figure 3.4 illustrates how the acceleration in the sway-direction turns out, with and without compensating for the roll. The extracted sample is from `USV1.bag`, with the

time period of the figure being chosen to include some movement in roll. The roll angle is fairly small throughout the samples of data, and as seen from the figure. The velocity in the roll-direction is approximately the same as the horizontal velocity. This ultimately provides two similar signals of the sway-velocity with and without roll compensation. Nevertheless, there might occur instances where the roll would be larger than in the two prerecorded bags used to simulate. In those cases, the compensating of roll could be much needed.

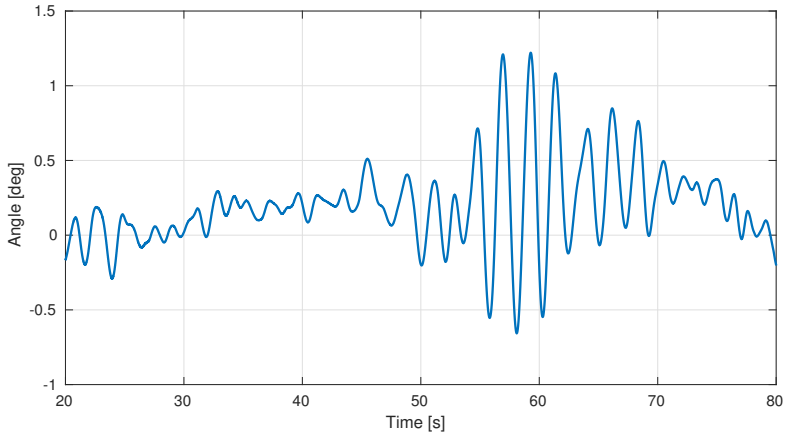


Figure 3.3: Calculated Roll Angle (from USV1.bag)

3.3.2 Estimating and Filtering Acceleration

After obtaining the horizontal velocity vector, the next procedure was to estimate an acceleration. With the acceleration in place, it will be possible to detect abnormalities in the signal when the USV makes contact the pier. The pose of the USV leading up to the pier is always slow and steady. Therefore, the acceleration will be near constant and averaging close to zero. At the moment of impact, a spike should be detected.

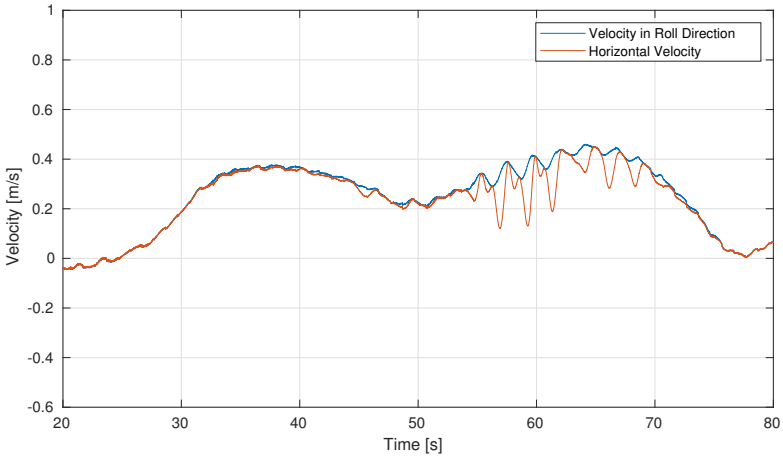


Figure 3.4: Comparing Velocity in Roll- and Horizontal Direction (from USV1.bag)

To make an estimate of the acceleration, which is the time derivative of the velocity, was obtained by the two variables being the time difference between two instances of velocity, and the actual difference in velocity.

$$a_y[i] = \frac{\Delta v_y}{\Delta t} = \frac{v_y[i] - v_y[i-1]}{t[i] - t[i-1]} \tag{3.1}$$

Figure 3.6 visualizes the estimated acceleration of the USV for both of `USV1.bag` and `USV2.bag` for their whole bag-length. It is clearly shown that the horizontal acceleration is quite noisy. The signal had to be filtered for it to be interpreted more accurately. A low-pass filter was applied in order to extract a better signal with the following formula for every instance new information is received by the USV.

Eq. (3.2) shows the low-pass filtering equation looks like. For the very first instance of the calculation, (3.2a) initializes the calculation of the filtered horizontal sway velocity.

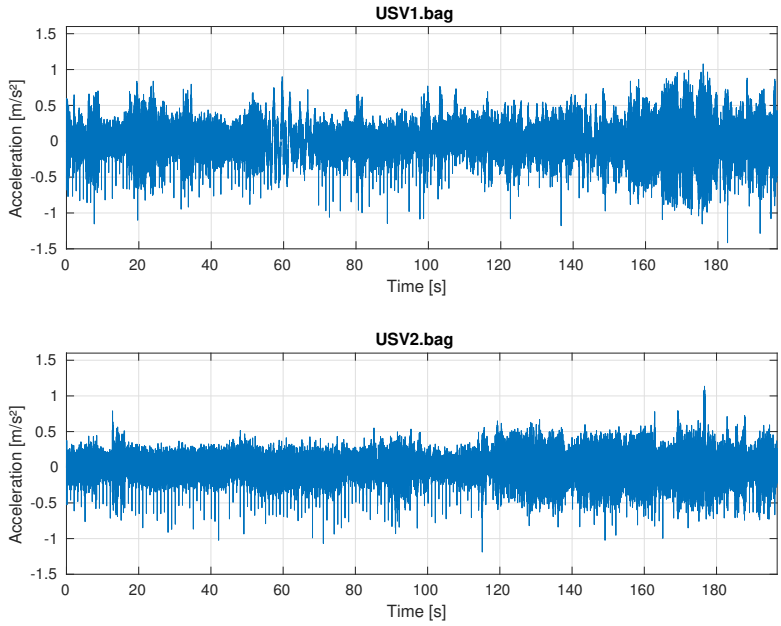


Figure 3.5: Acceleration in Sway Direction

Eq. (3.2b) is used for every calculation after the initial, where it uses the previous calculation to calculate the current. From Section 2.8 the details of how the low-pass filter operates were explained. Figure 3.5 shows how the estimated acceleration turns out after being calculated with the equations.

$$a_{y,LP}[i] = t[i] \cdot a_y \tag{3.2a}$$

$$a_{y,LP}[i] = t[i] \cdot a_y + (1 - t[i]) \cdot a_{y,LP}[i-1] \tag{3.2b}$$

After the signal is filtered, it became much easier to understand it's behaviour. Without any knowledge on the data beforehand, a good initial guess would be that the USV makes contact with the pier at the end of both bags, after looking at the filtered signal. The acceleration before and after the spike, both have a steady acceleration. The USV will have a slow and steady paced entry in the final section before arriving the destination. In the final phase, the velocity would approximately be constant, where the time derivative of the velocity, the acceleration, also being constant. Therefore, we now can observe an abnormality in the signal when the USV makes contact with the pier.

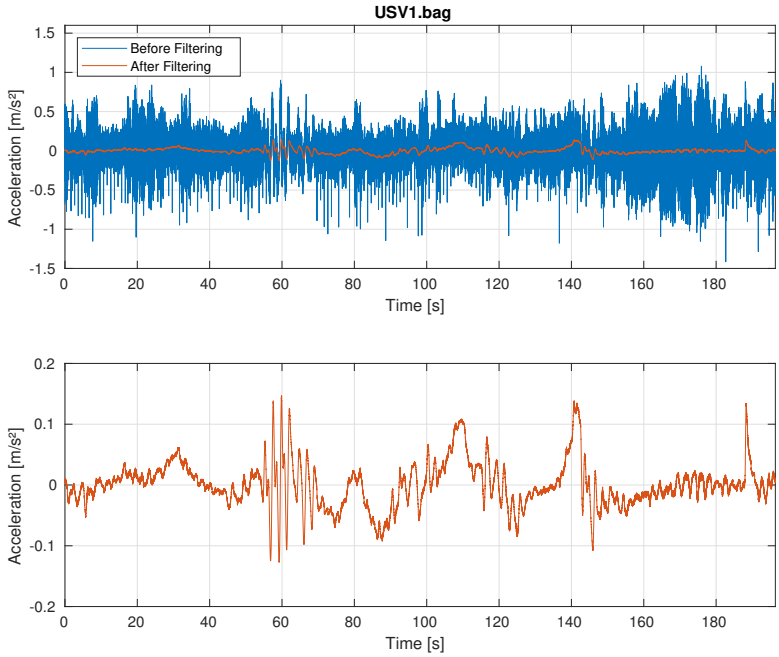


Figure 3.6: Acceleration in Sway Direction Before and After Filtering (from USV1.bag))

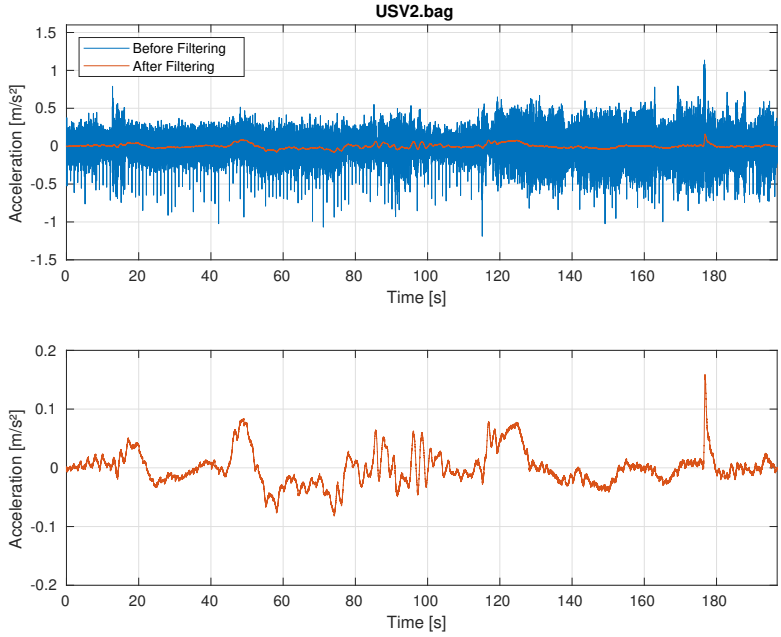


Figure 3.7: Acceleration in Sway Direction Before and After Filtering (from USV2.bag))

3.3.3 Average Calculation

With the filtered signal, we need to create another signal. This new signal is filtered acceleration averaged for a specified number of values. This specified number of values is called the `windowSize`. The value of `windowSize` was set to 20, which means that the average is calculated with the last 20 values of the filtered signal. A circular array was used to store these values. Keeping track of the sequence number, the circular array uses the modulus operator to make sure that the new value is assigned to the next index in the array. Once the array is full, it starts rewriting from the front of the array again. That is where the modulus operator come in handy. For example, the

current sequence number is 33. With the following formula `sequence % windowSize` = 33 % 20 = 13, which now tells us that the index of 13 in the array will be overwritten by the current value. Section 2.2 describes the approach.

The reason why we need this averaged acceleration signal is easy to explain. To be able to detect an abnormality in the filtered acceleration, the averaged signal can tell us when there has been a sudden change of value over a short period of time. Since an averaged signal will not be prone to sudden changes as large as the original one. The averaged signal can be compared to the filtered signal to detect a sudden change. A constant term must be added to the averaged signal. We want the signal to be informed of a larger change. The constant term added is equal to 0.07. It is a reasonable choice. This reason will be clearer in Section 3.5 - Simulation and Results.

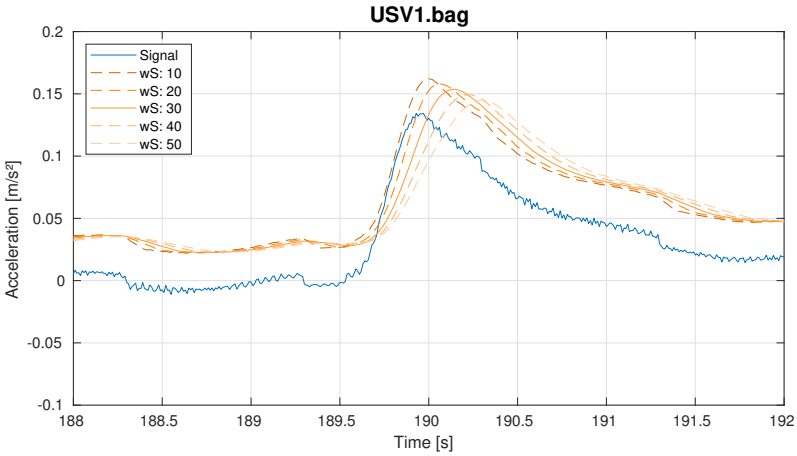


Figure 3.8: Comparing for Various Values of `windowSize` (from USV1.bag)

Figure 3.8 and 3.9 shows how the averaged signal is created for a various instances of the `windowSize` parameter. We wish to use a circular array that store the amount set to `windowSize` for the better performance to detect a spike. Looking at the figure, `windowSize` set equal to store 30 values seems like a reasonable fit after testing with both bags. The plot highlights the event of the USV making contact with the pier.

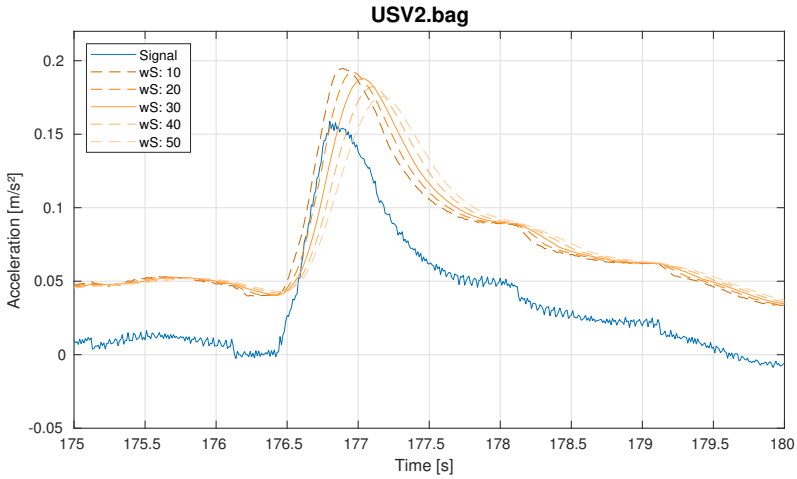


Figure 3.9: Comparing for Various Values of `windowSize` (from USV2.bag)

Why the averaged filtered signal is needed will be explained in the next Section - Implementation.

3.3.4 Desired Destination

It is wished that an operator of the system can be able to input the desired docking destination. The operator should be able to configure the destination when that is wanted. Once the destination is set, the distance from the USV's current position and destination is continuously calculated. Together with the spike, a destination threshold is set. A *Boolean* flag is set when the conditions of a spike are met, where the USV being closer than the threshold value is one of them. This flag can later be used to activate the *station-keeping-mode* already available on *Odin*. Figure 3.10 show how the destination can be set. Once the program is launched, the window pops up, allowing us to set the desired destination. The destination that has been set is saved and used

in our calculations towards docking detection.

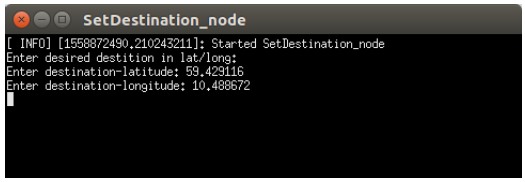


Figure 3.10: `SetDestination_node` Terminal Window

3.3.5 Calculating Distance

From the previous figures with the filtered horizontal acceleration, we can not for sure avoid the possibility of the flag being triggered by events that do not include the USV making contact with the pier. Another condition can be added to the system. This new condition is to continuously measure the distance from the current position of the USV to the set destination. The current- and destination position both consist of a latitude/longitude point. From Section 2.9.1, the distance calculation between two lat/lon points were explained. The distance is continuously being calculated every time the USV's position is updated. Figure 3.11 shows the distance between current USV position and the set destination. By looking at which lat/lon position both of the bags had in the end, this was the destination that was inputted in order to visualize the distance away from the destination for the entire recording. Before the destination has been set, the distance is not available (it is equal to a very high number). After `USV1.bag` had played for 20 seconds, the destination was inputted, where `USV2.bag` had played for about 35 seconds. This is visualized in the figure.

One of the conditions for the flag to return *true* will be whenever the distance between the current- and destination position is less than a pre-set threshold. In our case, this threshold was set to *7* meters. If the USV is closer or equal to a distance of *7* meters to the desired destination, the condition returns *true*. The reasoning behind the choice of

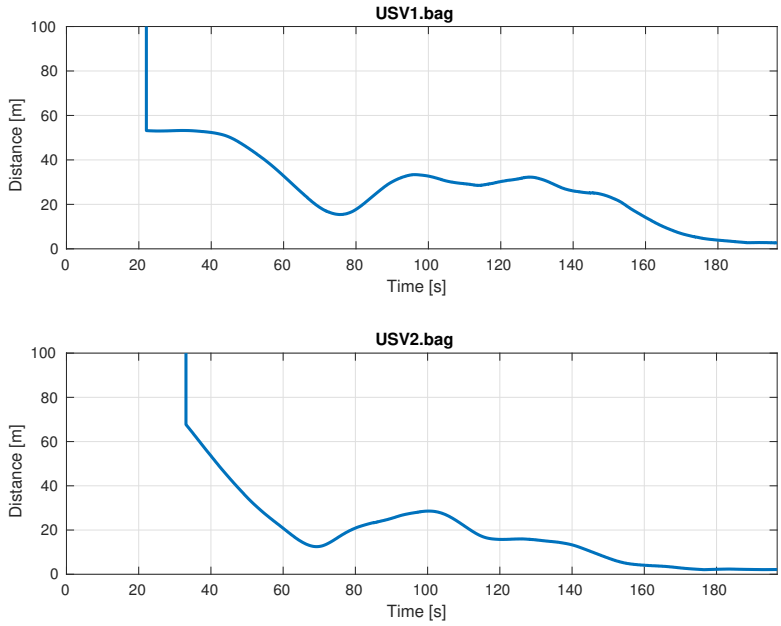


Figure 3.11: Calculation Distance to Desired Destination

7 meters is because there lies uncertainty with GPS measurements. Therefore, this is a reasonable choice, since the USV should have a composed entry.

3.4 Implementation

In this solution, it is created a separate node for setting the destination. In the node creating the */Flag*-topic, the procedure of generating it will not start as long as the destination is not set. Once the destination has a value, the procedure will be activated.

Figure 3.12 illustrates how the created nodes communicate with each other. The main node where basically every calculation occurs is in `/DetectionFilter_node`, which subscribes to two nodes (`/SetDestination_node` and `/USV1bag`), while it publishes to one node called `/ReadFlag_node`. It receives the set destination once it is set by the operator, as well as data from the USV (in this case data stored in a ROS bag). The input data is processed to produce the output, that is a flag informing of arrival at the pier. The `/ReadFlag_node` is added to mimic node which would have an interest in the topic `/flag`. This could, for instance, be a new node for activation of the *station-keeping* mode.

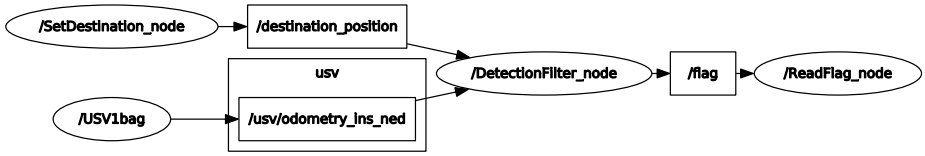


Figure 3.12: Communication Between ROS-Nodes

By calculating the moving average of the filtered acceleration in the horizontal direction that previously has been obtained, we can compare the averaged data-series with the original. The comparison is used to make sure that there really has occurred a spike in the acceleration. The reason is, being that the moving average is slower to reach a higher value and thus ignoring short-term signal oscillation or other small changes of acceleration. The constant term of 0.07 that has been added makes sure that

For the two recorded bags, it is tested and learned what the normal value a contact with the pier has. Therefore, we can add a constant term to the average acceleration to be even more sure of a spike. Then we create a condition telling the system to create a spike only if the actual filtered acceleration is above the averaged acceleration with the constant term. Together with the distance condition to be true, a spike is triggered.

3.5 Simulation and Results

A simulation of the *DetectionFilter* is performed after creating a launch file of the nodes that the package contains. The launch file includes the three nodes creates, as well as playing the ROS bag. As the launch file is executed, the window to set the destination pops up (fig 3.10). When the destination has been set, continuous calculations take place in the background.

Figure 3.13 shows a 6x1 plot of the events of interest in being able to detect docking, where the last three plots are zoomed in on the events of interest. The first plot contains the calculated distance away from the destination, together with the distance threshold. It is the distance between the USV's current position and the set destination that is being calculated. The second plot contains the filtered acceleration together with the averaged filtered acceleration raised by a constant value. The final and third plot contains the topic */flag*, which is created by the two plots above. This topic is the output of the package and could be published to any other nodes that may need this data.

There are, as mentioned earlier, two conditions needed to return *true* in order to create a spike. If the distance to the destination is lower than 7 meters, `AND`, the filtered acceleration is above the averaged filtered signal. In both conditions are met, a spike is created. From the figure, the distance to the destination is closer than 7 meters at approximately 170 seconds and forward. The filtered signal is slightly above the averaged filtered signal at approximately 188 seconds. This means that a spike is created at the instance of the last condition to be met.

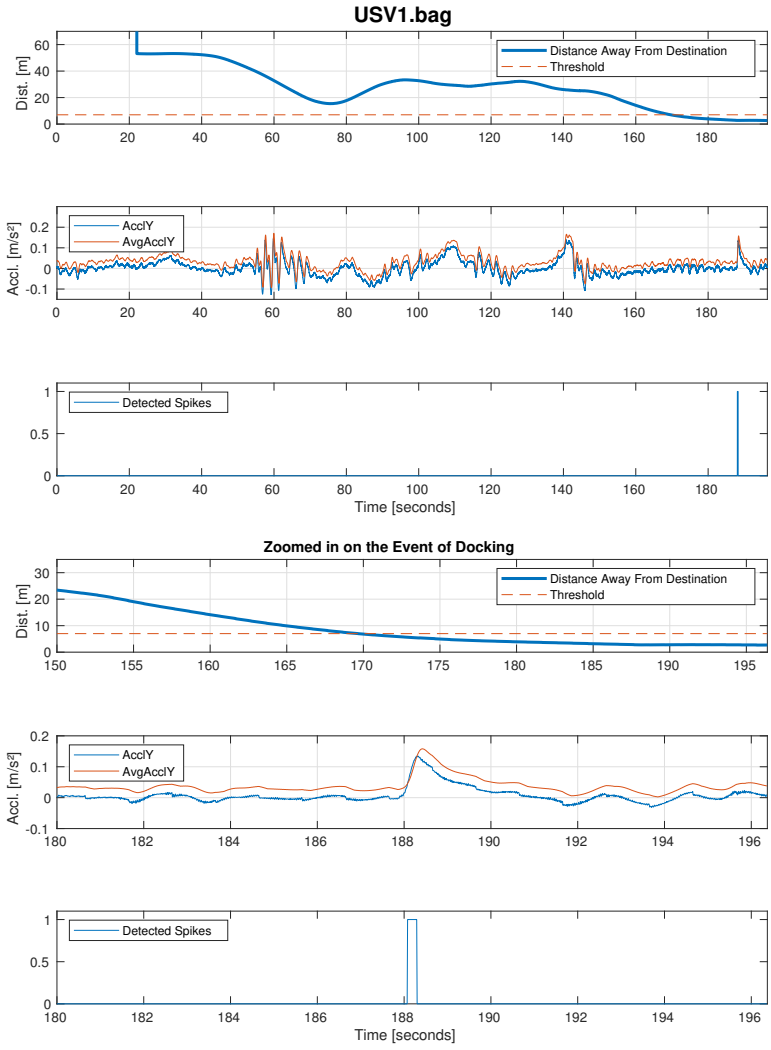


Figure 3.13: Detection Filter Package (from USV1.bag)

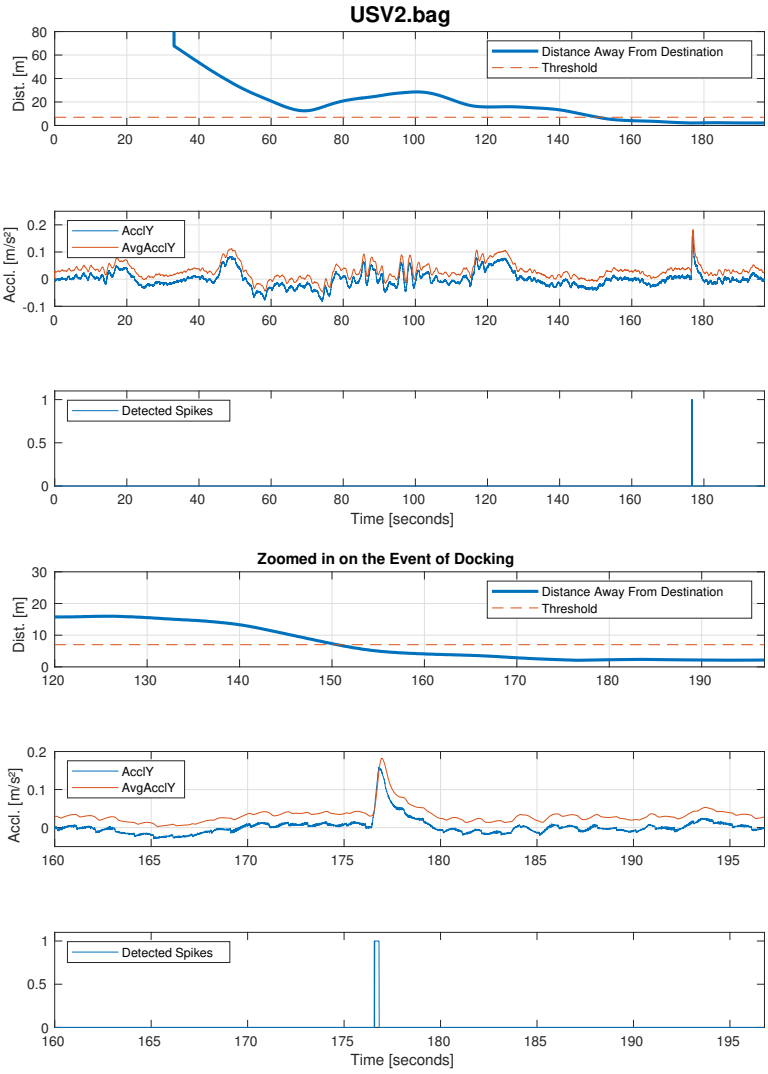


Figure 3.14: Detection Filter Package (from USV2.bag)

Chapter 4

Path Planner

4.1 Introduction

This package will handle any path from a starting point (the USV position when planning to dock) to the desired destination (which is the position at the pier where it is wished that the USV will park). The package is also written in `C++`, as the previous `Detection Filter`-package. There have been created multiple nodes, which serves its own purpose in ultimately obtaining the path from start to destination. The algorithm used to create the path is the A* search algorithm.

Since FFI has the ability of mapping it's surrounding area, the focus of this package lies on creating a path for the USV to follow in the last phase of an operation. Therefore, this package will function as a simulator of testing the A* algorithm, in the search of obtaining a feasible path. A feasible path for the USV would be a path that is not very demanding, with respect to sharp and sudden turns.

4.2 Idea

The idea is to create a simulator which generates a map of a chosen area. The map should consist of equally sized cells, with the opportunity of changing the dimension of the map, as well as the size of each square. For instance, with each square equal to 1 m^3, with a map size of 100x100, the dimension of the map would be 100 m x 100 m with a total of 10000 cells. Each cell can be marked as free or occupied. The occupied cells would be i.e. be the pier or any other obstacles.

Once the map is created with obstacles added, the USV's position and desired destination can be added such that the A* algorithm can create a path. From Figure 4.1, the required inputs will have to be the starting position, desired destination and location of the pier wished to dock at. The output of the package should be the planned path.

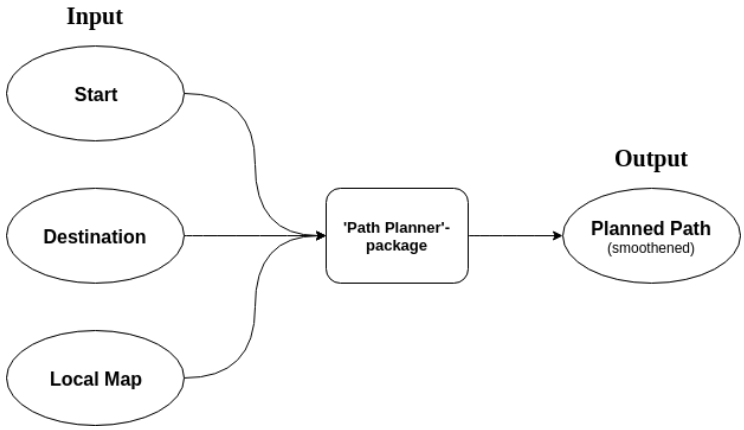


Figure 4.1: I/O of `Path Planner`

4.3 Development

Prior to this package, the `Detection Filter` package had been made. Similarly to `Detection Filter`, the position of the USV is retrieved in real-time from the navigation system (simulated by play-back from a ROS-bag. In the same manner, the `Path Planner` will subscribe to the topic `/usv/odometry_ins_ned` for the USV's current position. Also, another input to the package is the output from `/SetDestination_node`, created previously. Now, two of the three inputs visualize in Figure 4.1 are retrieved. These inputs are messages sent to the main node in the `Path Planner` package, called `/PathPlanner_node`.

The following steps presented in the sections to come leads to the prepared solution of this task.

4.3.1 Creating a Map

In order to create a feasible path for the USV, a map had to be created. The map consists of a 2D array. Each element of the array corresponds to a cell. The map takes the form of Figure 2.5, shown in Section 2.3. The array is initialized to the wanted size (row × column), where every element in the array is assigned the value *0*. The map will mostly be displayed as a 100x100, where every cell on the map equals to $1m^2$. As mentioned in the Section 4.2 - Idea, this correspond to 10000 cells. This means the map cover $1km^2$. With the array initialized to only include *0* in for every cell, this currently indicated that all of the cells are marked as *free*. Table 4.1 shows what a value on the map correspond to. As of now, only the status *free* has been explained. The rest of the status used will be explained at the proper time, in the sections to follow.

The map is connected to real-world positions by creating a new 2D array (referred to as *gridMap*). This array has the same size as the map, where every element of the

Table 4.1: Map Cell Indication

Value	Status
0	Free
1	Obstacle-Inside
2	Obstacle-Outline
3	Safety Extension
4	Path
5	Waypoint
6	Start & Goal

array has a lat/long point. By comparing the two arrays we retrieve the position of the cell in the map by a lat/lon point. The positions are calculated with (2.22), after the top left corner of the map is defined. From the top left point, a new lat/lon point is calculated every $1m$ with a bearing angle of 90°. The first row consists of 100 lat/lon points, as 100 is our amount of columns in the 2D array. With the first row, another iteration has created, to generate lat/lon points starting at each column at iterating through with a bearing angle of 180°. We have now created a new 2D array with lat/lon points correlated with the map. Later on, the *gridMap* will play an important role in adding obstacles to the map. Figure 4.2 visualizes how a map would look like. The figure is scaled to a 5x5 2D-array to simplify the thought process. Dependent of what the top-left position of the map is set to, considering each cell to be 1 m^2, the real-world positions of the cells of the first row are calculated using the top-left position, creating a new position every 1 m with a bearing angle of 90 °. From the equations derived in Section 2.9 - Geographical Coordinate Calculation, (2.22) shows the math behind calculating a new position. Each of the black dots corresponds to a lat/lon point generated. To sum up, there were created two similarly sized arrays, one containing integers with information about the cell (free, obstacle, etc.), and the other containing floating point values informing about the cells global position in the real world.

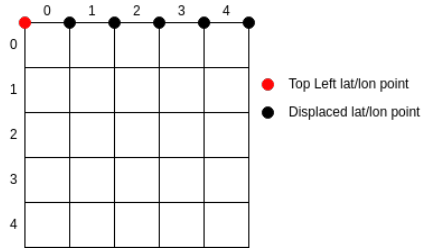


Figure 4.2: Initial Map

Consider knowing the position of the USV, the corresponding cell number of the USV's position can be extracted by finding out which index the 2D array of all the lat/lon-positions the current position lies within. By the Pythagorean theorem, the length of x and y can be found, after having calculated the distance between the top left and the USV position, and the bearing angle of the line drawn between the points. Further, the length of the two sides is divided by the length of a cell to extract the row and column number of the cell (integer division). From the illustration in Figure 4.3, the USV position is the located in cell **(3,3)**.

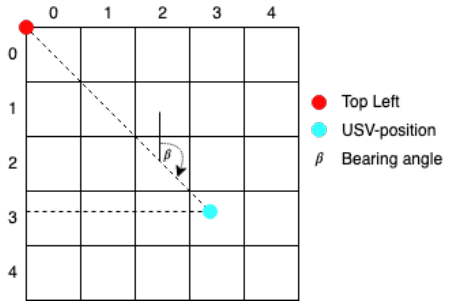


Figure 4.3: USV Position to Cell Number

Now we are able to identify the USV's position on the map and to the index which cell it is located in. The same procedure is performed to receive the cell number of the set destination. To obtain the set destination, we connect the `Path Planner`

package with the `Detection Filter` package. We are interested in subscribing to the topic `/destination_position`. Before the A* algorithm is constructed to plan a path between the start node and the destination node, we need to fill the map with obstacles. Otherwise, the A* algorithm would basically only draw a straight line between the two points.

4.3.2 Adding Obstacles

After being able to locate both the current position of the USV and the desired destination, the next task is to add the pier of Horten onto the map. From `https://www.google.com/maps/` it is possible to extract the corner points of the pier, using the satellite images. It is necessary to enter the position of the pier manually. The operator has the option to use the default pier that is already programmed for (Horten), or add a new pier. A node called `/SetPier_node` has been made to make it possible to use another pier without the need of entering the code. Figure 4.4 shows how the pop-up window looks like, for any operator of the system. The operator is given a choice of keeping the default pier (*'n'*), or using a new pier (*'y'*).

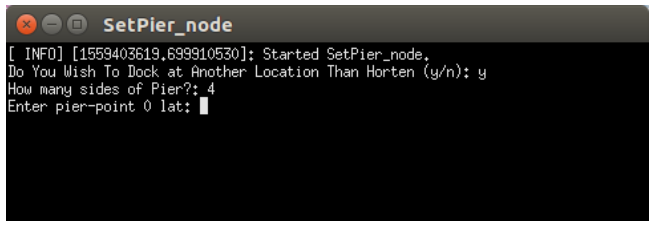


Figure 4.4: `SetPier_node` terminal window

To be able to add the pier as an obstacle, a function similar to the one used to generate points for the initial map was used. Firstly, by calculating the bearing angle of the line between a point to the next of the pier corners. Then, a new lat/lon-position was calculated every 1 m between two corners. Storing all the positions in a vector, the

vector can be converted to the corresponding cell number in the map. With the cell number, we overwrite the position on the map where the position lies. Now the outline of the pier has been added.

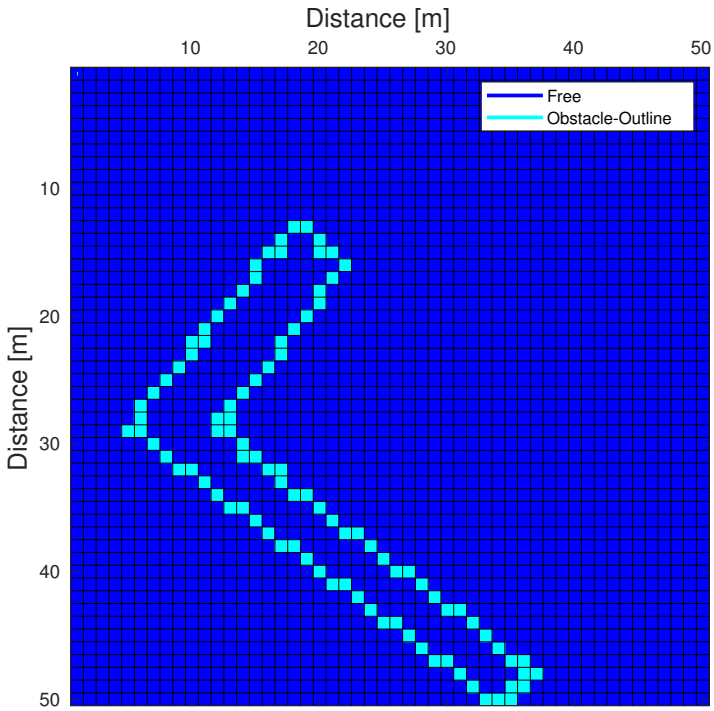


Figure 4.5: Outline of Pier Added to Map

We wish to fill the inside of the pier such that they are marked as occupied, where there is no possibility of the A* algorithm to plan through the pier. Explained in Section 2.3 - A* Search Algorithm, the A* algorithm may take a path from a cell to the next that is obliquely place according to the prior. The filler-function was created in a way such that any closed shape can be totally filled.

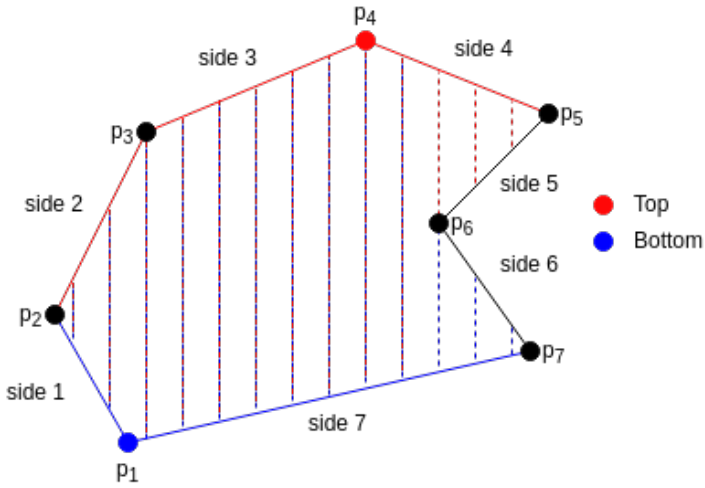


Figure 4.6: Filler Function Example

From an array of corners indicated by their cell number, the top and bottom point is extracted by finding the point with the highest and lowest row number, respectively. The top point then creates a new vector of the points which may be referred to as the top layer. The top layer includes the points involved in the red lines from Figure 4.6. From the index of the top position, it is being searched in both directions for other points that lie left or right from the top point. For example, if we search for a point with a smaller longitude than the top point, we search in that direction until the longitude of the next point is larger than the previous. If a point has equal longitude, we continue the search since the next point may have smaller longitude. Likewise for the search in positive longitude. The array for both the top and the bottom layer consists of the index of the side included in the layer. From the figure, the top layer is **[2,3,4]**, while the lower layer is **[1,7]**.

The filler-function needs two arrays as input, the top layer array and the lower layer array. The arrays need to be sorted before they are sent to the function. Since the points are added in order (from p_1 to p_7 following the outline of the shape), we wish to

sort the arrays such that sides in the arrays are aligned correctly. For example, for the top layer, the following sides may be **[3,4,2]** before sorting. After sorting, the array would be **[2,3,4]**. If an array would be **[1,7,6,2]**, the sorted array would be **[6,7,1,2]**, such that the points of the shape are in order.

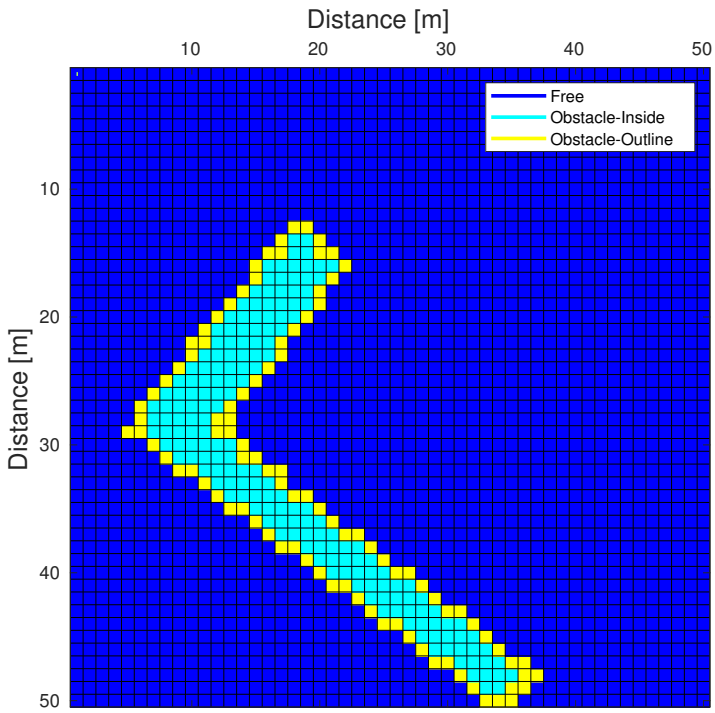


Figure 4.7: Pier Filled

Figure 4.8 presents a sketch of how the filling function works. The created function works for all different shapes, i.e. a pier or another obstacle at sea. It is now possible to add the points of an arbitrary obstacle with the knowledge of the obstacles corner

positions in lat/lon to produce a filled obstacle that can be added to the map. The cells on the inside of an obstacle are filled using the lower and upper arrays of sides. For instance, take the array consisting of the lower sides in a shape. Every cell that lies between the lower layer and the outline of the shape in the upwards direction will be marked as an occupied cell (with status *Inside of Obstacle*). The same procedure follows for the upper layer, however, cells that lie between the upper layer and the outline of the shape in the downwards direction.

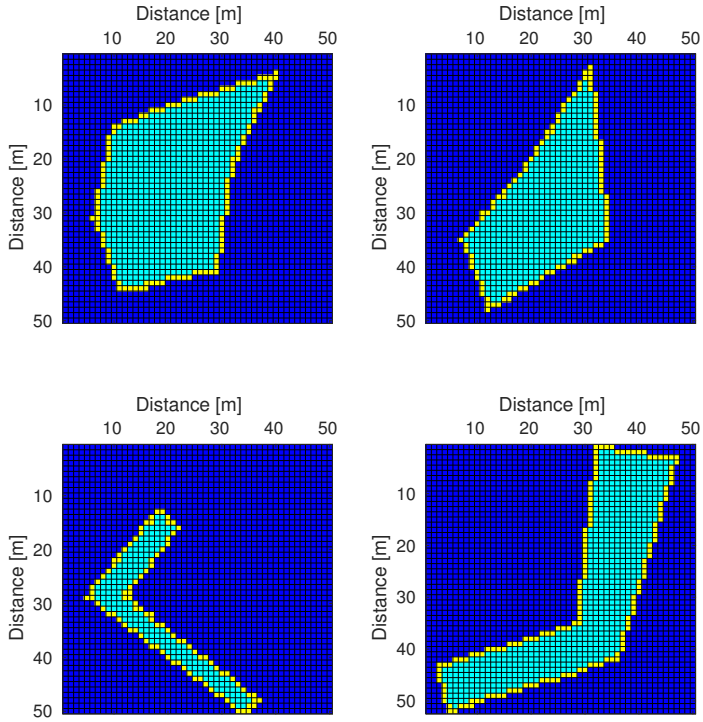


Figure 4.8: Filling Arbitrary Shapes

4.3.3 Running the A* algorithm

As the obstacles have been added to the map, we are now going to plan a path for the USV. Inspiration of how to implement the A* algorithm in C++ were looked up online [13]. The algorithm was explain in Section 2.3 - A* Search Algorithm. From a starting position to the destination position, the algorithm searches for the lowest costed path towards the goal, going around obstacles it may meet. From the start node, the search begins. The next node to be searched is the node with the lowest total cost, $f(n)$, from the neighbours of the start node. The final path consists of the predecessors from the goal node to the start node. Figure 4.9 shows how the path that the A* algorithm proposes from start node **(6,6)** to goal node **(31,15)**. The path nicely avoids the obstacle.

The A* algorithm has been set to only accept any cell without the value *1* (Obstacle-Inside) or *3* (Safety Extension). As seen from the figure, the path is allowed to use the *Obstacle-Outline* cells. The reasoning behind this is for allowing ourselves to set a destination at a *Obstacle-Outline* cell. The goal cell marked on the figure is really such a cell.

A drawback to this method compared to this project is that the planned path would traverse along the pier if it is in the way. The planned path for these instances does not follow the rules explained in the introduction of this package. A non-feasible path would be produced, as there is not expedient to follow. We want to create a safe path where the visited cells keep a distance from the pier until a safe path towards the goal has been deployed. Figure 4.9 shows how a path would look like using the A* algorithm on the created map, with the pier in Horten as the obstacle.

To deal with the problem of the algorithm traversing along the pier, three waypoints were added. It was then constructed various statement of which waypoint the algorithm should traverse to, in order to reach the destination in a feasible manner. Currently,

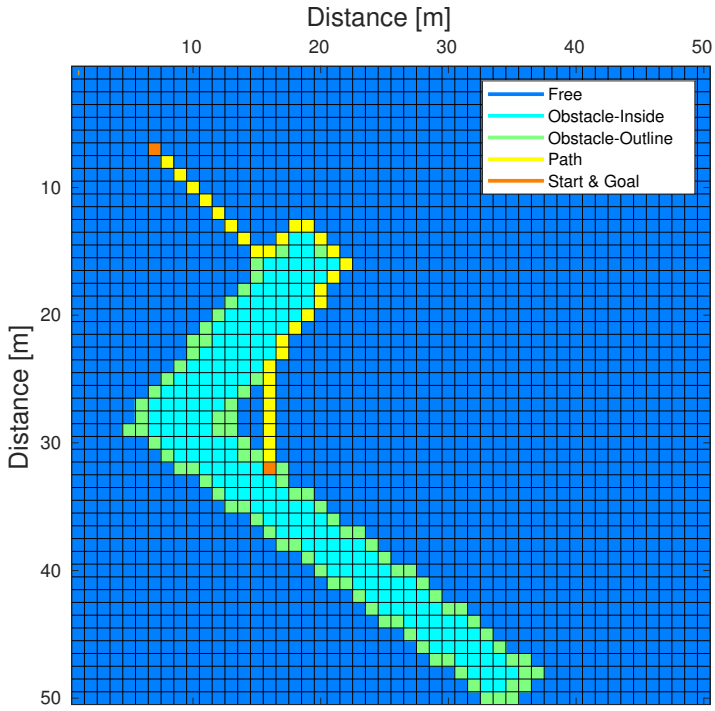


Figure 4.9: Original Path With A*

the working solution consists of these waypoints. This is a custom design, constructed directly for docking at the pier in Horten. Therefore, that part of the solution is not scalable in the sense of it working at an arbitrary pier at a different location than Horten. This is an issue further discussed in Chapter 6 - Conclusions.

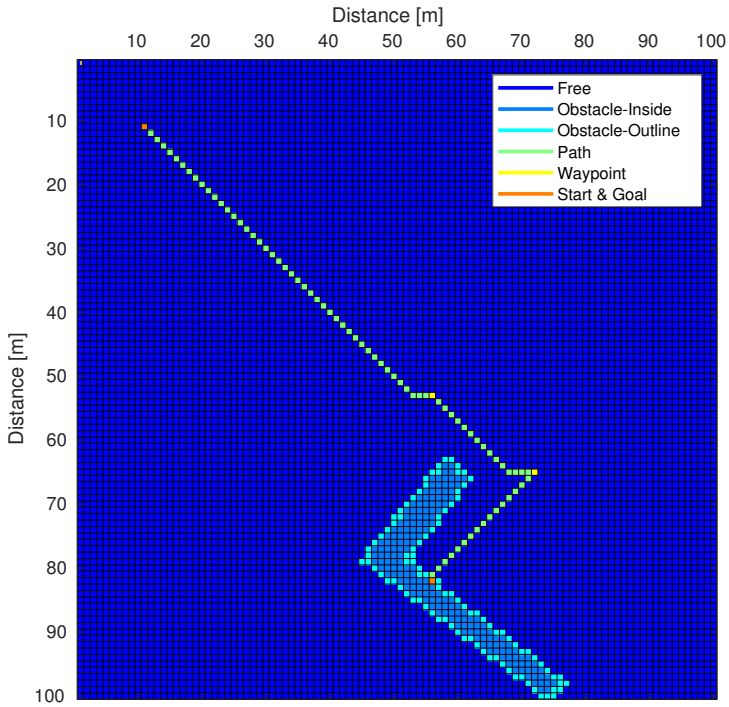


Figure 4.10: Map With Path to Pier (Lower-Side)

Nevertheless, the waypoints are created such that it is possible to dock at sides of the pier visualized by Figure 4.10 and 4.11. Depending on where the starting position and destination is set, the waypoints will be placed in the manner such that the path will keep a distance away from the pier. It does no longer traverse along the outline of the pier. Looking at Figure 4.10, the waypoint closest to the destination is created using the distance and angles of the pier sides. If a line were to be drawn between the destination and the closest waypoint, the line would be perpendicular to the side

of the pier where the destination lies. To generate the next waypoint, it displaced with a distance with a bearing angle similar to the side of the destination. Looking at Figure 4.11, a similar thought process is used. The waypoints a generated such that the A* algorithm finds the shortest path from the start point to the destination, with waypoints visited along the way.

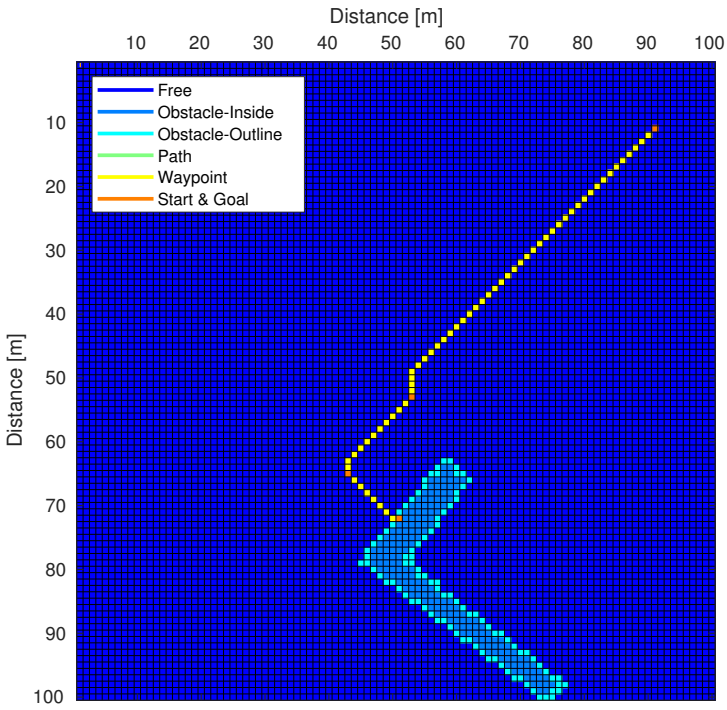


Figure 4.11: Map With Path to Pier (Upper-Side)

The path is not feasible yet, because the path currently consists of rough edges and sudden turns. The path has to be smoothed further, as explained in the next subsection.

4.3.4 Smoothing Planned Path

The path is smoothed similar procedure as averaging a signal. The circular array is used to store a number of previous points. Figure 4.12 and 4.13 illustrates the initial path with the smoothed path corresponding to Figure 4.10 and 4.11, respectively. The edges of the initial path now create a continuous turn, instead of a sudden one. The last segment which is the last way-point to the destination is almost identical to the planned path. The reason for this is because we wish to arrive at the set destination, and not an averaged one.

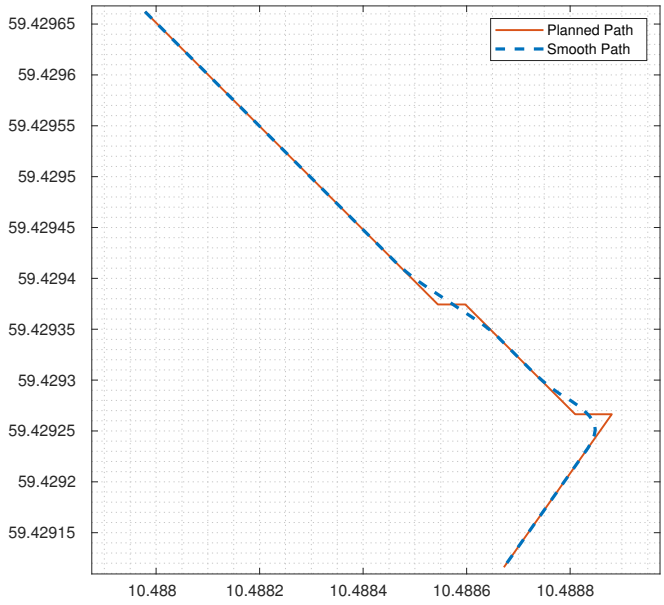


Figure 4.12: Planned- and Smoothed Path (Lower-Side)

There has been created a new node which processes the initial path to create the smoothed path. The node is called `/SmoothPath_node`, which only subscribes to the topic `/planned_path` that `/PathPlanner_node` produces. The circular array consists of 10 value. Until the array has been filled, the average produced is based on the sum of the values and divided by the number of values currently in the array.

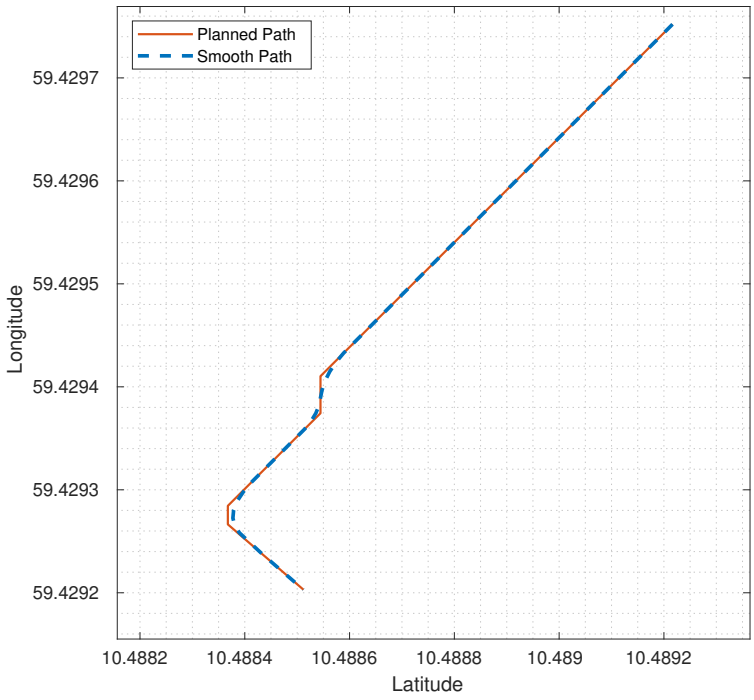


Figure 4.13: Planned- and Smoothed Path (Upper-Side)

4.4 Implementation

This chapter constitutes to the `PathPlanner` package created in ROS. Figure 4.14 show how the nodes created are connected. The main node where the map is created and the A* algorithm is executed, is in `PathPlanner_node`. The node subscribes to three topics; `/usv/odometry_ins_ned`, `/pier_position` and `/destination_position`. With the three inputs, the `/PathPlanner_node` is able to produce a planned path, where the A* algorithm is applied to the map created. The output of the node is the topic `/planned_path`, which `/SmoothPath_node` subscribes to. In this node the path is smoothed such that a feasible path for the USV to use is created.

This package is built with `DetectionFilter` as a dependency. It uses the `/SetDestination_node` that were created in the package.

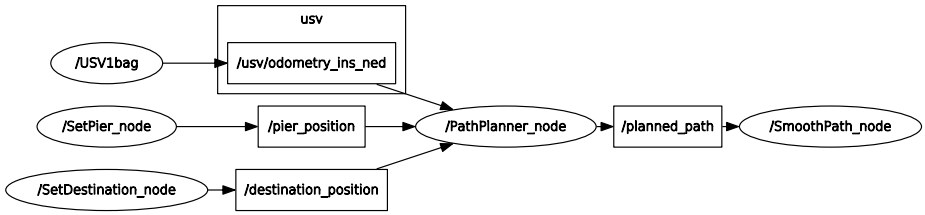


Figure 4.14: Communication Between ROS-Nodes

4.5 Simulation and Results

To run a simulation of the package, a launch file is created to simultaneously run all nodes at once. The four nodes from Figure 4.14 is included in this file, as well as playing the ROS bag. Once the launch file is executed, multiple windows pop up, allowing us to set certain parameters. Figure 4.15 shows the windows for the user to interact with before a path is generated. A destination must be set, as well as defining the pier

wished to dock for. It is possible to input another pier by the corner positions of the pier. However, the path generated works optimally only for the pier in Horten. For any other piers, the situation would look like the path in Figure 4.9.

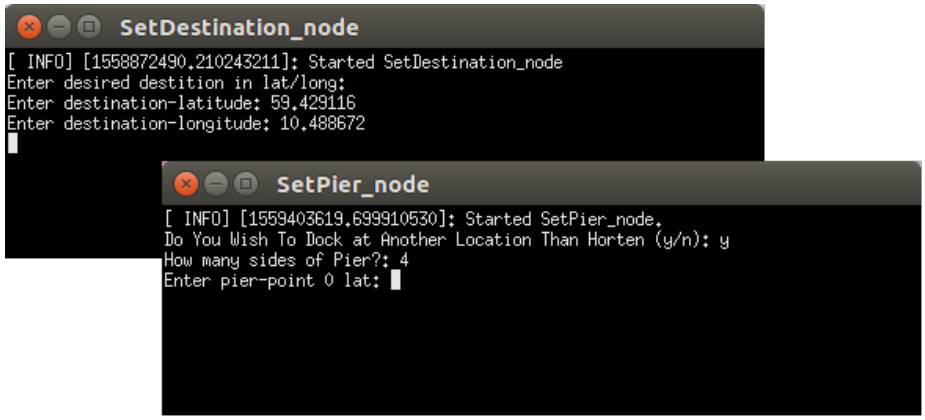


Figure 4.15: Launching `PathPlanner`

To further test the package, a simulation was performed was an obstacle was placed on the map. The obstacle can be looked like a little island, where obviously our path cannot go through. The obstacle was placed in a similar fashion as a pier would. The corner points of the island were set, and the filler function marks the inside cells ass occupied (*1 - Obstacle-Inside.* As we do not need to dock at the island, a safety extension of the obstacle was added for the simple case of avoiding the problem of A* traversing along the outline of the obstacle. The obstacle was padded by three layers around itself, and marked with *3* (safety extension). As previously told, cells with the value of *3* are looked upon as occupied cells in our A* algorithm implementation.

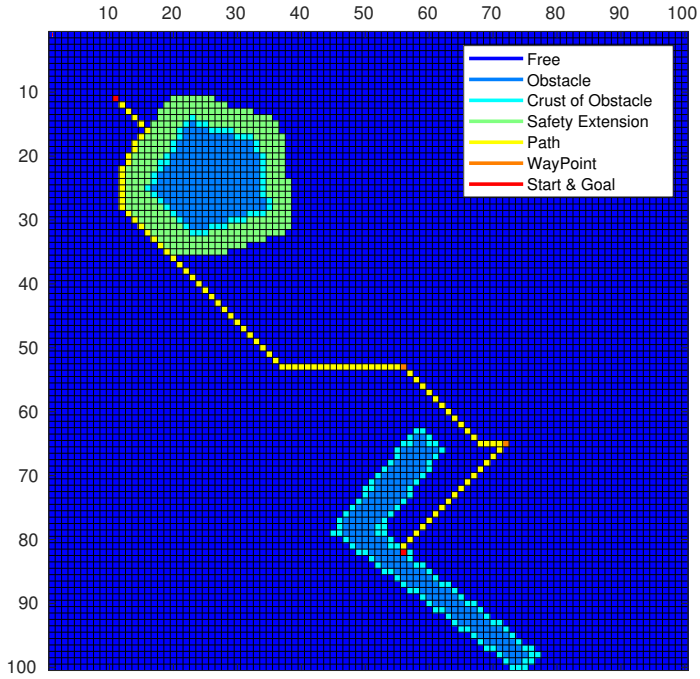


Figure 4.16: Planned Path with Additional Obstacle

Figure 4.16 shows the path from start node **(10,10)** to the goal node **(81,55)**. It is successfully created a path that avoids the obstacle added. The smoothed path is shown in Figure 4.17. It displays a nice path which the USV should be able to follow without too much effort. There lies no problem in smoothing the path around the added obstacle, as the padding of the obstacle is made large enough without the smoothed path entering the obstacle.

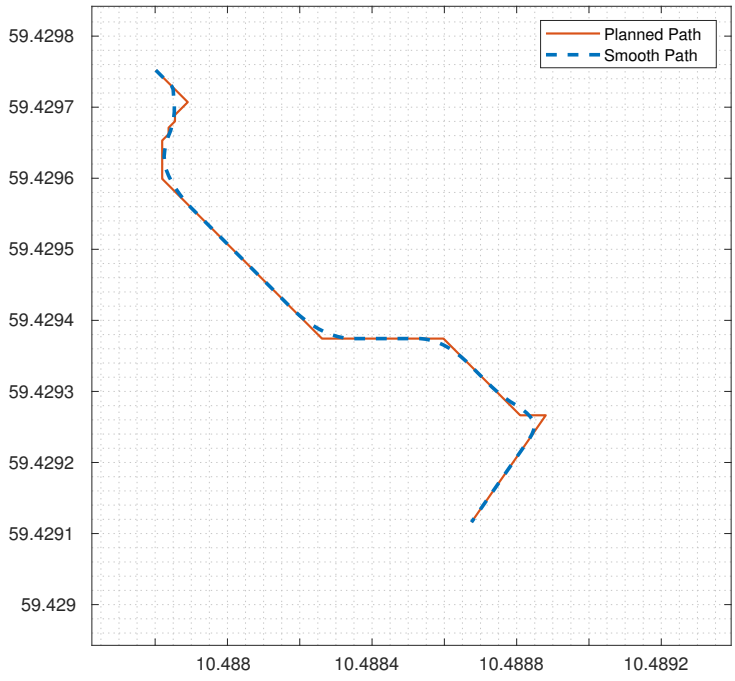


Figure 4.17: Planned- Smoothed Path with Additional Obstacle

Chapter 5

Discussion

This chapter will discuss the results from the `Detection Filter`- and `Path Planner` packages. The results were presented individually in Section 3.5 and 4.5, respectively. The discussion will summarize the results, as well as pinpoint the difficulties with the solution.

From the problem description, it was mentioned that the task could be divided into two separate problem. Both of the problems have been created as a ROS package, such that it is possible for FFI to integrate the packages in their system. The `Detection Filter` package produces a signal which indicated whether the USV has made contact with the pier. There has been success in using the package to detect docking using the prerecorded bags received. The `Path Planner` package produces a path from the USV's current position to the inputted desired destination. The initial path created using the A* algorithm is smoothed to obtain a feasible path for the USV. In that manner, the USV can navigate to the destination avoiding sharp turns. With obstacles added to the map, the A* algorithm manages to produce a path that around the obstacle.

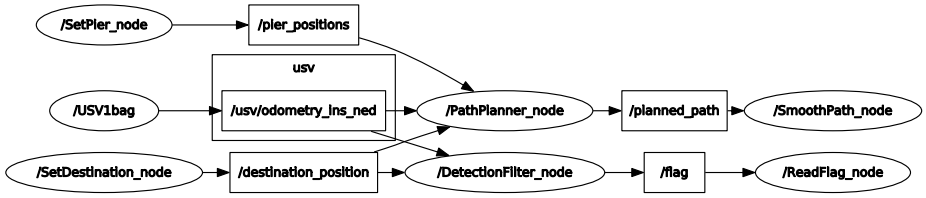


Figure 5.1: Communication Between ROS-Nodes of Both Packages

Figure 5.1 shows the interconnection of the packages when used at the same time. The `Detection Filter` is a stand-alone package which produces the signal of detected docking, while the `Path Planner` subscribes to the destination set in `Detection Filter`.

As the packages only have been tested on two specific cases, there is still some uncertainty with regards to the solution created. For instance, the prerecorded bags were captured in good weather conditions, which means no or little waves at the time of docking. There is implemented roll compensation, such that waves should be cancelled. Nevertheless, the moment the USV makes contact with pier, it may have a higher or more varying velocity than in the prerecorded bags. It is not known how the signal might act. And with that it is not know if the flag will be triggered at any instance towards the pier before making contact.

Possibly, only slight modification of the solution would have to be modified, if at all required. A problem might lie in a varying entry to the pier, such that a spike suddenly is detected before the USV has made contact with the pier, and is withing the distance threshold set.

When it comes to the `Path Planner` package, the simulator created to test the A* algorithm works as intended. The A* star algorithm in itself does not need to be changed. As it is not a feasible path for the USV to follow the outline of the pier, or any other obstacle. With regards to the pier, a couple of waypoint were added at various

locations outside of the pier, such that the A* algorithm traverses to the most optimal waypoints before traversing further to the destination. For any obstacle at a distance from the pier, there is created a safety extension around the obstacle. In this manner, the A* algorithm does not traverse along obstacles, the safety extension also is marked as occupied.

Also, the `Path Planner` package does currently not support docking for an arbitrary quay. The package is designed for docking at the pier in Horten. The waypoints are directly designed for docking at two sides of the pier. Therefore, the solution is not universal, with it working for an arbitrary pier. This subject will be further explained in the next chapter, Conclusions as further work.

Chapter 6

Conclusions

It can be concluded that the created packages does follow the purposes of the designated task presented, with the prerequisites that were. From the two data sets retrieved, the solution is able to detect when the USV has made contact with the pier, as well as creating a feasible path for the USV to follow.

There is still that can be done to improve the solution. In the following section, the elements of the task that can be improved will be explained.

6.1 Future work

Using a similar method as adding the safety extension around an obstacle that is not the pier, could be used around the pier to avoid the need of creating waypoints. Figure 6.1 shows how the pier looks like with a safety extension. The safety extension is

set to five iterations around the pier. With the safety extension, the A* algorithm can run more freely. It is only at the last segment from the end of the safety extension to the pier a solution has to be created. For instance, allow the USV to breach the safety extension once a straight line towards the target has been achieved. The point at the safety extension should then be perpendicular to the side that the destination lies on. Then the USV can slide into the desired docking position with the correct orientation, assumed that the orientation is attained for example at the safety extension point.

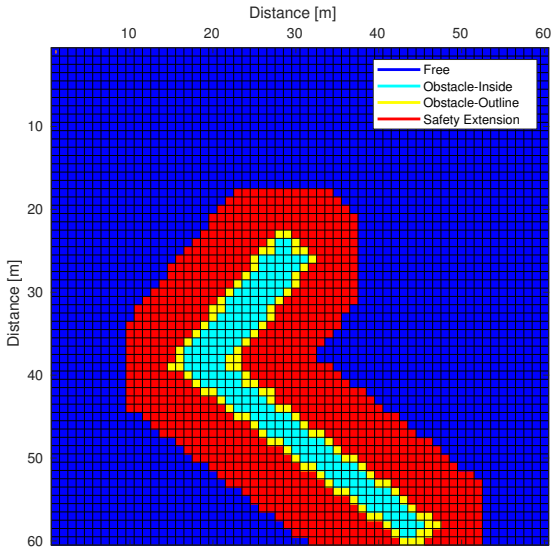


Figure 6.1: Pier With Safety Extension

Another factor, in being more confident of the solution, more tests could be executed for various situations and conditions. The situations being from different starting positions and desired destination, and the conditions being the speed of the USV and weather.

As this project only produces the path to the pier, there is also still work to be done with regards to the USV following it. The aspects of how to use the thrusters according to the paths behaviour is something that needs to be addressed. It was contemplated how to generate a path in order to make it easier for the USV to follow it. The last segment of the path exists of a straight line perpendicular to the side where the desired destination is set. This makes it possible for the USV to maneuver sideways in yaw direction towards the destination, if the orientation is obtained at the last waypoint (or at the safety extension).

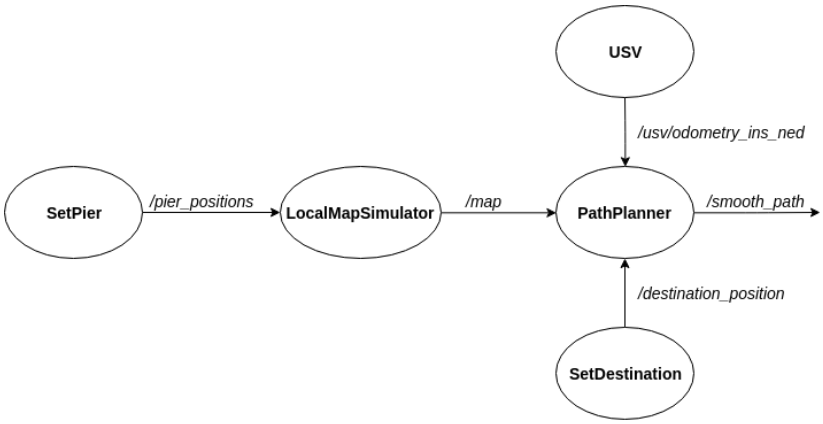


Figure 6.2: Future Setup of `Path Planner` package

A little bit of work should also be done in rearranging the package `Path Planner`. Figure 6.2 shows a more optimal setup, than the one currently used. Right now, both the map and the path is being created in the `PathPlanner`. It would have been nice to create a new node, which could be called `LocalMapSimulator`, where the simulated map is created. By doing this, it will be easier to use the `PathPlanner` with the actual map created by the USV, where it is not simulated. In that case, we can remove the `LocalMapSimulator` such that only the `PathPlanner` where the A* algorithm is implemented, is used. Also, the `SmoothPath_node` could be removed, such that the smoothening of the path happens in the same node as the path is generated. Then *PathPlanner* will produce the topic */smooth_path*.

Appendix A

Software

A.1 Installing Ubuntu OS

The distribution used in this Masters Thesis has been UBUNTU 16.04 LTS. It is the same distribution that FFI is running on. The installation can easily be found under the `Download`-tab at `https://www.ubuntu.com`.

A.2 Installing ROS

ROS comes in various versions and releases. Install the distribution `Kinetic Kame` entering the following command in terminal from Ubuntu.

```
$ sudo apt-get install ros-kinetic-desktop
```

The rest of the installations procedure can be found at `http://wiki.ros.org/kinetic/Installation/Ubuntu`

Creating a ROS environment

Once the ROS-distributions is installed, we must create a workspace to use it. The following commands creates the workspace needed (called `catkin_ws`):

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Without the need of sourcing the workspace every single time a new shell is launched, we run the following command:

```
$ sudo gedit ~/.bashrc
```

Now a text-editor pops up, and we add the following line at the bottom of the file that has presented itself.

```
source ~/catkin_ws/devel/setup.bash
```

The workspace has now been built and sourced.

A.3 ROS-launch

The created ROS packages can be launched by inserting the following commands in the terminal window. A launch-file runs the nodes listed, simultaneously. Therefore it is not needed to start the nodes one by one. The packages `DetectionFilter`

and PathPlanner can be launched. PathPlanner includes DetectionFilter, while DetectionFilter can be run on its own.

```
$ roslaunch detection_filter DetectionFilter.launch
$ roslaunch path_planner PathPlanner.launch
```

The first command runs on USV1.bag. To run the package with USV2.bag simply replace DetectionFilter.launch with DetectionFilter2.launch.

The second command also runs on USV1.bag. The same procedure is performed as the previous, to run wiht USV2.bag.

A.4 PlotJuggler

PlotJuggler is a package created in ROS that can be downloaded at https://github.com/facontidavide/PlotJuggler. The program PlotJuggler can in real-time subscribe to ROS topics. The package was used diligently in the work leading up to solution.

A.5 MATLAB

MATLAB is used to create all plots shown in the report. The tool can be used to display data nicely. To plot the maps, the *imagesc* command was used.

A.6 Atom

Atom is a source code editor used to produce the C++-code. It is a simple to use editor which apply syntax highlighting for C++. Atom is available in the Linux operating system, and has been used in this project. In Atom there exists an integration with GitHub, which also has been used to store the code, such that other computers can access the worked code. Atom can be downloaded at https://atom.io/.

Appendix B

Detection Filter-package

This appendix will contain an short explanation of the structure and functions used and created in C++, that lead to the end the product of detection a spike in the horizontal sway acceleration of the USV.

B.1 Structure

The tree-structure of this package is shown below. It includes a *'config'* folder containing a .xml-file to load a pre-set window look in `PlotJuggler`. This config file is included in the launch-file, such that when `PlotJuggler` is launched, we launch it together with the config file. The *'include'* folder contains the header files with function declarations. Folder *'launch'* includes two launch-files, one for each of the ROSbags. Folder *'msg'* includes message-files with various data-types, used to publish desired variables. Folder *'src'* contains the C++ code files. Lastly, *'CMakeLists.txt'* and

'package.xml' is needed to build the package, and to provide meta information about the package, respectively.

```
/detection_filter
├── config
│   └── DetectionFilter.xml
├── include
│   └── detection_filter
│       ├── Buffer.h
│       ├── Constants.h
│       ├── Filter.h
│       ├── Math.h
│       └── Structs.h
├── launch
│   ├── DetectionFilter.launch
│   └── DetectionFilter2.launch
├── msg
│   ├── filter.msg
│   └── positionUSV.msg
├── src
│   ├── Buffer.cpp
│   ├── DetectionFilter_node.cpp
│   ├── Filter.cpp
│   ├── Math.cpp
│   ├── ReadFlag_node.cpp
│   └── SetDestination_node.cpp
├── CMakeLists.txt
└── package.xml
```

Appendix C

Path Planner-package

This appendix will contain an short explanation of the structure and functions used and created in C++, that lead to the end the product of planning a path from a starting point to the destination along the pier.

C.1 Structure

The tree-structure of this package is shown below. It includes a *'config'* folder containing a .xml-file to load a pre-set window look in `PlotJuggler`. This config file is included in the launch-file, such that when `PlotJuggler` is launched, we launch it together with the config file. The *'include'* folder contains the header files with function declarations. Folder *'launch'* includes two launch-files, one for each of the ROSbags. Folder *'msg'* includes message-files with various data-types, used to publish desired variables. Folder *'src'* contains the C++ code files. Lastly, *'CMakeLists.txt'* and

'package.xml' is needed to build the package, and to provide meta information about the package, respectively.

```
/path_planner
├── config
│   ├── DetectionFilter_and_PathPlanner.xml
│   └── PathPlanner.xml
├── include
│   └── path_planner
│       ├── Algorithm.h
│       ├── Math.h
│       ├── Obstacle.h
│       ├── PathPlanner.h
│       ├── SmoothPath.h
│       ├── Structs.h
│       └── Utilities.h
├── launch
│   ├── DetectionFilter_and_PathPlanner.launch
│   ├── DetectionFilter_and_PathPlanner2.launch
│   ├── PathPlanner.launch
│   └── PathPlanner2.launch
├── msg
│   ├── path.msg
│   └── pier.msg
└── src
    ├── Algorithm.cpp
    ├── Grid.cpp
    ├── Math.cpp
    ├── Obstacle.cpp
    ├── PathPlanner.cpp
    └── PathPlanner_node.cpp
```

```
/path_planner
├── src
│   ├── SetPier_node.cpp
│   ├── SmoothPath.cpp
│   ├── SmoothPath_node.cpp
│   └── Utilities.cpp
├── CMakeLists.txt
└── package.xml
```

Acronyms

FFI Norwegian Defence Research Establishment. i–iii, 1–5, 7, 27, 28, 30, 31, 45, 65, 73

GPS Global Positioning System. 4, 40

IMU Inertial Measurement Unit. 3

NTNU Norwegian University of Science and Technology. i, 4–6

OS Operating System. 8

ROS Robot Operating System. 7, 8, 27–30, 41, 42, 47, 61, 65, 73–75, 77, 79

SNAME Society of Naval Architects and Marine Engineers. 15

USV Unmanned Surface Vehicle. ii, iii, ix, 1–5, 7, 27–33, 35, 37–42, 45–47, 49, 50, 55, 61, 63, 65, 66, 69–71, 77, *Glossary:* USV

Bibliography

[1] HamiltonJet. Waterjet Overview, 2015.

[2] Thor I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control.* John Wiley & Sons, 1st edition, 2011.

[3] Kongsberg Gruppen. Yara and Kongsberg Enter Into Partnership to Build World's First Autonomous and Zero Emissions Ship. https://www.kongsberg.com/maritime/about-us/news-and-media/news-archive/2017/yara-and-kongsberg-enter-into-partnership-to-build-worlds-first-autonomous-and/, 2017.

[4] Volker Bertram. Unmanned Surface Vehicles – A Survey. 2008.

[5] Joachim Spange. *Autonomous Docking for Marine Vessels Using a Lidar and Proximity Sensors, MSc thesis.* 2016.

[6] Thomas Stenersen. *Guidance System for Autonomous Surface Vehicles, MSc thesis.* 2015.

[7] Morten Fyhn Amundsen. NTNU ITK thesis template, 2018.

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basic for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, 1968.

[9] SNAME. Nomenclature for Treating the Motion of a Submerged Body. *Technical and Research Bulletin*, (1-5), 1950.

[10] J. C. K. Chou. Quaternion Kinematic and Dynamic Differential Equations. *IEEE Transactions on Robotics and Automation*, 8:53–64, 1992.

[11] Adel S. Sedra and Kenneth C. Smith. *Microelectronic Circuits*. Oxford University Press, 5th edition, 2004.

[12] R. W. Sinnott. Virtues of the Haversine. *Sky and Telescope*, 68:158, 1984.

[13] Andris Jansons. A Star (A*) Path Finding C++, 2018.