ACIT5930

MASTER'S THESIS phase III
in
Applied Computer and Information Technology
(ACIT)

Spring 2021

Applied Artificial Intelligence

Making an AI powered ASV

Magnus Kjelsaas

Department of Computer Science
Faculty of Technology, Art, and Design

Preface

I have always been in love with the ocean. It is a thing of unmatched beauty, and home to an infinite amount of resources and life. To be able to traverse the seas using only the wind is an engineering feat like no other, which have been done for thousands of years. Discovering new lands, new resources, and connecting peoples across the globe. Enabling trade, enabling life, and enabling a better life for all.

The ocean is one of the most important resources we have. Both in terms of health, economy, and transportation, the ocean is invaluable and we could never have lived without it. Unfortunately, not everyone is able to enjoy it as it deserves. Having a boat is considered a privilege, and requires a certain amount of time and money to maintain. I would love to see a world where the ocean is more accessible to everyone, as it is literally big enough to fit all of us at once.

The project in this report is inspired by the ocean and its opportunities. It was formed from a motivation to share the ocean and explore its potential. It is based on a hope that I may contribute to a grander project where humanity works together to share its resources, and where everyone can take part in the gifts that the ocean gives us.

I would like to thank my supervisor Vahid Hassani for everything he has done related to this project. He has been a great mentor, and a great source of both inspiration and criticism. This project would never have been possible without him, or his work related to robotics and the ocean.

I would like to thank the entire community related to robotics and autonomy related to the ocean. There are so many wonderful people out there with brilliant ideas and a drive to make the world a better place, and it is truly an inspiration to see. I hope you know that you are doing important work, and that you are valued. It is a huge honor to feel a part of this community.

I would like to thank Maritime Robotics for their support during this project. You have been helpful and supportive during difficult and stressful times, and this project would never have been the same without you. I love the fact that you are helping to enable ocean autonomy, and therefore helping to realise the full potential of the ocean.

Also, I would like to thank you. The reader. I would like to thank you for taking an interest in this topic, and the ocean in particular.

The entirety of the code used in this project can be found on GitHub, located at https://github.com/MrKjelsaas/school_projects/tree/master/master_thesis.

I would also like to mention that OsloMet decided to publish a press release related to this project. The press release can be found at https://kommunikasjon.ntb.no/pressemelding/vil-utvikle-selvkjorende-bater-pa-oslofjorden?publisherId=15678779&releaseId=17925396&fbclid=IwAR2wFPM-k2exOrBfJ21BeVbpCUg1ql0L2O-Fq9MeXgrqTtv1TlyujiqL2Ww. The press release has resulted in 8 news articles being published, including Finansavisen.no, Vårt Oslo, Bil24, and ITavisen.

Abstract

This thesis presents my master thesis project on creating an Autonomous Surface Vehicle that is designed to transport humans and goods. The vessel itself is not designed to carry neither humans nor goods, but is used as a platform for developing methods that can be used for transportation tasks like passenger ferries or cargo ships. The ASV developed in this project is mainly powered by Artificial Intelligence methods.

The project is divided into five key areas. These are path planning, path following, obstacle detection, collision avoidance, and auto docking. The path planning and collision avoidance methods are based on the A* algorithm, and the remaining use AI methods. Different Deep Reinforcement Learning methods are tested for auto docking, and an Evolutionary Algorithm called NeuroEvolution of Augmenting Topologies is used for auto docking and path following. A Computer Vision model called YOLOv5 is applied for obstacle detection.

To test and verify these methods, a simulation environment that can be used to both train and test AI models is designed and developed. The base simulation environment was tweaked to make a special environment tailored for path following, and one for auto docking. After training the AI models in the simulation environments, the models are deployed to an Otter USV (Unmanned Surface Vehicle), and tested in the waters at Filipstadkaia, Oslo.

The contributions of this project is twofold. The first is creating a simulation environment that can be used to train and test AI models for controlling an Otter USV. The second is showing that it is possible to control a maritime vessel using AI methods, more specifically Evolutionary Algorithms.

The Otter USV appears to be able to both follow a path and auto dock using the NEAT algorithm. This is based on the results from the simulations. When deploying the USV in the harbor, the behavior deviates from the simulations. This may be caused by malfunctioning sensors, weather conditions, a poorly trained model because of the simulation environment, or a combination of these.

Contents

Abbreviations

2D	Two Dimensional
3D	Three Dimensional
AI	Artificial Intelligence
ASV	Autonomous Surface Vehicle
AUV	Autonomous Underwater Vehicle
COLREGs	Collision Regulations (International Regulations for Preventing Collisions at Sea)
CV	Computer Vision
DDPG	Deep Deterministic Policy Gradient
DQL	Deep Q Learning
DRL	Deep Reinforcement Learning
EA	Evolutionary Algorithm
GPS	Global Positioning System
LOS	Line of Sight
MIMO	Multiple Input Multiple Output
ML	Machine Learning
MPC	Model Predictive Control
MVP	Minimal Viable Product
NEAT	NeuroEvolution of Augmenting Topologies
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
USV	Unmanned Surface Vehicle
VO	Velocity Obstacle

Chapter 1

Introduction

For as long as we can remember, Norway has been a seafaring nation. We have built boats that can both cross the open sea, and sail up shallow rivers. It is said that we are born with skis on our feet, but it may be more appropriate to say that we are born with sea water in our blood.

Our naval tradition continues to this day, and is as strong as ever. Today, Norway is a leading country in developing maritime technology. Major companies like Kongsberg Seatex, SINTEF Ocean, and Rolls-Royce Marine are coming to Norway to test and develop their technology (Nikel, 2017).

With the age of autonomy dawning upon us, there are amazing opportunities arising in the field of maritime technology. Underwater robots can stay underwater for extended periods of time to monitor and maintain underwater constructions. Autonomous vessels can explore huge areas on their own without the need for human intervention. And marine vessels can autonomously drive themselves from one place to another. Now, the time has come to make a reality of what was science fiction only a few years ago.

1.1 Motivation

Centralization has been an important aspect of the development of humanity. As time progressed and industrialization became a thing, many people moved to the city for work. This also increased productivity by having people live closer to their workplace, and gave a more effective logistic system by more easily distributing raw materials. But centralization doesn't come without disadvantages.

There is a limit to how tall it is realistic to build buildings, meaning that at a certain point one will have to expand horizontally. And apart from the physical limitations, many people dislike living in the city, and are happier living in the countryside, closer to nature. The solution for them is to move out of the city. The downside with this is that the commute between work and home takes longer.

To counteract this, humans have been constantly developing solutions to both travel and transport faster and more effectively. Notable examples are trains, cars, motor highways, and ferries. But while land-based transport seem to be rapidly advancing, water-based transportation hasn't gotten the same focus. There is a lot of hype around self-driving cars, but why is there less attention towards self-driving boats?

Self-driving boats have the potential to revolutionize the way we transport both humans and goods, and there are several reasons why this is the perfect time for this advancement. Micro-mobility is becoming increasingly popular. The sharing economy is a thing. Artificial Intelligence (AI) and robotics have come to a point where we can automate the tasks required to make a fully autonomous boat. We are standing on the verge of a whole field of technology that can and will revolutionize the way we transport both humans and goods, and there are several reasons for this.

Transport ships will have the opportunity to literally work around the clock. A ship will be able to dock automatically at its destination, and unload the goods. While stationed there, it can charge itself, load up new cargo, and find an optimal route to get to its new destination. And all this without the need for human intervention.

Piracy is unfortunately not a thing of the past. To this day, both passenger and cargo ships are being raided. These incidents often result in injury or death for innocent people. By using autonomous transport boats, one can drastically reduce the number of human casualties caused by these incidents.

The taxi market will never be the same. Instead of needing a human driver to be available, an autonomous boat can pick you up where you are, and drive you to your destination. All you need to do is to provide the destination. And with the ocean and fjords being what they are, there are plenty of lanes available, meaning you will never be stuck in traffic again.

The main motivation behind this project is to demonstrate that a self-driving boat can be a great contribution to the field of logistics and transportation. I believe that there is a huge untapped potential in the seaway, which can be used as a means of transportation for both humans and goods. This will in turn

benefit society as a whole both economically and socially.

1.2 Objectives

The ultimate goal of this project is to build a functioning prototype of an Autonomous Surface Vehicle (ASV). The ASV will take a destination as input, autonomously travel to the destination while avoiding obstacles, and dock there.

The project and it's goal will be broken down into the following key areas:

- Path Planning
- Path Following
- Obstacle Detection
- Collision Avoidance
- Auto Docking

All of these key areas will have their own sub-objectives.

The strategy for this project is to find the most fitting methods for each of these areas, and combine them to produce a functioning prototype of a vessel that can autonomously transport both people and objects to a desired destination. Each key area will have their own sub-objectives and restrictions.

Path Planning

Path planning is the process of planning a set of moves or actions that will take something (or someone) from one place to another. For humans, this can be planning a hiking trip, or checking which buses to take to get to the theater. Another familiar example is when using map-applications such as Google Maps, where one can enter a desired destination, and the app will show which road one needs to follow. The goal of path planning is to find a path that will take the person or object in question to its desired destination.

The objective of path planning in this project is to find a path for the ASV to follow. The path does not have to be optimal in terms of time and energy efficiency, but it does have to be realistic. This means that the path will take it safely to its destination if followed properly.

The specific goal that determines whether or not the path planning stage is successful will depend on the method chosen, but a general requirement is that the path must be able to lead the vessel to its destination.

Path Following

Path following is the act of realizing the planned path. This simply means that one follows the path planned earlier. If you are on a hiking trip for example, it

would mean that you keep walking on the trail without deviating too far from it. The goal of path following is to reach the desired destination.

In this project, the objective of the ASV will be to follow the predetermined path to its destination. The operation is considered successful if the vessel does not lose its way, and arrives at the desired destination. Deviating from the path does not reduce the success of the vessel. This is simply because it can be necessary for avoiding obstacles.

Obstacle Detection

Obstacle detection is the act of becoming aware of an obstacle that will have an impact on your path following. This implies two things. Firstly, only objects or people that are located along the planned path are considered obstacles. For example, a car driving next to you on the road is not an obstacle, because you are not going there. Secondly, only obstacles that are at a certain point along the path at the same time as the vessel are considered obstacles. This means that a car driving in front of you is not obstacle, but it will be if it suddenly stops, because it will lead to a collision.

The reason this is addressed is because the goal for obstacle detection in this project is only to detect ships, and not determine their position or movement. The reasoning behind this will be elaborated later in the report. The success criteria for this sub-objective is not binary, but is dependant on how well the vessel detects ships.

Collision Avoidance

Collision avoidance is about properly reacting to the situation that arises when detecting obstacles. This is mainly done in two stages. First, the vessel must find out how to avoid the obstacle, and then it must perform these actions. The first part is the hardest one, as it includes an element of uncertainty. For example, the detected obstacle can suddenly change direction, or it moves different from how it was perceived.

For this project, the objective of collision avoidance is to make sure that all obstacles are avoided. This means that the ASV and the obstacle does not make physical contact with each other. The sub-objective is considered a success if our vessel does not collide with any obstacle.

Auto Docking

Docking is the act of securely attaching a vessel to a wharf or a dock, which is a structure made for sheltering a vessel. In the case of an autonomous ship, it means going from simply being in the same area as its destination to being safely locked into a position which allows loading and unloading of people and goods.

Our objective for auto docking is to ensure that the ASV successfully docks at its destination. This does not include attaching, as this is another project

Figure 1.1: Frognerkilen is a bay in the Oslofjord, acting as a harbor for boats (Wikipedia, 2018)

entirely. The goal is to have the vessel positioned and oriented in such a way that it would be ready for attachment.

1.2.1 Specification

Below is a list of clarifications and specifications regarding what the ASV should be expected to do.

- Only input required is the destination
- Plans a path to the destination
- Follows the path to reach the destination
- Is able to detect ships
- Takes action to avoid collision
- Recognises its destination dock
- Maintains its position and orientation when reaching its destination

1.2.2 Limitations

Below is a list of limitations set to the project.

- Will not focus on the ability to transport humans or things
- The path that is planned does not have to be optimal
- Does not have to follow the path to the destination exactly
- Does not need to comply with the COLREGs
- Does not have to dock at an arbitrary place

Chapter 2

Timeline

This project is divided into three phases, which again are divided into sub-goals.

2.1 Phase 1 - Background

The first phase consists of formulating a problem statement, and exploring the state of the art by finding and evaluating different methods used in similar projects.

2.1.1 Formulating a problem statement

One of the first things that needs to be done in any engineering task is properly identifying and formulating a problem statement. This tells us exactly what needs to be done, and sets the scope for the project. This means that it is necessary to decide on the goals, and what limitations to set.

As a problem statement can always be improved upon, and can change during the course of the project, it is very difficult to say when one is "done" with making a problem statement. But one can follow certain principles when defining a problem statement. For example, the problem statement needs to be specific enough to be understandable and tangible, but also wide enough to allow for changing the plan underway when something occurs.

2.1.2 State of the art

State of the art is about researching and exploring current methods used by the community that deals with tasks related to the problem statement. In this case, it means that it is necessary to find methods used in ASVs that attempt to do things that are similar to what is desirable in the project in this thesis. In this case, it means finding methods used in path planning, path following, obstacle detection, obstacle avoidance, and auto docking.

As exploring the state of the art is a continuous process, it can not said to be done, or achieved. It can be deemed sufficient when the reader has an

insight into what technologies and methods are commonly used in the field. It is unrealistic to explore every option, but one should have a decent overview of some available methods for comparison and evaluation.

2.2 Phase 2 - Simulation

The second phase is method testing and evaluation. This involves deciding on relevant existing methods, developing new methods where needed, and testing all methods both individually and together.

2.2.1 Deciding on relevant existing methods

After gathering a good collection of different methods, some of them need to be selected for use in this project. This does of course require that there are existing methods that work. When evaluating the methods, a comparison between the pros and cons with what we want to achieve must be done. Then, I must decide on the ones that seem most appropriate. This stage is complete when an initial method for each of the five key areas is found.

2.2.2 Developing new methods where needed

In some cases, no good methods exists for achieving the specific goals in this thesis. There can be many reasons for this, including implementation issues, time, and compatibility. If this happens, I will need to either make a new method, or improve on the ones that already exist. This can almost be thought of as a "project within a project" with its own problem statement and goal. The stage can be considered done when a successfully developed method that works for our project in each of our five key areas is developed.

2.2.3 Testing and evaluating methods

After methods for everything I want to achieve have been gathered, they must be tested. This is to make sure that the methods we found work for our project. At this stage, the methods are tested individually. In some cases, it can be seen that the chosen methods do not work for this project. When this happens, one must go back and find another method to try. If all other methods that have been found fail, it is necessary to develop an in-house method. The goal of this stage is to find individually working methods for everything that I want to achieve in this project.

2.2.4 Putting everything together

After a method has been tested individually, it needs to be tested alongside the other methods. This is because one can discover that all methods work very well individually, but they can interfere with each other and cause problems that way. For example, path following and obstacle detection methods can want to

go different ways, so I may need to find a system that decides on which one to follow.

In the worst case, some methods may even be useless when combined with others, meaning that we need to go back to finding new methods. What this means is that phase 2 is not linear. There is a high chance that I need to move back and forth between the sub-goals until the desired result is achieved. This also means that it can be difficult to track the progress of the project at this stage.

The desired result at this stage is to see that all the found methods work together, and are ready for simulations.

2.2.5 Simulations

After working methods have been found, and seen that they work together, the simulation phase can begin. Here, not only whether or not the methods works by themselves is evaluated, but how they work when used in the environment that will be developed. Simulations can be thought of as a visualization of the methods to be used, so that it is possible to gain some intuition and understanding of how they make the USV behave. This is done to make it easier to find improvements and faults in the system.

Except for making the actual simulation program work, there aren't any formal requirements to the simulations. This is mainly because the simulations can be considered a tool. But what I want from the simulations is insight into how our system behaves, and get an estimate on how the USV will behave in the real world.

As mentioned above, phase 2 is not linear. After simulating, it may be necessary to go back to the methods and change them, to alter the behavior of the USV. This process is repeated until it is seen that the ASV is acting the way it should, and it is safe to deploy it in a real world environment.

Simulations are also an essential part of training Artificial Intelligence models, which we will see later in this report.

2.3 Phase 3 - Field experiments

The final phase is the experimentation phase. In this phase, all the methods are put into practice by conducting real world experiments. The experiments are then evaluated, and reflected upon. This also includes notes about suggestions for improvements and further development.

2.3.1 Experimentation

To conduct experiments, the USV loaded with the methods used in simulation will be tested in a real world environment. This means that I am going to actually put it in the water and see if it behaves the same way as it did in the simulations.

As with simulations, one is never done experimenting. There is a continuous evaluation going on, which means that the development will consist of going back and forth between experimentation, simulation, and altering the methods, to improve USV as much as possible. This means that all spare time can be used to keep improving our prototype. But considering the problem statement, the goal for this stage is to perform one successful experiment where the USV behaves in accordance with our problem statement.

2.3.2 Reflection

Reflection means evaluating the performance of the ASV, and deciding on future work. Here it is important to formulate what worked well, what did not work as well as hope, and what is missing. This lays the foundation for future work and what should be focused on next.

This stage can be considered done when the results of the experiments are presented, along with an outline with suggestions for future work.

Chapter 3

Theory

This chapter will cover the theoretical foundation for the project. The chapter is divided into sections that will give an explanation of each of the concepts, and relevant findings will be presented and evaluated. The theory chapter is intended to give the reader an overview over what concepts and solutions exist today in each of the fields required to make an autonomous surface vehicle, along with strengths and weaknesses.

3.1 The ASV and its components

In this section, the most important parts of the ASV is covered. A wide overview is first presented before going into detail on the most important components that make up an ASV.

3.1.1 What is an ASV?

An Autonomous Surface Vehicle (ASV) is a subcategory of Unmanned Surface Vehicles (USVs) that operates on the surface of water, fully autonomously. A USV is "any vehicle that operates on the surface of the water without a crew" (Yan et al., 2010).

The first use of USVs was at the end of the second world war (Council, 2005), where they were used to sweep for mines. Today, there is rising interest in the field of ASVs, as modern advances in automation and AI enable vehicles to do much more. Examples of areas where ASVs are well suited are minesweeping (Manley, 2008), data gathering (Dunbabin et al., 2009), commercial shipping (Yara, 2018), passenger ferries (BBC, 2018), and more.

An ASV is equipped with several components that enable it to operate on its own for extended periods of time without human intervention. The most common and essential components will be described in sections 3.1.3 and 3.1.4.

Advantages

Some examples of advantages of ASVs:

- Can remove or reduce routine tasks such as watchkeeping
- Reduces need for manual labor
- Can perform tasks that would put human lives at danger, such as minesweeping and work in radioactive areas
- Can operate for extended periods of time, and over large distances
- Removes human error
- No or little requirement to health, skill, or sobriety

Disadvantages

Some drawbacks of ASVs:

- Safety issues
- Unforeseen events
- Increased vulnerability to hacking
- Sensor dependency

3.1.2 The Otter USV

The Otter USV is a marine vessel developed by Maritime Robotics, a Norwegian company with its head office in Trondheim. The standard version is 200cm long by 108cm wide by 81.5cm tall, and weighs 55kg. It is equipped with dual electrical fixed thrusters, and has a top speed of 6 knots. The Otter is pictured in figure 3.1.

The Otter comes in two versions, which is the standard version, and the pro version. The pro version is 10kg heavier, 25cm taller, and includes AIS, camera, high-bandwith comms, and an optional SVP winch. The pro version is used in this project.

3.1.3 Sensors

LiDAR

LiDAR is an acronym for "Light Detection And Ranging", or "Laser Imaging, Detection, And Ranging". Lidar is not a component in itself, but a technique used to measure the distance to an object. It achieves this by sending out one or more beams of light, and calculates the distance based on the time traveled. Depending on the wavelength of the light, the lidar can reflect on different objects.

Figure 3.1: The Otter USV (Wevolver, n.d.)

The quality of a lidar device ranges from a simple short-ranged single-point sensor like the TFMini-S, to more advanced devices like the RoboSense RS-Ruby Laser Rangefinder. The main differences are range, accuracy, measurement rate, and field of view.

Lidar is most commonly used for mapping. For example, it can be used by attaching it to a vehicle and sent to make a 3D-projection of the area. In this project, it could be relevant in the areas of obstacle detection and auto docking.

Camera

A camera is a piece of equipment that allows the user to save images or videos of its surroundings. It works by taking in light and saving that information to a surface, which can then be stored and retrieved at a later time. The most important attributes of a camera for our purpose is field of view, resolution, and capture rate.

A camera is essential in self-driving cars, as it is the main instrument in observing its surroundings. It is vital to gathering information on where the road is (path following), and in obstacle detection. For the project in this paper, it can play a major role in obstacle detection.

Global Positioning System

Abbreviated GPS, the Global Positioning System is a system for acquiring position and time for devices using a network of satellites. A GPS is an essential piece of equipment aboard marine vessels today, as it allows the operator to know the location of a ship with an accuracy of 30cm (Moore, 2017).

A GPS will be invaluable in path planning and path following on larger paths. If a vessel is moving about in a harbor or other very restricted area, a GPS may not be the most efficient instrument. But when carrying out missions that cover a much larger area, like crossing the Atlantic, a GPS is vital for finding the way. It is also very good for emergencies, as it can broadcast its

position if it should get lost or run low on energy. This drastically improves the chances of recovering the vessel, and preventing casualties.

Inertial Measurement Unit

Abbreviated IMU, the Inertial Measurement Unit is an electronic sensor that measures forces working on it, and angular rotation. There are mainly two types of IMUs, which are the 6-DOF IMU, and the 9-DOF IMU. 6-DOF stands for 6 degrees of freedom. A 6-DOF IMU contains a gyro and an accelerometer, while a 9-DOF IMU contains a magnetometer as well. This enables it to determine its own rotation by sensing the Earth's magnetic field.

An IMU is invaluable on a marine vessel because it allows the vehicle to approximate its own velocity and rotation. This is especially useful when it does not have access to GPS. One of the biggest issues with IMUs is bias and drift. This can make the sensor unreliable over time. Higher quality IMUs have less bias and drift compared to cheaper options. A kalman filter can be used to address the issue of sensor drift.

3.1.4 Actuators

Thrusters and propellers

A propeller is a device that rotates to produce forward motion, while a thruster is a propulsion device that causes a forward movement. For example, a rocket engine is a thruster, but does not include a propeller. In the case of marine vessels, the thrusters are propellers.

A propeller is the main device used to accelerate a marine vessel. It works by converting rotating force into a linear thrust by pushing against water. The propeller is located at the rear of the vessel, and is usually paired with a rudder.

Thrusters are often located at the bow and/or the stern of larger ships. They give the ship increased maneuverability, most notably allowing it to turn on the spot. They can also be used to make the ship move sideways (sway). These two abilities are valuable when maneuvering in tight spaces, for example when docking. The biggest disadvantage of thrusters is that they are inefficient when it comes to power usage.

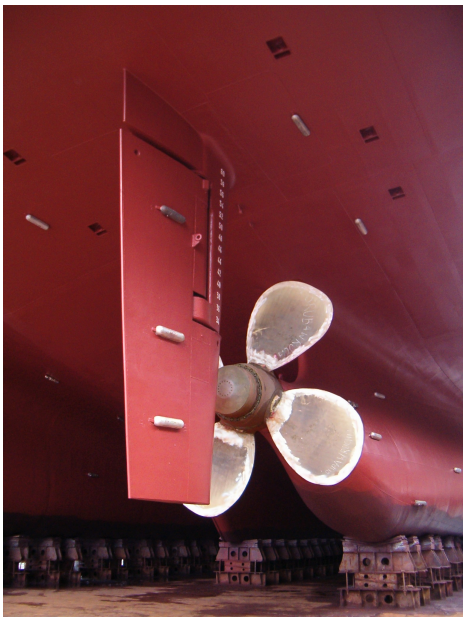


Figure 3.2: A propeller and a rudder (Wikipedia, n.d.-d)

Rudder

A rudder in its simplest form is a piece of flat material that is used to redirect the flow of a current, to change the direction of the vessel it is attached to. This makes the rudder the main component used to steer a ship. Despite being the most important steering mechanism, it actually can't produce any motion on its own. For this reason, it is usually paired with a propeller.

3.2 Ship dynamics

Ship dynamics is about studying how a ship moves and behaves in relation to the forces that affect it. Fossen, 2011a divides dynamics into two parts: *kinematics*, which treats only geometrical aspects of motion, and *kinetics*, which is the analysis of the forces causing the motion.

3.2.1 Fossens unified theory

To be able to make a system that moves the ASV in a desirable way, one should have an understanding of what affects it. Fossen, 2011a presents what he calls the "unified theory of ship dynamics" in the following equation:

$$M\dot{\nu} + C(\nu)\nu + D(\nu)\nu + g(\eta) + g_0 = \tau + \tau_{\text{wind}} + \tau_{\text{wave}}$$

Figure 3.3: Fossen's Robot-Like Vectorial Model for Marine Craft on vectorial form, Fossen, 2011a

The components are:

- M - Inertia
- $C(v)$ - Coriolis
- $D(v)$ - Damping
- η - Vector of translational position and movement
- v - Vector of rotational position and movement
- $g(\eta)$ - Vector of generalized gravitational and buoyancy forces
- g_0 - Static restoring forces and moments due to ballast systems and water tanks
- τ - The generalized forces in 6-DOF

Using this equation, we have a compact way of representing the forces that work on a ship, and how it will react. I will not go into further detail on the different parts of this equation, but interested parties are recommended to read Fossen's "Handbook of Marine Craft Hydrodynamics and Motion Control".

The equation serves as a good baseline of what forces affect our vessel. It can therefore be used as a starting point for deciding on which inputs that could be relevant for an AI model.

Marine vehicle classification

According to Faltinsen, 2005, a vessel can be placed in one of three categories. The category is determined by what is called a Froude number, and is given by:

$$Fn = \frac{U}{\sqrt{Lg}} \tag{3.1}$$

U is the ship speed, L is the overall submerged length of the ship, and g is the acceleration of gravity.

Fossen, 2011a explains the three categories as follows:

Displacement Vessels ($Fn < 0.4$): The buoyancy force (restoring terms) dominates relative to the hydrodynamic forces (added mass and damping).

Semi-displacement Vessel ($0.4 - 0.5 < Fn < 1.0 - 1.2$): The buoyancy force is not dominant at the maximum operating speed for a high-speed submerged hull type of craft.

Planing Vessel ($Fn > 1.0 - 1.2$): The hydrodynamic force mainly carries the weight. There will be strong flow separation and the aerodynamic lift and drag forces start playing a role.

The Otter USV has a length of 2 meters, and a top speed of 6 knots (ca 3 m/s). Calculating the Froude number at top speed gives:

$$Fn = \frac{3}{\sqrt{2 * 9.81}} \approx 0.677 \tag{3.2}$$

This classifies the Otter as a semi-displacement vessel. However, if we operate at a speed of 3 knots (ca 1.55 m/s), we get:

$$Fn = \frac{1.55}{\sqrt{2 * 9.81}} \approx 0.35 \tag{3.3}$$

This means that if we operate the Otter at a maximum speed of 3 knots, we can treat it as a displacement vessel. As stated above, in a displacement vessel the dominating force is the buoyancy. With semi-displacement and planing vessels, the hydrodynamic force has a much bigger impact. The reason this classification is important is because it determines what the dominating forces are. This can in turn help determine what inputs should be focused on for the AI model. The forces involved will be described in more detail in section 3.2.5.

3.2.2 Degrees of freedom

In ship dynamics, degrees of freedom is the number of axes a ship can either move along (translate) or around (rotate). Somewhat simplified, a train has one degree of freedom because it can only move straight forward. A car has two degrees of freedom, since it can move forwards and backwards, and turn around one axis. The highest degree a ship (or any physical object) can have is 6-DOF, meaning it can both move along and rotate around the X-, Y-, and Z-axes.

For any ship to function properly, it is necessary to have at least two degrees of freedom (2-DOF), which enables it to theoretically move to any position in the water. By adding bow and stern thrusters, the ship gets three degrees of freedom (3-DOF), and allows it to moves sideways (sway).

As an ASV only operates on the surface of the water, only 3-DOF are relevant. The heave, pitch, and roll movements can be used to move under water, but have little value on the surface apart from stabilizing the ship. This can be a very desirable attribute because these motions are usually what causes sea sickness. It is not relevant for this project, but should be implemented at a later stage when adding human passengers.

3.2.3 Reference frames

To describe the motion, position, and orientation of a vessel, it is necessary to use a reference frame. When describing an object's position, velocity, or orientation, it is always in relation to something else. In this section, the most

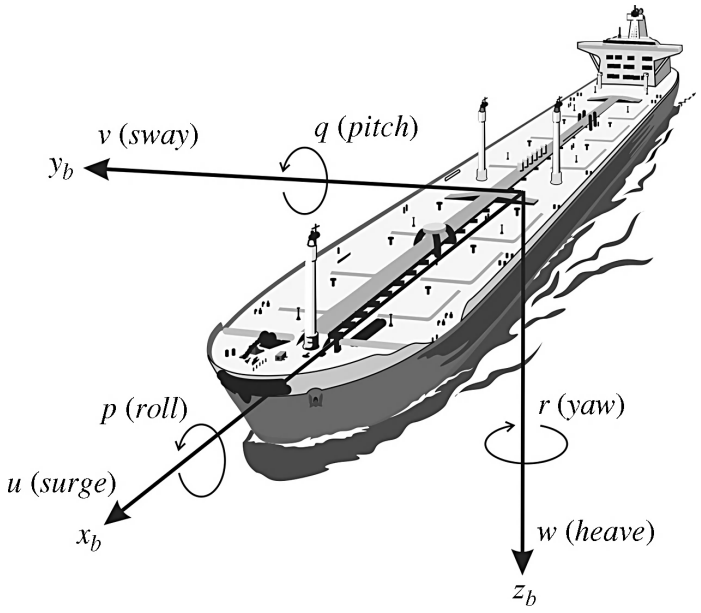


Figure 3.4: Visual representation of the DOF that a ship can have (Fossen, 2011b)

common reference frames used in maritime navigation are described, and the ones that are the most relevant for this project.

The most notable importance of reference frames is the conversions between them. For example, it is very common to use a BODY reference frame when representing velocities and forces on a ship, but GPS coordinates use an ECEF reference frame, so it is often necessary to be able to convert between these.

ECI

The Earth-centered inertial (ECI) reference frame has its origin at the center of the Earth, and has a fixed orientation in relation to extraterrestrial objects not affected by the Earth's rotation. For example, one can imagine standing on a stationary space station, and looking down on Earth. The Z-axis runs from the center of the Earth and has a positive direction through the North Pole. The X-, and Y-axes pass through the equator.

ECEF

The Earth-centered Earth-fixed (ECEF) reference frame is very similar to the ECI-reference frame, except that it is fixed to a point on the Earth's surface. Like with ECI, the Z-axis runs from the center of the Earth through the North Pole in the positive direction, and the X-, and Y-axes run through the equator.

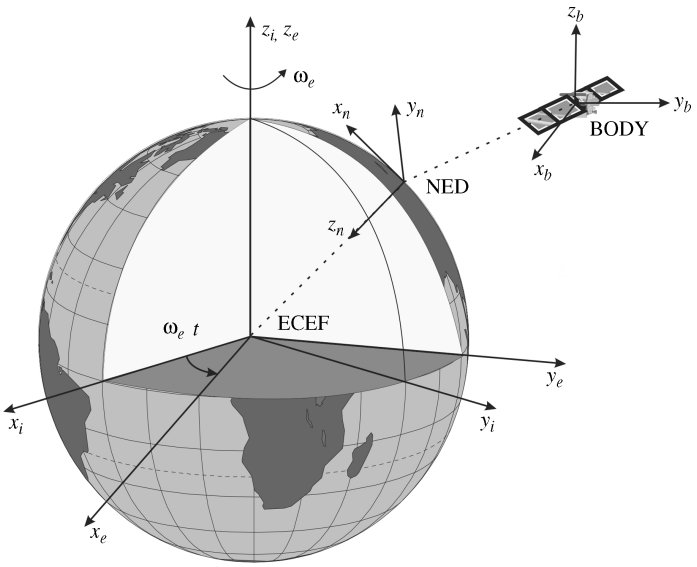


Figure 3.5: The ECI, ECEF, and BODY reference frames (Fossen, 2011c)

The X-axis runs from the center of the Earth and through 0° latitude and 0° longitude. The GPS coordinate system uses an ECEF reference frame.

NED

The North-East-Down (NED) reference frame is most commonly used in relatively small places, such as a small forest or a harbor. If you move along the X-axis in a positive direction, you are moving along the ocean surface towards the North Pole. One thing about this reference frame that may be counter-intuitive is that the z-axis points downwards, meaning that a value of +15 will be 15 units below sea level, and not 15 units up in the air.

ENU

The East-North-Up reference frame is very similar to the NED reference frame, except that the directions are switched. The first value indicates a direction where the positive direction is east, the second value has a positive direction north, and the final value has a positive direction upwards (away from the Earth's centre).

BODY

BODY is a body-fixed reference frame that has its origin fixed to a position relative to the ship. This means that when you are standing in the middle of the ship, you are at position (0, 0, 0). When looking forward, you are looking

along the positive x-axis. Looking to the right (starboard) is the positive y-axis, and the positive z-axis is straight downward.

GPS

GPS (Global Positioning System) is one of the most widely used and most well-known reference frames. It uses a system that is based on degrees, which is the number of degrees from the prime meridian and the equator. This is known as latitude and longitude. For example, the OsloMet Ocean Lab is located at approximately N59° 54.524', E10° 43.141'.

3.2.4 Representing pose and velocity

Orientation matrix

An orientation matrix (for 3 dimensions) is a 3x3 matrix that has the following properties:

$$RR^T = R^T R = I$$
$$det(R) = 1$$

A rotation around the x-, y-, and z-axis will look like this:

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

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

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

Figure 3.6: Rotation matrix around x-, y-, and z-axis

In the above matrices, θ is the angle around the corresponding axes. An object that is oriented with 0 radians around every angle will have a rotation matrix equal to the identity matrix $I(3)$. The rotation matrix is based on Euler angles, which are three angles introduced by Leonhard Euler to describe the orientation of a rigid body with respect to a fixed coordinate system (Wikipedia, n.d.-a).

Pros and cons:

\+ Relatively intuitive

+ Easy to implement

– Suffers from what is called a Gimbal lock, or singularity

A Gimbal lock is simply a loss of degree of freedom, meaning that there is a configuration where a system is unable to move in a certain direction.

Quaternions

Strictly speaking, quaternions are an extension of imaginary numbers, that has one real part and three imaginary parts. Quaternions are usually written on the form: $a + bi + cj + dk$ where a, b, c, d are real numbers, and i, j, k are unit vectors pointing along the three spatial axes.

In the case of rotation, quaternions can be used to represent an orientation. The mathematics behind this is pretty advanced, so we dare not go into the details of this here.

Pros and cons:

+ Avoids the singularity problem

+ Faster computation than rotation matrix

– Hard to get an intuitive understanding because of 4 dimensions

3.2.5 Forces that act on a ship

Wind

Wind is simply the collective movement of gas particles. While every particle with a temperature different from 0 kelvin moves randomly, wind is recognised by the movement of many local particles in the same direction. This phenomena usually occurs because of differences in pressure.

Depending on the strength, wind has a major impact on how a ship moves in the water. Wind that is coming directly from the side can effectively push a ship off its course.

Drag

Drag is a force acting opposite of the direction of the movement of an object through a fluid. It can be thought of as the fluid equivalent of air resistance. Drag is the force that the bow of a ship experiences when moving through water.

Buoyancy

Buoyancy is a force directed upward that is exerted on an object that is immersed in a fluid. Archimedes' principle states that *Any object, wholly or partially immersed in a fluid, is buoyed up by a force equal to the weight of the fluid displaced by the object.*

Waves

Waves are fluctuating hydrodynamic forces. Waves have a frequency greater than 0, and will move a vessel up and down, and in the direction the wave is moving. The force applied to the vessel is therefore time-variable. The most important characteristics of a wave are the amplitude and the period.

Currents

Currents are constant hydrodynamic forces. This means that there is a continuous stream of fluid from one place to another, and will therefore provide a constant force against the vessel. The most important characteristics of a current are the cross-section area, and the speed.

3.3 Path planning

3.3.1 What is path planning?

Path planning (also called motion planning) is the act of finding a set of movements or actions that will get an object to reach its destination. This can be an entire object, or a part of a bigger object, like a robot's "hand". Depending on the environment and the object to be moved, path planning can be very computationally expensive. The object in this project is restricted to a confined area, as an ASV only operates on the surface. This means that the path planning can be limited to a 2D plane. Before beginning the path planning, one must have some knowledge about the object to be moved (in this case the ASV), and the environment (the ocean/water where the ASV can move).

The most important thing regarding the object is how it can move, which is usually represented as degrees of freedom (as mentioned in section 3.2.2). This is important because it limits the number of possible actions the object can take to reach its destination. With robots and other multi-jointed objects, the possible configurations and shapes of the object must also be considered. In other cases, attributes like size and weight become important.

The most important thing to know about the environment is where the obstacles are. This requires a description of the area where the ASV will move. For a 2D space, this can be represented by a map. The map/description must contain what is considered "free space", and what is considered "obstacle space". Free space is anywhere in the environment where the object can move to without colliding. Obstacle space is any place the object can not move to. In the case of an ASV, the free space will be open water, and obstacle space will be land and constructions in the water like docks.

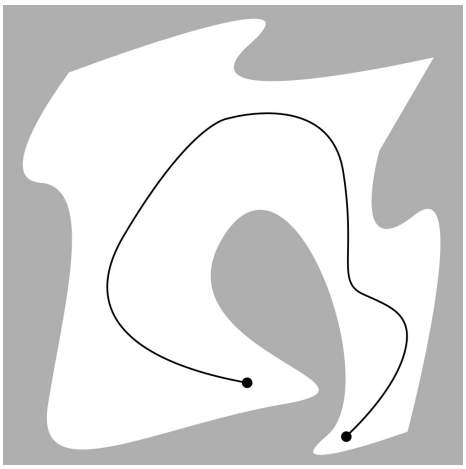


Figure 3.7: Example of a path (Wikipedia, n.d.-c)

3.3.2 Path planning methods

Grid-based decomposition

Grid-based decomposition is an important concept related to path planning. It is not a path planning method by itself, but the concept provides the basis for many path planning methods, which is why it is necessary to mention it first.

Grid-based decomposition is a way of turning an environment with continuous space into an environment with discrete positions. A common example of this is the way a world map is divided into squares based on coordinates. These discrete positions are often used to make up waypoints, and many path planning methods are based on finding the optimal set of waypoints.

Dijkstra's algorithm

Dijkstra's algorithm is a path-finding algorithm created by Dr. Edsger W. Dijkstra, published in 1959. It is an algorithm that finds the shortest path from a source node to all other nodes in a network.

From Navone, 2020:

- Dijkstra's Algorithm starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.

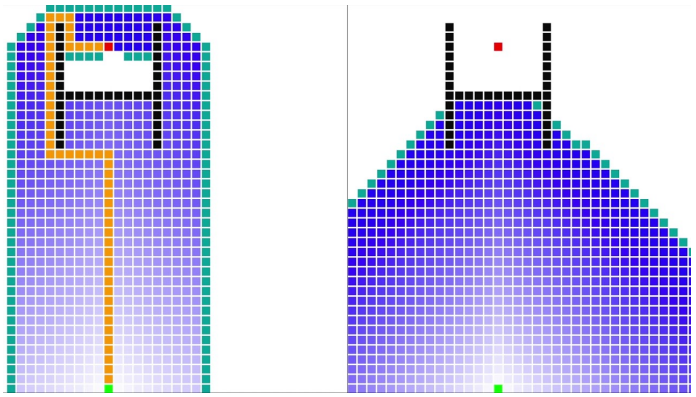


Figure 3.8: A* search algorithm (left) vs Dijkstra's algorithm (right) (Robots, 2019)

- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Pros and cons:

+ Finds the shortest/cheapest way to the destination

– Not deterministic

– Can take a long time to find the optimal route

As this algorithm is not deterministic, it will not look for a specific node. This is a disadvantage when one is only looking for the destination node. An exception is if you stop the algorithm when the destination node has been reached. However, the algorithm is useful for finding the distance to several nodes. For example when you want to find the route to all harbors in a region.

A practical example of the use of Dijkstra's algorithm can be found in Wang, Yu, et al., 2011, where it is used to navigate a robot through a room with physical barriers.

A* planning

A* can be seen as an improvement or extension to Dijkstra's algorithm. The biggest difference is that the A* algorithm includes a deterministic element that rewards a node based on the distance it has from the destination node. This can be mathematically expressed as $f(n) = g(n) + h(n)$, where $g(n)$ is the length of the path from the origin/source node, and $h(n)$ is a heuristic function that estimates the cheapest path from n to the goal (Wikipedia, n.d.-b). Dijkstra's algorithm can be seen as a special case of A* where $h(n) = 0$.

Pros and cons:

+ Relatively fast

+ Deterministic

+ Will always find a route to the destination (if it exists)

– Memory heavy

Reinforcement learning

Reinforcement learning is an area of machine learning where an agent acts in an environment and attempts to maximize a reward (and/or avoid a punishment). The agent is awarded when it performs a desirable action, and punished if it does something undesirable. In the case of finding a route to a destination in a homogeneous terrain, one can set the reward for reaching the destination to 1 and everything else to 0. Additional positive/negative rewards can be implemented, like the amount of fuel used, time spent, or whether or not it crashed.

There is a trade-off between exploration, and exploitation. Exploration means that the agent will test out new actions, and exploitation means picking the best action. This ratio between exploration and exploitation can be adjusted using a coefficient $0 < \varepsilon < 1$ that determines the chance of an agent performing a new action (exploring). In path planning, one can set the coefficient close to 1 to make the agent mostly explore new ways. When it is time to perform the path following, the coefficient is set to 0 and the agent will always follow the optimal route it has found to its destination.

Pros and cons:

+ Will constantly improve

+ Can find paths that do not seem intuitive

+ Can include multiple conditions beyond distance

– Can take a very long time to find an optimal or good route

Reinforcement learning has been used in Guo et al., 2020 for autonomous path planning.

3.4 Path following

3.4.1 What is path following?

Path following is the process of moving along a predetermined route. The main goal of path following is for the object in question to arrive at its destination, but additional considerations can be included, like turning rate and energy consumption.

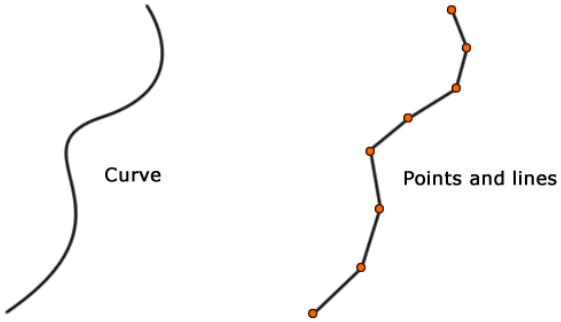


Figure 3.9: Example of waypoints making up a path (Bevilacqua, 2013)

One of the most important things to consider when choosing a path following method is the object itself. Its construction, and therefore its degrees of freedom, determine how it is able to follow a path.

3.4.2 Path following methods

Waypoints

Waypoints is not a path following technique in itself, but it is a concept that many path following methods are based on. For this reason, it is introduced here. A waypoint is *an intermediate point on a route or line of travel* (Mirriam-Webster, n.d.). Each waypoint can be considered as a node, and several waypoints make up a path, see figure 3.9.

Waypoints are represented by coordinates in a physical space, and represent a position in the environment. The size of the waypoint itself can vary. The size is usually the acceptable range for where the object is considered to have reached the waypoint. For example, when driving a car across the country, the waypoint can be a town or gas station. When moving a robot arm, the waypoint can be as little as a few centimeters or millimeters. In the case of grid-based decomposition (discussed in 3.3.2), a waypoint can be a square on the map, which represents a n*n sized area.

The simplest way of using waypoints to navigate is that the vessel drives towards the waypoint in a straight line, and turns when it reaches the waypoint. It then steers straight towards the new waypoint, and repeats this process until it reaches the final waypoint.

Lane following

This is the method you use when driving a car, where you want the car to stay within the borders of the road. Whereas a line following algorithm will attempt to steer a vessel along a specific line, a lane following algorithm will attempt to keep the vessel within two (usually parallel) lines.

A simple way to create a lane is to draw a line from the outer edges of the current waypoint to the next one.

There are usually two variants to the lane following method. In the first one, the vessel will turn when crossing or approaching the border line. In the second one, the vessel will attempt to stay in the middle of the two lines. This can be considered a special case of line following, with the extra step of calculating where the line is.

Pros and cons:

+ Gives the vehicle flexibility in terms of where it can go

– Can make the path longer depending on the variant used

Angular direction path following

In angular direction, a direction represented by an angle is used to determine the direction the vessel needs to head to reach its destination. The angle to the destination is compared with the current heading of the vessel, and the vessel will turn based on the difference. This method makes the vessel draw a straight line from its current position to its destination. This is different from a line following algorithm, where a line is drawn between the two waypoints, and the vessel attempts to approach that line.

Pros and cons:

+ Simple calculations

+ Only relies on GPS and compass (gyroscope)

– Can make some awkward turns at each waypoint

Line-of-sight path following

Line-of-sight (LOS) path following is a method used to minimize the cross-track error to the path between two waypoints. This is done by heading the vessel towards a point p_{los} which is located on a line between the previous and next waypoint, see figure 3.10. The concept of LOS path following for underactuated marine vehicles is introduced by Fossen et al., 2003, and the interested reader is encouraged to read this paper for a closer look at the mathematics behind the algorithm.

Pros and cons:

+ Can be integrated with default autopilot systems

+ Makes smooth turns at each waypoint

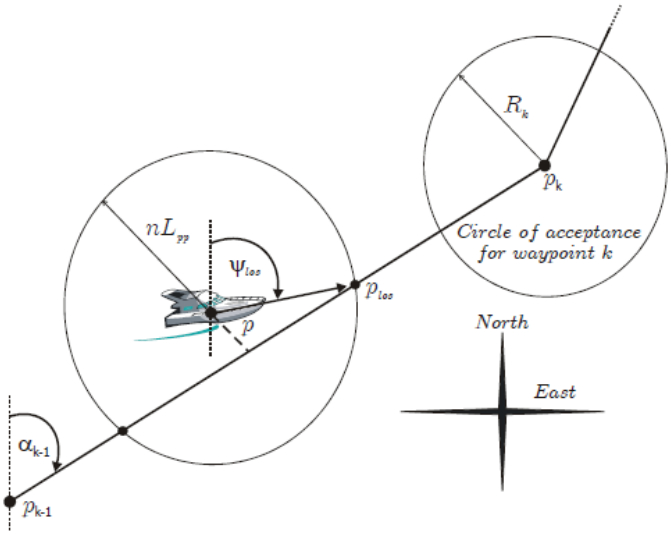


Figure 3.10: Illustration of the LOS path following method (Fossen et al., 2003)

This method makes the vessel perform smooth turns at each waypoint, meaning it starts its turning before reaching the actual waypoint. This means that the vessel will not have to stop at each waypoint, turn, and resume driving. However, this can cause some complications depending how the waypoints are placed. For example, if a waypoint is placed too close to a corner, the vessel runs the risk of crashing into the corner if it starts turning too early.

Reinforcement learning

When using reinforcement learning, the agent (our vessel) can be rewarded positively for reaching the destination waypoint, staying close to the path, and/or moving in the direction of the path. This requires many of the same calculations as the above methods, but the main difference is that the agent teaches itself the optimal strategy for navigating.

An example of using (deep) reinforcement learning in path following (combined with collision avoidance) can be seen in Meyer et al., 2020.

Pros and cons:

+ Will improve continuously over time

+ Can form policies that will work in unknown environments

+ Can be combined with collision avoidance

– Can take a long time to find a policy that works

3.5 Obstacle detection

3.5.1 What is obstacle detection?

According to Matthies, 2014, obstacle detection can be defined as follows: *Obstacle detection is the process of using sensors, data structures, and algorithms to detect objects or terrain types that impede motion.*

On the open ocean, there is very little terrain that can affect the vessel on its journey. But when navigating through shallow waters, the ground becomes an important hindrance. Crashing into land is an obvious example, but a ship can also quickly get stranded or receive critical damage if it drives over an area where the vessel is deeper than the water. For this reason, the ASV needs to have a system for obstacle detection that can cover both above and below the surface of the water.

The main thing that makes obstacle detection challenging is the variety of obstacles that can appear. Just spending a few hours looking at the ocean from Tjuvholmen in Oslo, one can see swimmers, paddleboards, kayaks, and ship and boats that come in all shapes and sizes. The vessel therefore needs to be able to detect all these objects to safely avoid collision. In this project, only ships will be considered for detection.

3.5.2 Obstacle detection methods

There are three major categories of obstacle detection methods. Distance-based, vision-based, and feeling-based. There are several methods within all these categories. To get a better overview, the categories themselves will be reviewed, and not the individual methods within each of them.

An important remark is that these three categories are not mutually exclusive. This means that a hybrid of two or three of these methods is not only possible, but can vastly improve the overall ability of the vessel to detect obstacles.

Distance based

In distance-based obstacle detection, a lidar is used to measure the distance to its surroundings. It returns a point map of the distance at every angle, as seen in figure 3.11. By analyzing this map, the vessel can "see" which way it can go. An example of how a lidar can be used to interpret its environment can be found in Asvadi et al., 2016.

A disadvantage to the distance-based method is that it can become unreliable when facing waves and other uneven terrain. A wave is not an obstacle in itself, but can be interpreted as such. Additionally, if the vessel is approaching land and only relies on the Lidar, it can be very difficult to actually see the land if it has a very gradual slope, like at a beach.

Pros and cons:

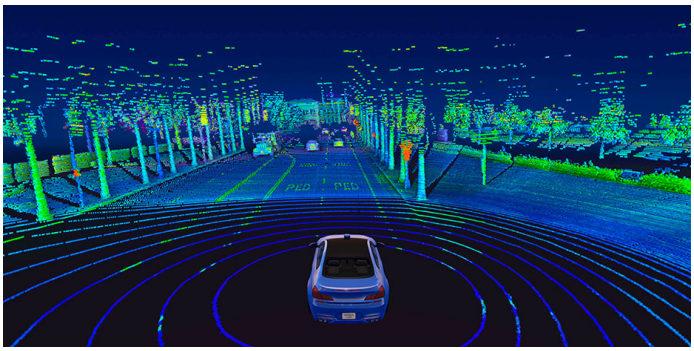


Figure 3.11: What a Lidar "sees" (Lidar, 2021).

+ Can get an accurate distance to objects
+ Works well in low-light areas
+ Can be used underwater
– May interpret something as an obstacle that isn't
– Difficult to detect land with a gradual slope

Vision based

Vision-based obstacle detection relies on cameras to perceive the environment, rather than lidars. An important distinction between vision-based and distance-based is that the vision based requires additional processing before being able to make a decision. For example, one has to first scan the picture for *potential* obstacles, and then determine whether or not it actually is. A lidar could simply check if an obstacle is too close. Additionally, obstacles can come in all shapes and sizes, so it can be difficult to correctly classify entirely new obstacles.

There are several methods within vision-based obstacle detection. One of these is based on the color of surrounding objects. For example, in Ulrich and Nourbakhsh, 2000, a robot is trained to recognize the ground based on its color. This is a method that can work quite well indoors where the floor has a distinct color from other objects. But in water, there is a lot of overlapping color. Additionally, water often reflects light and other objects, making it even harder to distinguish.

Another example is in Wang, Wei, et al., 2011. By using different filters on the photos, they are able to highlight potential obstacles. For example, they use something called "saliency detection", meaning that the vessel can detect objects that stand out in the sea landscape, by looking at color and texture patterns. They had good results with being able to detect obstacles up to 300m away.

Pros and cons:

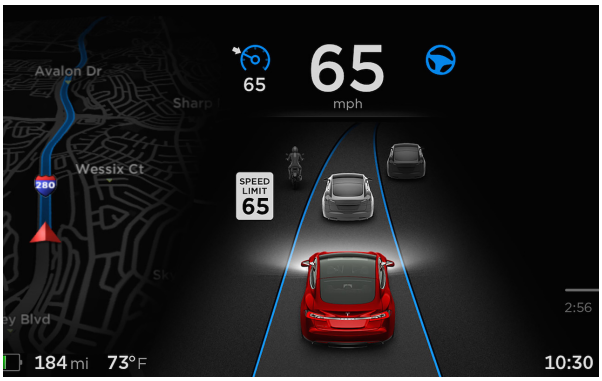


Figure 3.12: The Tesla autopilot (which is vision-based) in action, having detected two cars and a motorcycle (Falson, 2016)

+ Good resolution
– Requires a certain amount of processing power
– Can be difficult to 100% recognize what is an obstacle
– Vulnerable to weather conditions (foggy lens for example)

Feeling based

Feeling-based obstacle detection involves using some kind of stick or rod to feel for obstacles. While it can be impractical in many open-sea situations, it is worth a mention because it can have some neat applications in auto docking and other situations with low velocity and distance. For example, they can be extended when approaching a dock or beach to get an accurate measure of where the ground/dock is.

Pros and cons:

+ Doesn't necessarily require electronic devices
+ Reliable
+ Works in practically all weather conditions
– Very low range

3.6 Collision avoidance

3.6.1 What is collision avoidance?

Collision avoidance is when the ASV performs actions that steers it clear of any object that is on collision course. Collision avoidance involves determining the

position and movement of the obstacle, finding the necessary actions, and then performing these.

Obstacles can be divided into two categories, which are static and dynamic obstacles. This distinction is very important, as it determines the complexity of avoiding it. If an obstacle is stationary, the vessel only needs to brake and/or turn. If the obstacle is moving, its speed and direction must be estimated before an action can be taken.

International Regulations for Preventing Collisions at Sea (COLREGs)

The International Regulations for Preventing Collisions at Sea (COLREGs) is a convention from 1972 that is published by the International Maritime Organization, which is a specialized agency of the United Nations. The COLREGs contains rules that maritime vessels must follow, with the purpose of preventing collisions. This includes rules on steering, speed, and signals. This convention often serves as a benchmark for the requirements of an ASV.

3.6.2 Collision avoidance methods

Emergency brakes

A solution implemented in most cars today, the emergency brake is perhaps the most (in)famous example of collision avoidance. The emergency brake protocol is used when there is an object in front of you that is approaching at such a speed that collision is imminent. There are a few variations on this concept, but the most basic one is fully engaging the brakes when a distance sensor detects an object that is within a certain distance. More advanced versions include factors like vehicle speed, obstacle speed, object recognition etc.

Pros and cons:

\+ Easy to implement

– Can be quite uncomfortable for passengers/goods

Velocity Obstacle

In the Velocity Obstacle (VO) method, a cone-shaped obstacle in the velocity space is generated, and ensures that there will be no future collisions as long as the vessel's velocity vector is outside the VO (Kuwata et al., 2013). This means that our vessel calculates a range of velocities (with direction) that will cause it to crash with the obstacle, see figure 3.13.

This method is in itself a method for checking whether or not a (moving) obstacle will collide with the ASV at its current speed. To actually avoid the collision, the velocity must be changed, which can be done by braking or turning. One option is to simply take the action that requires the least amount of change in velocity for the ASV. But compliance to the COLREGs can be added by performing an additional step that check which actions are in compliance.

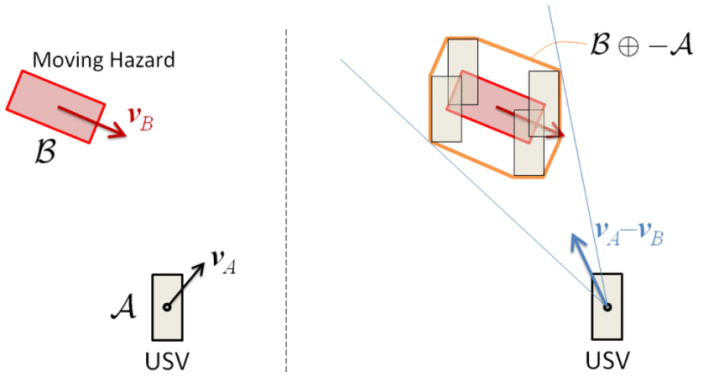


Figure 3.13: Graphical interpretation of VOs (Kuwata et al., 2013).

Pros and cons:

+ Can predict arbitrary trajectories

+ Can implement compliance with the COLREGs

– Computationally expensive

A* algorithm

The A* algorithm (discussed in section 3.3.2) can also be used for collision avoidance. To do this, an aerial view is produced and sent to the vessel. Then, the A* algorithm is used to find a path that avoids the obstacles detected. As an additional feature, the most likely trajectories of the detected obstacles (other ships for example) can be estimated, and this information can be incorporated into the A* algorithm. This is done by calculating in time-steps and making certain areas restricted space at certain times. The A* algorithm is used as a verification in Schuster et al., 2014.

Pros and cons:

+ Can be used as a combination of path planning and collision avoidance

– Requires aerial view of the area

Model predictive control

Model Predictive Control (MPC) is a forecasting algorithm that attempts to determine the optimal configuration for a set of variables based on future states. In slightly simpler terms, it attempts to find the best action for your system, based on the immediate future. This is done by calculating the outcome of different scenarios several timesteps ahead, and applying the first action of the

best scenario. This process is repeated for as long as the system is active, at every timestep.

Pros and cons:

+ Can include constraints on the movement of the vessel

+ Can control multiple input, multiple output (MIMO) systems

– Requires external computation of the obstacle's velocity and position

– Computationally expensive

Reinforcement learning

As with path planning and path following, reinforcement learning can also be applied to the field of collision avoidance. The same basic principles are used. Given a state x, an action a is performed, and the outcome is evaluated and used as a feedback to the agent performing the action. In the case of collision avoidance, the feedback will be based on whether or not the vessel crashed, or how close it was to crashing.

Additionally, the vessel can be "punished" based on the steepness of the turn or brake. This means that it is possible to prioritize what is most important. Obviously the most important thing is to avoid a collision, but the agent may learn that using the emergency brake every time is the most efficient, but this is not an optimal solution for passenger ferries for example.

While reinforcement learning is a very promising technique, it does have a few limitations. One of the most important ones is safety. Computing every single scenario and action is intractable, meaning there will be situations that the agent has not prepared for. In this case, the agent will "guess" what the best action is, which can be very risky in certain situations. This can be addressed by using Deep reinforcement learning, which is basically a combination of reinforcement learning techniques and deep neural networks.

In Huang et al., 2005, it is shown that a robot can successfully learn to avoid obstacles in a new environment using reinforcement learning techniques. While successful, it is important to note that this is done in a simulation only with a very simple robot.

Another example is from Wooa and Kim, 2020, where they use deep reinforcement learning to not only simulate, but actually perform live experiment on a USV. To quote their paper: *Throughout the full-scale USV collision avoidance experiments under various scenarios, the applicability and the effectiveness of the proposed method in the real-world domain was verified.* This means that according to their experiments, using deep reinforcement learning for collision avoidance is a valid and working technique.

Pros and cons:

+ Can be used as a combination of path following and collision avoidance

+ Can take multiple considerations into account

+ Can individually adjust the punishment for specific actions

– Training in simulation may not reflect real world behavior

– Can be computationally inefficient for simple problems

3.7 Auto docking

The goal for the ASV in this project is to safely approach a dock, and position itself in such a manner that it can be attached there. This can for example be close enough for someone to board it, attach a rope to it, or retrieve it. The success of the docking will depend on whether the ASV was damaged in the process, the dock and anything on and around it was damaged, and how close the ASV was able to get to the designated docking space.

An important thing to take into account in the docking process is the obstacles that can exist in the docking area. In a simple pier that is sticking out in the middle of the ocean, the docking process can be very straight-forward. But in more complex environments like Frognerkilen (figure 1.1), there can be multiple obstacles including other boats.

The auto docking method used will depend on how the ASV will dock. In general, there are two ways for the vessel to dock. The first one is simply driving up to the dock and stopping with the bow facing the dock, which is what is done by many ferries around Norway (like the Nesodden ferry). The second method is when the vessel approaches the dock, and it docks by placing its side to the dock.

Auto docking involves path planning, path following, obstacle detection, and collision avoidance. This means that we can treat auto docking as a separate case of autonomous transportation, by using techniques from the previous sections in this chapter. The main difference is that the destination requires a certain orientation, and not only position. We can define the auto docking process as everything that happens after the ASV approaches the docking area, until it is docked.

According to Bitar et al., 2020, the number of reported existing methods is limited in research literature and in commercial applications. Some methods have been tested on autonomous underwater vehicles (AUVs), there has been conducted experiments that use neural networks, and MPC-controlled optimal path following methods. The main issue with these experiments is that they do not account for obstacles in the docking area, which is very important for real-world applications, especially if the ASV needs to operate in areas where there are humans present.

What Bitar et al., 2020 does is to modify a method used by Martinsen et al., 2019, that also includes the ability to take dynamic obstacles into consideration. A potential drawback with this method is that it requires a geographical map of the docking area. This is fine in most cases, but the issue is that the method

relies on some prior knowledge. This means that it will be necessary to create a dynamic map that can be updated live as obstacles are detected.

This can also mean that a part of this project will be to develop an auto docking method that can work in areas where the environment is partially unknown. A geographical map of the docking area is allowed, but the important thing is that the ASV needs to be able to detect and react to obstacles as it goes.

Chapter 4

Initial method selection

In this chapter, the methods chosen as the initial methods will be presented. This chapter will be relatively brief, as the implementation and details of the methods will be discussed in further detail in their corresponding sections in chapter 5.

In general, there has been a strong emphasis on choosing simple methods that are easy to implement. This is because it was considered desirable to do rapid prototyping, and make a Minimum Viable Product (MVP). It is understood that the methods chosen are meant as a starting point, and will be further developed in future work.

4.1 Path planning

For path planning, the A* algorithm is chosen. The main reason for this is that it is the fastest of the three methods mentioned (A*, Dijkstra's, reinforcement learning). This gives the advantage that it can be easily calculated on-board, which in turn makes it viable to do re-calculations during the journey. This is a benefit as this means it can potentially be combined with collision avoidance.

The A* algorithm requires a grid-decomposed map, but this is expected to be easily obtainable as the world is already grid-decomposed by coordinate systems.

Another benefit is that since the A* algorithm is an established method, it should be easy to find a working solution to implement. This can potentially save a lot of development time, and means that more focus can be put into the more difficult parts of the project.

4.2 Path following

The Otter has a pre-programmed mode called "Station mode", which drives it in a straight line to the destination. This resembles an angular direction path following method.

Initially, the idea is simply to use the Otter's station mode to drive to each waypoint. This means that it will drive in a straight line towards each waypoint. This in turn means that the total movement may not be completely smooth. However, this should be fine as it is only meant to be a starting point. The biggest advantage with this approach is that it saves a lot of time, and more focus can be directed towards other parts of the project that may require more work.

As with path planning, this method of path following is very simple, and should not be viewed as a final and definite version. For now, it is simply used as a starting point to create something that is minimally viable. As with all parts of this project, it is expected that this method will be improved upon in future work.

4.3 Obstacle detection

The method that can be used for obstacle detection is limited by the equipment available on the Otter. The Otter comes delivered with a camera, which means that computer vision methods can be used. One disadvantage with the camera is that it has a very narrow field of view. To comply with the COLREGs, it is desirable to install more cameras to be able to "see" what is needed in order to detect the relevant obstacles.

When choosing a method for detecting obstacles, it is also necessary to take into consideration which obstacles that are to be detected. There is a long list of objects that can be potentially found in the ocean/fjord.

- Large boats (like cruise ships)
- Personal boats
- SUP (Stand Up Paddleboards)
- Kayaks
- Swimmers
- Skerries
- Diving flags
- Buoys

And this is only a short list of all the possible obstacles one can encounter. It would be infeasible to create an algorithm that can detect any object, it would simply take too long for this project. Therefore, the initial method for detecting obstacles will be restricted to detecting the most common obstacle found in the Oslo fjord, which is ships.

YOLOv5

Based on the reasons given above, the YOLOv5 net will be used, which is an acronym for You Only Look Once version 5. The original paper is written by Redmon et al., 2015. The VOLOv5 net is used for classifying and detecting objects in an image, and is trained to categorise 80 different objects, where one of them is ships (another being humans).

The YOLO model has become popular for two main reasons. It is very fast, and it is open source. The YOLO model presented in the original paper (Redmon et al., 2015) processed images at 45 frames per second. This is very fast, which is be very useful when doing live obstacle detection.

Another benefit with the YOLO model is its ease of implementation. This is because it is included in the "torch" python library, which we will see in section 5.4.

4.4 Collision avoidance

The simplest way of performing collision avoidance is by treating every obstacle as static, and driving around it. Therefore, this will be the initial collision avoidance method to be tested in this project. One way to accomplish this is by setting the Otter to "manual mode" and driving to the right of the obstacle (because of the COLREGs). Between each waypoint, the Otter can check for obstacles. If an obstacle is detected, it can manually drive to its right, and plan a new path once the obstacle has been passed. The position of the obstacle will have to be evaluated continuously until it has been passed.

This method can work fairly well if all detected obstacles are static. Unfortunately, most ships you encounter on the ocean are moving. To avoid moving obstacles, it is necessary with some information on the distance from the Otter, how fast it is moving, and in what direction. All this information requires more sensors than is available for this project, and is therefore beyond the scope of this project.

An additional challenge is that it is very difficult to predict human behavior in a simulation. It is possible to make a simulation environment with randomly placed static obstacles, or even moving ones in simple patterns, but simulating complex behavior (like humans driving) can be too complex for simulation and training purposes. This is another reason why this project is restricted to only simulating static obstacles.

4.5 Auto docking

Deep reinforcement learning will be used as the initial method for auto docking in this project. More specifically, deep Q learning.

There are two main reasons for this. One: reinforcement learning can be extended to work for the whole trip. Two: with a correctly trained model, the Otter can adapt to docking in an arbitrary dock.

The first reason means that it is possible to make a holistic model for an ASV. The Otter can begin by training to simply drive into a berth and stay there. After that, it can start further away and at a different angle. Then, static obstacles can be added, and the Otter can train to avoid these, and so on. Finally, the Otter can hopefully be trained to perform a full tour from start to finish.

The second reason is a consequence of the first reason. With a well-trained model, the Otter would simply need the position and angle of the berth to dock there. This can potentially make it very easy to add new docks to the vehicle's available destinations.

Chapter 5

Setup

In this chapter, more details on the initially chosen methods are given, along with their implementation. This also includes the most relevant equipment and software needed to use these methods.

5.1 The Otter interface

The Otter interface is any piece of software that is related to communicating with the Otter. The hardware inside the Otter will not be covered in this report.

5.1.1 TCP/IP

Transmission Control Protocol over Internet Protocol (TCP/IP) is a communication protocol used to communicate over the internet and similar computer networks. It works by establishing a connection between a client and server, and "wraps" data packages in metaphorically the same way a letter is wrapped in an envelope. The protocol writes the (IP) address of the receiver on the envelope, and transmits it over the internet (or local network).

The main alternative is User Datagram Protocol (UDP/IP), which is an insecure way to transfer data. This simply means that the protocol does not guarantee the arrival of all data packages sent.

The reasons for choosing TCP over UDP is because there is no strict time frame when transmitting commands, and it is important to make sure that all the relevant information arrives correctly. For example, you would rather have the Otter receive the correct coordinates one second later, than having it arrive with 1 longitude degree off.

5.1.2 NMEA 0183

NMEA 0183 is as a format that is a standardized way to communicate between electronic devices used in maritime environments, like GPS and sonars. This is the message format the Otter uses to receive and broadcast messages.

An example of a message sent as a NMEA 0183 datagram could be:

$PMARMAN,<force_x>,<force_y>,<torque_z>*<checksum><CR><LF>

Where checksum is the checksum of the message up to and including the asterisk symbol, CR is Carriage Return, and LF is Line Feed.

In this particular message the Otter is told to set a linear force on the thrusters, and or a torque.

The way this is implemented in this project is by writing a Python function for each message type, and returning a message on NMEA 0183 format. The functions transmits the constructed message to the Otter over TCP/IP, and returns a Boolean value indicating whether or not the transmission was successful.

5.1.3 Python

Python is an interpreted high-level general-purpose programming language. According to StackOverflow's annual survey, Python is one of the most popular languages out there (StackOverflow, 2020).

The reason for choosing Python for this project is mainly because it is one of the most popular languages used in Artificial Intelligence and Machine Learning, with a wide selection of supporting libraries. Additionally, writing code is fast and easy (compared to many other languages), which means less time can be spent on writing code, and more time can be spent on exploring different solutions. Another big advantage is that Python code written on a Windows computer can be easily transferred to a Linux or iOS computer and run there.

One disadvantage of Python is that it is relatively slow at execution-time compared to other languages like C/C++, which is compiled. This shouldn't be an issue in this project, as the ASV is not very time-sensitive.

The version of Python used in this project is Python 3.9.6, developed on a Microsoft Windows 10 computer. One exception is that some of the AI models are trained on an Ubuntu computer. The code is written using the Atom text editor.

5.1.4 Otter commands

The Otter has a list of messages that it can receive. These are all on NMEA 0183 format (mentioned in section 5.1.2). The messages include commands for setting the Otter to drift mode, autopilot mode, manual control mode, course mode, leg mode, and station mode.

Python methods are written for each of these messages. In these methods, the values are taken as arguments and sent to the Otter over TCP/IP. The most important message for this project will be manual mode.

For further details on the messages, readers are referred to the TN1000 - External Control Backseat Driver Interface_v1.2 documentation from Maritime Robotics.

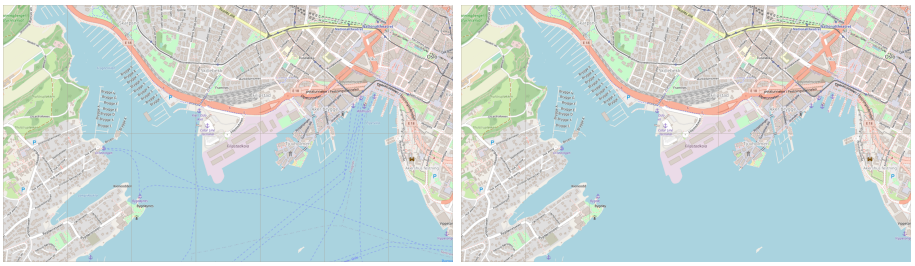


Figure 5.1: The Oslofjord map before and after "prettifying" it.

5.2 A* path planning

The A* implementation (the code) was shamelessly copy-pasted from Swift, 2017, found here: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2.

The function takes three inputs, which is a binary representation of a map, a starting point, and an end point. Ideally, one would have a binary map of the world where it is possible to input coordinates and calculate the route this way. But this was not available for this project with the required resolution. There does exist some (free) libraries that test whether a point on the earth is in the ocean or not, but the best resolution found for this was ca 1km. This isn't very helpful when maneuvering in tight spaces like harbors and fjords. So therefore, it was necessary to create one tailored for this project.

5.2.1 Simulation map

For testing the A* path planning algorithm, a binary map of the area around Filipstadkaia was created. This was done by taking a screenshot of a map of Filipstadkaia and its surrounding area. The map was found at https://map.openseamap.org.

After this, the map was cropped to the coordinates 59°54', 59°55', and 10°41', 10°44'. This was done by simply cropping at the grids shown on the map.

Then, the map was manually "prettified" to make the conversion easier. The color of the ocean on the map has a certain color, which is (170, 211, 223) in RGB representation. So what was done was filling in the parts of the map that were covered by other things like grid lines and numbers. See figure 5.1.

After this, an empty matrix was created with the same shape as the map, and it was filled with 1s and 0s depending on whether there was an obstacle (land) there or not (water/ocean).

When the map was generated, the geographical coordinates were scaled to points on the binary map, and translated back to coordinates when the route was found. This route can be used by sending one coordinate at the time to the Otter, as described in the next section.

5.3 Otter Station Mode path following

The method for path following has simply been to iterate over a list of the coordinates found during the path planning. The first coordinate is sent to the Otter, and when the destination is reached, the next coordinate is sent. This continues until the Otter reaches its destination.

The exception to this is if there's an obstacle is between the Otter and the next waypoint. If this happens, the Otter will perform a collision avoidance routine, and re-calculate the route to its destination.

A simple version of the Python code for this routine would look like this:

```
destination_reached = False
while destination_reached == False:
    for next_waypoint in the_otter.route:
        #Check for obstacles
        if not obstacle_found:
            the_otter.set_station_mode(next_waypoint)
            #Wait for Otter to arrive at waypoint
        else:
            #Drive around the obstacle
            the_otter.find_route(current_position, destination)
            continue

        #Check if this is the final destination
        if next_waypoint == route[-1]:
            destination_reached = True
            print("Final Waypoint reached")
            #Begin docking sequence
```

For the path following itself, the Otter's built-in "station mode" function is used. This command can be sent to the Otter over TCP/IP when set to "external mode". The message has the form:
$PMARSTA,<latitude>,<longitude>,<speed>*<checksum><CR><LF>

This means that at this stage it is simply assumed that the Otter will use the built in station mode to travel to each waypoint, one at a time.

5.4 Obstacle detection

For simulations, the YOLOv5 model was tested on a dataset of static images. This means that a dataset was loaded and tested to see how it performed on the images. The model was tested on a dataset containing images of ships, and some datasets that contained other objects than ships, to check for false positives.

The model was imported into a Python script and used to detect obstacles in a dataset known to contain images of ships, and on the datasets that contained other things than ships. Importing and using the model is as simple as this:

```
obstacle_detection_model = torch.hub.load("ultralytics/yolov5", "yolov5l")
image_frame = the_otter.get_image_frame()
result = obstacle_detection_model(image_frame)
print(result.pandas().xyxy[0])
```

The model was tested on the following dataset of ship images: https://www.kaggle.com/arpitjain007/game-of-deep-learning-ship-datasets

When checking for false positives, the model was tested on the following dataset of fish images: https://www.kaggle.com/crowww/a-large-scale-fish-dataset

Also, some images of pandas were included, found at: https://www.kaggle.com/holoong9291/pandaimagedataset

Then, some pictures of water bodies (taken from sattelites) were added: https://www.kaggle.com/franciscoescobar/satellite-images-of-water-bodies

Finally, images of dandellions and grass from this site were added: https://www.kaggle.com/coloradokb/dandelionimages

5.5 Collision avoidance

Initially, no specific method for collision avoidance was implemented and tested.

The main reason for this is because the path planning and path following algorithms were developed first, and then the auto docking methods. While working on the auto docking method, the docking scenario/environment was altered to represent the edges of the dock as obstacles. This was done because it was desirable to give the Otter some information on where not to drive, or what path to take.

Building on this, it was decided that it should be observed if the Otter could be taught to navigate through an environment with static obstacles using the reinforcement learning methods used in auto docking. In other words, the Otter was given the task of driving to a destination through an environment that contains obstacles, but it was up to the Otter's AI to learn how to avoid these obstacles. The goal therefore became to make a holistic solution, rather than implement a specific collision avoidance method.

5.6 Auto docking with deep Q learning

For the auto docking part of this project, a deep Q learning model was used. This was combined with a simulated environment that represented a "dummy dock". The model, which was made up of a neural network, was trained on the OsloMet AI lab's servers.

5.6.1 Simulation environment

A (deep) reinforcement learning agent needs an environment to act in. This is often the most challenging part of the process, because it is difficult to create an

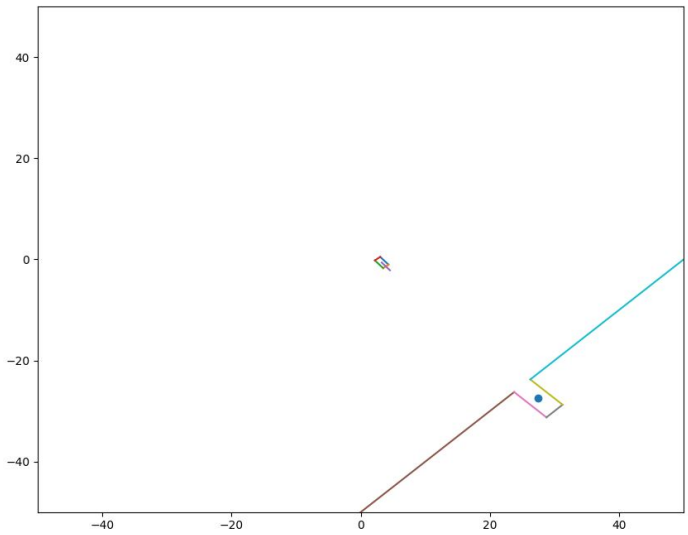


Figure 5.2: Visualisation of the Otter simulating a docking procedure

environment that accurately represents real world situations and environments.

Luckily, a professor at NTNU, Thor I. Fossen, has a Github-repository called PythonVehicleSimulator that is used to simulate a number of maritime vessels, including the Otter USV. The repository can be found at https://github.com/cybergalactic/PythonVehicleSimulator. This library was used to simulate the Otter's motion in the water for this project.

For the docking environment, a "dummy" dock is created. This is done by marking the dock as a point in the centre of the dock, and drawing lines around it to simulate the dock. See figure 5.2.

The dock is placed 27.5m east, and 27.5m south of the centre of the frame. The dock is represented as straight lines for simplicity.

5.6.2 Deep learning model

Pytorch was used to make a DQN (Deep Q Network) class, and an Agent class. The reason for choosing Pytorch over the other most popular machine learning library, Tensorflow, is because it is more widely used in scientific communities, and it is easier to prototype for simple tasks. A more detailed comparison between Tensorflow and Pytorch is given by Dancuk, 2021.

For writing the implementation itself, there are many good tutorials out there. One honorable mention is the Youtube-user "Machine Learning with Phil" who made a great tutorial called "Deep Q Learning is Simple with PyTorch | Full Tutorial 2020". His Github-repositories along with the code used to create the deep Q agent in this project can be found at: https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/DeepQLe

arning/simple_dqn_torch_2020.py

Hyperparameters

After some trial and error, the following hyperparameters were used:

- Gamma = 0.99
- Learning rate = 0.0001
- Batch size = 64
- Max memory size = 100,000
- Epsilon end = 0.01
- Epsilon descent = $1/(120 * N_{sim}/2)$
  - Here, N_{sim} is the number of simulations, which is 250,000

Observations, actions, and rewards

Following is a list of observations, possible actions, and rewards that the Otter can receive:

Observations:

- Surge
- Yaw rotational velocity
- Distance to dock
- Orientation relative to dock
- Which the direction the dock is

Possible actions:

- Move forward
- Stand still
- Move backwards
- Rotate counter-clockwise
- Rotate clockwise

Possible rewards:

- Colliding with the dock (negative)
- Distance to dock (positive)
- Orientation relative to dock (positive)

5.6.3 Putting it together

When both the agent and the environment is created, they can be put together to have the agent act in the environment. In short, this is done by running the simulation for a set amount of time (1 second), and then having the agent act according to its observation. The Otter will be simulated for 1 second using Fossen's library, and then the agent either determines the best action or takes a random action to explore. Based on this decision, the environment is simulated for another second, and the process is repeated until termination.

The simulation starts with the agent being placed at a random position and a random angle around the center of the environment. It then acts on the environment, based on exploration versus exploitation.

At the end of every simulation, the Otter (or agent) is given a reward based on how well it performed in that simulation. The reward is a combination of the angular difference and the distance between the dock and the Otter.

Mathematically, it is calculated as:

$$r = (d_{od} - d_{vd})/d_{od} + (\frac{\pi}{2} - |\omega_{vd}|)/(\frac{\pi}{2}) \tag{5.1}$$

Where d_{od} is the distance between the origin and the dock, d_{vd} is the distance between the vessel (Otter) and the dock, and $|\omega_{vd}|$ is the absolute angular difference between the vessel and the dock.

In the case of the Otter crashing, the reward is the same as the above, but with an added negative value corresponding to the amplitude of the total velocity of the vessel.

Mathematically:

$$r = -\sqrt{v_{su}^2 + v_{sw}^2 + v_h^2} \tag{5.2}$$

Here, v_{su} is the surge, v_{sw} is the sway, and v_h is the heave of the Otter.

The simulation ends when one of three things happen: The Otter drives out of bounds, the Otter crashes into the dock, or the time limit is reached.

Chapter 6

Simulation results

This chapter presents the results of the simulations. Most importantly whether they worked, or to what degree they were satisfactory.

6.1 A* path planning

An A* path from Tjuvholmen swimming pier to a dock at Filipstadkaia was generated based on a given set of coordinates of the starting point and the destination.

6.1.1 Verification

The degree of success of the method was based on whether the method was able to find a valid path to the destination. By valid path, it is meant that the path is coherent, leads to the destination, and does not go through impassable terrain (land).

6.1.2 Method evaluation

The method itself is very simple, which was the intention from the beginning. This provides a few advantages, where the biggest ones are ease of implementation, and computational cost.

The method provides a short path, which is good. This means that the vehicle will reach its destination using little fuel and time. There is one drawback to this however, and that is that the found destination can be very close to land or other obstacles. Because of this, it is necessary to include additional steps to avoid the vehicle from driving too close to impassable terrain.

The A* algorithm avoids obstacles. This does of course require that the map that is handed to the algorithm includes all the obstacles. But if given such a map, the route will be adapted to avoid these. The main benefit of this is that detected obstacles on the map can be marked, and then given to the A*

Figure 6.1: A binary representation of Filipstadkaia, where the A* algorithm has found a path to the entrance of the dock

algorithm to calculate a path to avoid it. This is further enabled by the next benefit of the algorithm, which is calculation time.

The method takes a short time to calculate, which is great for on-board computation. This means that recalculating a path is a viable option, and can be a very useful tool to have for collision avoidance. In practice, the Otter will have to deviate from its intended path to avoid an obstacle, but can quickly re-calculate the route once the obstacle is passed.

All in all the method can be considered a success at this stage. This is because it manages to do what it is supposed to do, which is finding a path that can be used to reach the destination, and it does so relatively quickly. The result can be seen in figure 6.1. The white dot at the beginning of the line is the Otter, and the line is the path generated by the A* algorithm.

6.2 Otter station mode path following

As the Otter's built in functions are being used, there hasn't been extensive testing of this functionality. The functions have been tested in three ways. These are using the Otter's own simulation program, the Otter out in the ocean, and a simple Python script that iterates through a list.

6.2.1 Verification

The verification of this method is based on whether the Otter is able to reach its next waypoint (current destination), the degree of accuracy, and how long it took. The Otter must also be able to set a new destination once its current one is reached, and adapt its path following to the next waypoint.

6.2.2 Method evaluation

The Otter comes delivered with a computer that has a specially designed application that is used to monitor and control it. In this application, the user can control a simulated Otter, and the physical Otter in the water. Both methods

have been tested, and it has been shown that the Otter will in fact follow this route with an acceptable degree of accuracy. As the Otter continuously reports its position, it is possible to simply check how close it is. A "radius of acceptance" can be set, which is the distance from the waypoint where the Otter determines that the waypoint is reached.

As for setting new destinations, a list of coordinates has been iterated through, and transmitted to a localhost server for testing.

In total, two methods (one for following paths and one for transmitting destinations) appear to be working, where both appear to work well in simulations. As with the other parts of this project, it will be necessary to test in a real physical environment to further verify the effectiveness of the methods.

6.3 Obstacle detection

The chosen obstacle detection model shows a relatively high degree of accuracy. The model is able to correctly detect a ship in 97.59% of images of ships, and detects a ship in 0.00015% of images of non-ship objects.

6.3.1 Verification

The chosen model (YOLOv5) was tested on a dataset containing 8932 images of ships, and 13204 images of objects other than ships. The degree of success was based on the number of true positives, and false positives. Additionally, the model is required to operate at a sufficient speed, but no specifics are given.

6.3.2 Method evaluation

Out of 8932 images of ships, the model detected a ship in 8717 of them. This gives an accuracy of 97.59%.

97.59% accuracy on ships can be considered acceptable. Of course, 100% is ideal, but this is an extremely challenging task, even for humans under certain conditions. It may help with larger images, as the images of the ships are quite small. The images in the testing datasets (of ships) have varying sizes, but the largest number of pixels in either a row or column was 210.

Out of 13204 images not containing ships, 2 of them were classified as containing ships. This gives a false positive rate of 0.00015%. These two images are shown in figure 6.2. A 0.00015% false positive rate is very low, and therefore acceptable. 0% is ideal, but it is extremely difficult to tune a model to get a 0% false positive rate without sacrificing much of the true positive rate.

Model size

In general, one should use a smaller model to make it work faster, or a larger model for more accuracy. It was decided to use the XL version of the network, because of the better accuracy on both true positives and false positives.

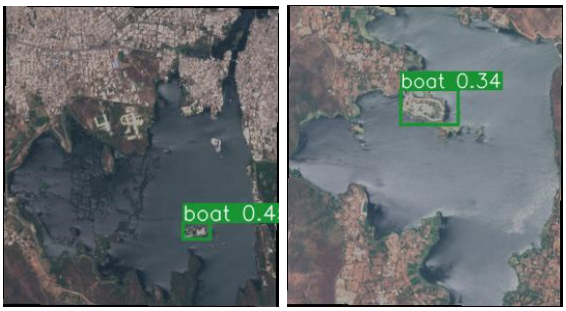


Figure 6.2: Not boats

The images of non-ships was scanned with the L (large) version of the model, and it took 375.01 seconds to scan the 13204 images. This gives a rate of 35.21 image scans per second. The L network gave a true positive rate of 94.57%, and a false positive rate of 0.00023%.

The XL version of the network was also tested, which took 453.98 seconds to scan the same images. This gives a rate of 29.08 image scans per second. The XL network gave a true positive rate of 97.59%, and a false positive rate of 0.00015%.

It should be mentioned that the scan was performed on a computer using an AMD Ryzen 9 5950X processor, and a Nvidia GeForce RTX 3070 graphics card. These components are relatively powerful, which means that processing is likely to be longer on edge computers like a Raspberry Pi.

The XL model is observed to be slower, but has better accuracy on both true positives and false positives.

Confidence threshold

One action that can be considered is to change the "confidence threshold". The confidence threshold is the number that must be exceeded for the model to classify something as (in this case) a ship. In figure 6.2, the confidence is shown to be 0.4, and 0.34. By increasing the confidence threshold one will get less false positives, but also get less true positives. The confidence threshold of the YOLOv5 pre-loaded model is set to 0.25.

The confidence threshold for the model was not changed for this project, as the model had a good balance between true positives and false positives.

6.4 Collision avoidance

As covered in section 5.5, no specific collision avoidance technique was implemented. The verification and evaluation of the collision avoidance has therefore been based on the auto docking scenario, where the Otter is penalized for colliding with the dock. The collision avoidance is therefore seen as a subset of

auto docking, and the degree of success for collision avoidance is correlated with the degree of success for auto docking. Because of this, collision avoidance will be evaluated as a part of the auto docking process, rather than an independent field.

6.5 Deep reinforcement learning auto docking

A deep q learning agent was successfully implemented in a simulated environment for the Otter. The results appeared promising, as the Otter was both turning towards and approaching the dock. When given 60 seconds to dock, the Otter did not reach the dock. When given more time, the Otter struggled when using a deep q learning method. In this section, the results of using the deep q learning model are discussed, and some techniques used to attempt to improve it is presented.

6.5.1 Verification

The method was verified by loading the trained agent's neural network model, and placing it in the centre of the simulation area (local coordinates [0, 0]), angled towards north. This is equivalent to a distance of $d_{vd} = 38.89m$ from the dock, with an angular difference of $\omega_{vd} = 135°$. The degree of success was determined based on how close the agent came to the dock in terms of angle and distance. Ideally, $d_{vd} = \omega_{vd} = 0$.

6.5.2 Method evaluation

While the auto docking phase can not be considered a success, some progress was made. The Otter does not dock by itself, but it was possible to see indications of it approaching. Unfortunately, the Otter does not show any sign of actually docking. This means that the chosen method does not appear to work, and something needs to be changed.

The same agent was tested on another environment (the CartPole-v0 environment from OpenAi Gym), and the agent performed well. By this it is meant that the agent was able to complete the challenge presented to it. But when tested in the docking environment, the agent was unable to dock when placed a certain distance away from it. This can mean that there is either something wrong with the docking simulation environment, or the chosen method is not appropriate for this specific task.

Reward function

The reward function was the part that was worked on the most. It was found that the best approach was to reward the agent at the end of every simulation, and then discount this reward back through the timesteps in the simulation.

Rewarding the agent directly at every timestep didn't work well. It was noticed that the agent did not want to rotate away from its orientation when it

was parallel to the dock, as this would give a reduced reward. This is because the difference in angle increased (negative reward), and the agent didn't get any closer to the dock (no reward).

What was done instead was to let the agent run the entire course, and then reward it. The reward was then "discounted" to all the previous steps by a chosen factor, called the gamma (0.99 in this case). This way, the agent avoided ending up in "deadlocks" where it would find a semi-random position out in the harbor, and just stay there, collecting rewards. The observation from this simulation experiment is that the agent appeared to end up in what can be compared to a local minima of the reward function.

Another reason for only rewarding at the end is safety. If an agent is rewarded more for reaching the dock quicker, one might risk that the agent will drive at full speed towards the dock, and then quickly brake when it arrives, to earn the most points. This is undesirable when making a vessel that is intended to transport humans.

Time

In the first round of testing the method, the Otter was given 60 seconds to dock. The Otter was turning towards the dock, and approaching it. However, there was simply not enough time for the Otter to reach the dock. This means that it approached the dock, but the simulation ended when the Otter was ca 5 meters from it.

Therefore, the Otter was given 120 seconds instead. But this led to the training simulation becoming more complex, and the agent (Otter) did not train as well. The Otter was not able to dock successfully when it was given double time. However, it must be noted that the number of simulations for training was reduced from 250,000 to 100,000, because of lack of time. For example, training 250,000 simulations for 60 seconds each took approximately 6 days.

Another disadvantage with giving the Otter this much time is that the discounted rewards become much smaller. For example, with a discount factor of 0.99, and a final reward of 2, the Otter gets the following initial rewards:

$2 * 0.99^{60} = 1.09431$

$2 * 0.99^{120} = 0.59876$

The initial reward is then halved. When the numbers become this small, it becomes increasingly difficult for the agent to separate bad actions from good, as the difference in feedback is not as great. This can lead to very long training time, and in the worst case not being able to find a solution at all.

6.5.3 Increasingly distant starting position

Since the Otter was struggling when approaching the dock, it was believed that it might be a good idea to train it on the final stage of the docking (the last few meters) first, and then expanding the training scenario to increasing distances and angles.

First, the Otter was placed 3.54m away from the dock, angled straight towards the dock centre. During this training scenario, the Otter managed to learn to drive straight forward for a bit, and then stop. The same results happened when it was placed 10.61m away from the dock, also starting with an angle straight towards the dock.

When placed at an even further distance (14.14m) and at a slight angle (22.5°), began struggling to dock again. Even after having trained the model on closer scenarios, the Otter was unable to dock. In other words, this didn't work.

6.5.4 Multiple agents

In an attempt to increase and diversify exploration, several agents with varying degrees of exploration/exploitation (epsilon) were introduced. Each agent was given a static epsilon of (0, 0.1, 0.25, 0.5, 0.75, 0.9, 1), and set out to explore/exploit. After collecting experiences, all the experiences were pooled in a common memory and trained on. The newly trained model was then distributed among all the agents.

While this was a very exciting thing to test, as it covers some interesting sociological phenomenons (for example, among the human population, some are more or less risk-taking, curious, and exploring than others), it did not lead to an improvement in the final performance of the Otter. The model improved more rapidly in the beginning, but then stagnated. The "alpha" agent, which always chose the exploiting option, showed early signs of approaching the dock, but no stable solution was found for approaching the dock and actually docking. The agent did not show any sign of taking the same route with minor improvements for each simulation episode, but rather it took a completely new path every time.

6.5.5 Teacher

Since both the observation space and action space is quite large (for being continuous), it was thought that it would be difficult for the agent to find an initial system for docking in the environment. Therefore, the concept of a "teacher" was introduced, which was named "determinist". The job of the determinist was to follow a simple set of rules that would make it approach the dock and attempt a simple docking. By doing this, the memory of the determinist would be saved together with the other agents, to "show the agent how it could be done". It was hoped that this would give the agent a good set of training data, which would both increase its performance and reduce training time. It didn't.

There is one benefit of having done this however. By implementing a simple docking routine using the same observations (features) as the Otter was given, it was shown that it is possible to dock the vehicle using these features, without very complicated calculations. This was interpreted as meaning that a simple

solution should exist, and the features that were created should be sufficient for an AI to learn a policy for auto docking.

6.6 Holistic evaluation

All in all, the initial methods were considered to be moderately successful. This is mainly based on the fact that methods have been found for most parts of the project, and the methods that do work seem to work very well.

As the method for auto docking didn't work as well as hoped, it will be necessary to find a new method to replace it. In the following chapter, some alternatives that were evaluated and tested are presented.

Chapter 7

Method re-evaluation

Because of the lack of progress in the auto docking section using deep reinforcement learning, it was necessary to re-evaluate this method. Testing the deep q learning algorithm in the docking environment gave some valuable experience, which can be used when re-evaluating this stage.

It is believed that the environment and simulation works. This can be assumed from the fact that a "determinist" was made that was able to both approach and dock at the dock. This should mean that not only does a solution using the selected features exist, but a simple solution exists. This is valuable information when choosing a new method for the auto docking challenge, but also when deciding on the topology of a neural network (if the new method involves one).

For now, it is believed that the method is wrong for this task. This is mostly based on the fact that the algorithm (deep q learning) works on other tasks. The docking simulation environment appears to work fine, and the deep q learning implementation used works fine, but the combination of the two does not seem to work.

In this chapter, the alternative methods that were considered and tested are presented.

7.1 Deep deterministic policy gradient

7.1.1 Introduction

Deep Deterministic Policy Gradient (DDPG) is an algorithm that concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy (OpenAI, 2018). DDPG falls under the sub-category of AI known as deep reinforcement learning methods, along with deep q learning and proximal policy optimization (elaborated below).

Deep reinforcement learning methods use (deep) neural networks to approximate the value of a state, or an action. In DDPG, a neural network is used to

learn an approximator for both the optimal action-value function $Q^*(s, a)$ and the optimal action function $a^*(s)$. The most important difference between deep q learning and DDPG is that DDPG is *adapted specifically for environments with continuous action spaces* (OpenAI, 2018). This can be quite valuable in the auto docking challenge, as the thruster configuration of the Otter is a continuous action space (-100% to 100% force). Interested readers are encouraged to learn more about the DDPG method and its implementation at OpenAI's Spinning Up documentation.

Like the deep q learning method, the implementation was largely based on the GitHub repository from the "Machine Learning with Phil" YouTube-channel. The code for the agent and its networks can be found at https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/DDPG/pytorch/lunar-lander. The rest of the setup (mainly the simulation environment) is the same as with the deep q learning method. The main difference is that the agent was given direct control of the thrusters, instead of having to set discrete values. Also, the reward function has been changed from the deep q learning, where the maximum score is now 100. The agent can receive a score of 50 depending on the distance to the dock, and another 50 based on its angle relative to the dock. The reward for relative angle is now only awarded if the agent comes within 1m of the dock.

7.1.2 Test results

The DDPG agent did not perform significantly better than the deep q learning agent. While the agent did have some simulations where it performed decently, these were rare and far apart. This means that the results were very unstable, and therefore unpredictable.

The agent's performance was monitored by reporting the final reward (the score). The test ran 10,000 simulations, and during this time the agent never got to an acceptable point. The maximum score the agent received during the simulations was ca 75. This means that it got within 1m of the dock, but did not end up in an angle that was very parallel to the dock. The results can be see in figure 7.1.

It may be possible that the agent will learn a pattern given enough time, but due to the time constraint of this project there was no time to test the method for extended periods. This can be backed up by OpenAI's blog post on Proximal Policy Optimization, where it is stated about deep reinforcement learning techniques that *They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks* (OpenAI, 2017).

Another theory is that the method seems to be stuck in a local minima, because it seems to never find an optimal solution. This can be a result of too steep training steps, which is counter-acted by PPO (below). It was therefore decided to test this method as well.

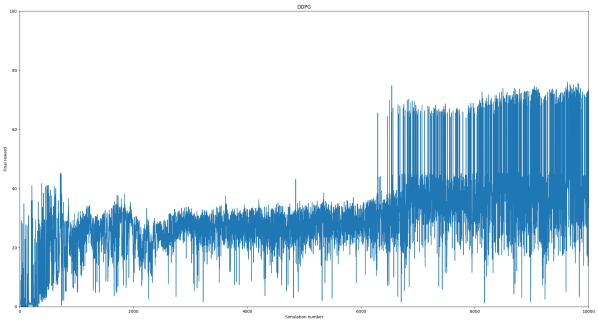


Figure 7.1: Final rewards from 10,000 simulations using a DDPG agent

7.2 Proximal policy optimization

7.2.1 Introduction

In 2017, OpenAI released a new class of deep reinforcement learning algorithms, called Proximal Policy Optimization (PPO). According to OpenAI, PPO *performs comparably or better than state-of-the-art approaches while being much simpler to implement and tune* (OpenAI, 2017). As with deep deterministic policy gradient, PPO also works in continuous action spaces.

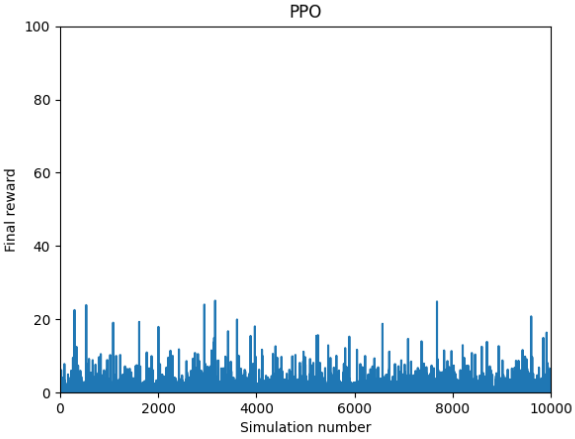
The most important contribution of the PPO method is the "clip" function used when training the network. The clip function constrains the loss/penalty to a certain interval, usually between 0.8 and 1.2. By doing this, the model updates less "harshly" by reducing the magnitude of the training steps.

Interested readers can learn more about the algorithm in the official OpenAI blog post.

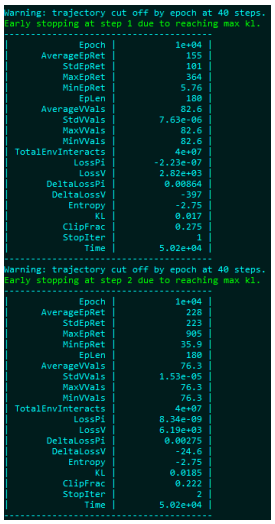
7.2.2 Test results

For the first simulation experiment with PPO, Phil Tabor's Github repository was used (again): https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/PPO/torch. The implementation was used and tested in the same manner as with the deep deterministic policy gradient agent. More specifically, the simulation environment was the same, the reward functions were the same, and also the general structure of the code was the same. The only practical difference was that the agent was now a PPO-agent, and not a DDPG-agent. Just like with the DDPG agent, 10,000 simulations were run, and the final reward (score) was recorded. The results can be seen in figure 7.2a.

It can be observed that the PPO agent actually performed way worse than the DDPG agent. It is not immediately obvious why this is the case, but one



(a) Phil Tabor's PPO implementation



(b) SpinningUp's PPO implementation

Figure 7.2: PPO simulation results

theory is that it is not the implementation itself, as the code has been successfully tested on another environment (OpenAI's CartPole-v0 gym environment).

One major difference is that this implementation outputs discrete actions, which we believe to be a disadvantage for the overall performance of the Otter. To allow for continuous actions, and to make sure that the method itself works, it was decided to test the original implementation straight from the source. The official Pytorch implementation from OpenAI was used.

For this algorithm to work, it was necessary to re-design the simulation environment to resemble the gym environments from OpenAI's Gym.

The standard hyperparameters for the ppo_pytorch class was used. 10,000 simulations were run (called epochs in this implementation), and the results were printed out as they went. The final outputs are shown in figure 7.2b.

The results show that this implementation of the PPO algorithm actually performed quite terrible as well. Note that the episode return is the cumulative sum of the rewards given at each timestep, not the reward given at the final step. So if the Otter stayed at its starting position for the entire episode, it was given a reward of $180 * 1.02320 = 184.176$. For comparison, if the Otter started at a perfect docking position and stayed there the entire episode, it would receive a return of $180 * 100 = 18,000$. In the final epoch, it is seen that the average return for each episode is 228. So as can be seen in this implementation as well, the agent barely moves from its starting point.

At this point, it can be considered a possibility that maybe (deep) reinforcement learning may not be suitable for this specific problem. When even the official implementation of a state-of-the-art method performs poorly, this can

be an indication that it is time to re-evaluate not only the specific method used, but also the whole category of methods to use.

7.3 NeuroEvolution of Augmenting Topologies

As (deep) reinforcement learning methods seem to work poorly for this task, it was decided to test another field of AI methods. This field is called Evolutionary Algorithms (EA), and is a programming paradigm where models and networks are allowed to evolve through techniques inspired by natural evolution. One such method is presented in this section, which is called NeuroEvolution of Augmenting Topologies, also known as NEAT.

7.3.1 Introduction

NeuroEvolution of Augmenting Topologies (NEAT), is an evolutionary algorithm that evolves not only the weights, but also the topology of a neural network. The NEAT algorithm starts with the simplest possible structure, where the inputs are directly connected to the outputs. The weights and topology are then evolved using techniques inspired by natural evolution. This process is called *complexification*, meaning to make more complex.

The algorithm begins with creating an initial population of individuals. The algorithm then loops through a process of selection and genetic operations to evolve this population until a termination criteria is met.

The NEAT algorithm has two major advantages. Firstly, it is often very quick to find a solution compared to other reinforcement learning methods (Stanley and Miikkulainen, 2002). This is of course a big advantage when one wants to do rapid prototyping and test out several different methods, bounded by a time constraint on a project. The other advantage is that it often finds a simple solution. This can be great because it means that the computation cost is likely to be very low, and will require very little memory. This in turn provides a benefit because it means that it can be implemented on low-cost edge computing devices like microcontrollers.

Initialization

The first thing that happens in an Evolutionary Algorithm (EA) is the initialization of a random population of individuals. In most AI methods, the way this is done varies, but one common way of initializing is to distribute the possible solutions over the entire search space. Another option is to "seed" the initial population so that it begins its search in an area that is likely to contain a solution.

In NEAT however, the initial population is a single "perceptron" based neural network, where the inputs are directly connected to the outputs, see figure 7.3. This network is then evolved and complexified through coming generations.

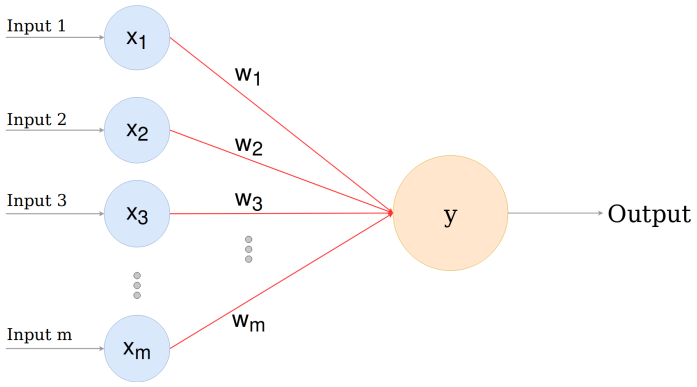


Figure 7.3: The perceptron, the most basic neural network configuration

Fitness and selection

After initialization, the individuals in the population are evaluated based on a fitness function, and given a fitness score. The fitness of an individual is how well it solves the task at hand, or how well it performs in a given environment. In the case of auto docking, this is how well they managed to dock the Otter in the simulation environment. Depending on the size of the population and the computational power available, one may choose to evaluate the entire population, or a random sample.

The fitness function is arguably the most important part of designing a working EA model. This is mainly because it directly affects the outcome and behavior of the individuals. One must therefore be very careful when choosing what to optimize for. A classic example is from a challenge where the goal was to win a game of racing boats. The individuals were rewarded for maintaining a high velocity, and the "fittest" individual simply drove around in circles, because it found that to be the best way of maintaining a high velocity over time.

After calculating the fitness scores, the best performing individuals are selected to breed a new generation. This process is known as "selection" (as in natural selection). The number of individuals that are selected is a hyperparameter.

Crossover and mutation

Crossover and mutation are known as "genetic operators". Genetic operators are used to change the structure of the individuals, hopefully in the right direction. Crossover is the combination of two "parents", that are selected from the pool of fittest individuals. This combination produces a "child" or an offspring, that is a resemblance of the two parents. These new children are then used to make up the next generation.

Depending on the type of algorithm used, one can make a new population of these children with our without mutations, or use a concept called elitism.

Elitism is a way of maintaining the best fitness of a generation by simply transferring the N number of best performing individuals unaltered into the next generation (where N is a hyperparameter, usually 1 or 2).

Mutation is the act of randomly altering an individual, for the purpose of diversity. The mutation rate (also a hyperparameter) determines the frequency of mutations. The mutation rate can somewhat imprecisely be related to the learning rate of a gradient descent based neural network. A too high mutation rate can result in a poorer final solution, but will explore faster.

There are two types of mutations that are specific to the NEAT algorithm. One is to insert a new node between two previously connected nodes, and the other is to create a new connection between two nodes. There are also other mutations available, like disabling certain connections, or altering a single or multiple weights.

To keep track of the mutations, and to measure compatibility, a historical marking is used to uniquely identify each gene. A unique number is given for each new mutation that occurs, which is the order it first appeared in the evolution. So the first unique mutation would be given number 1, the second 2, etc. For this reason, it is also called an "innovation number". These markings are used during crossover, where the genes are checked for activation, which means whether or not it is present. For example, gene number 1 can be the connection between node A and B, and gene number 2 can be the connection between node A and C. During crossover of two parents, the activation of these historical markings are used to determine the genotype of the child. An example of using innovation numbers in crossover can be seen in figure 7.4.

Speciation

A new mutation is often disadvantageous (Stanley and Miikkulainen, 2002). To "protect innovation" as the authors call it, new individuals are assigned a species (the one they are most similar to) and compete within this species. This gives the new individual time to adapt to its environment, and basically test out the fitness of a mutation.

The way this is done is by measuring the number of excess and disjoint genes between a pair of genomes (illustrated in figure 7.4), along with the average weight differences of their matching genes. This distance measure δ is checked against a compatibility threshold δ_t. If δ is less than the threshold δ_t, the genomes are placed into the same species. If a genome does not match any other known species, a new species is created.

7.3.2 Setup

When testing the NEAT algorithm, the same simulation environment that was used when testing the PPO algorithm from OpenAI was used. As for the NEAT algorithm there is a library called neat-python which was used. The documentation for the code can be found at https://neat-python.readthedocs.io/en/latest/.

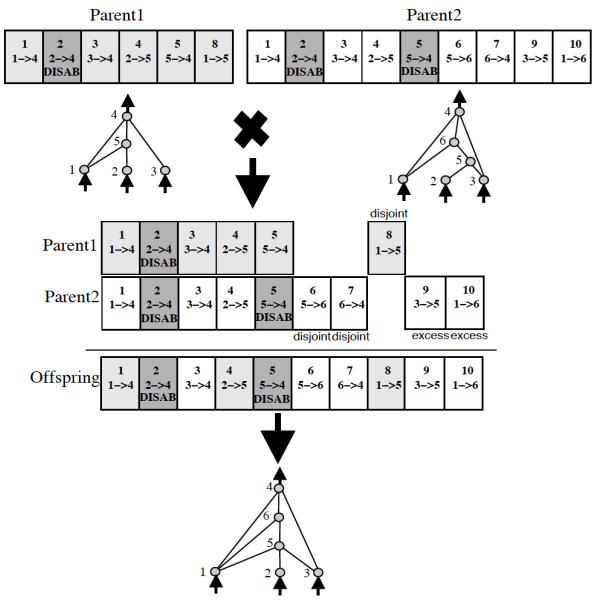


Figure 7.4: Example of using historical markings to represent a network during crossover

There is little need to tune the method before testing it, but there is a config file that must be included in the working directory that sets the parameters used in the method. The specific parameters used will not be covered here in this report, as the hyperparameters were frequently changed during all stages of this project. The config file used in the most successful run is found in the code repository belonging to this project. The only parameters that must be defined are the number of input and output nodes for a default genome. Initially, the population size was set to 500 and the activation function to ReLU (Rectified Linear Unit).

The reward was given only at the end of each simulation. There are two reasons for this. One: It was desirable to give the Otter freedom in exploring how to get to its destination. Two: The starting positions are random, which means that an individual that happened to start closer to the dock is more likely to get a higher fitness score.

The termination criteria was either upon complete extinction of all species, or if an individual managed to achieve an average score of 98 from running 5 simulations with random starting positions and orientations.

The inputs used for the EA model were:

1. Surge
2. Yaw rate

3. Distance from vehicle to dock
4. Angular difference between vehicle and dock
5. Which direction the dock is in
6. Distance to closest obstacle
7. Angle to closest obstacle

7.3.3 Simulation results

In the first scenario, the Otter was set to start at a set position and orientation in the middle of the local simulation area ($x = y = yaw = 0$). During this initial testing, the algorithm showed great promise by quickly finding its way towards the dock. After some tweaking with the hyperparameters, the Otter successfully managed to approach and dock at its destination, receiving a fitness score of over 95.

While this looked very promising, it was necessary to see if the algorithm could find more generalized solutions, and not just a solution to starting from a specific starting point and orientation. Therefore, in the next round of testing, the Otter started at a random position and orientation somewhere in the local docking area, where the x- and y-position is picked from a normal distribution with the mean $\mu = 0$ and standard deviation $\sigma = 10$ for the position, and $\mu = 0$ and $\sigma = \frac{\pi}{4}$ for the yaw. For every generation, each genome ran 5 times in the simulation environment.

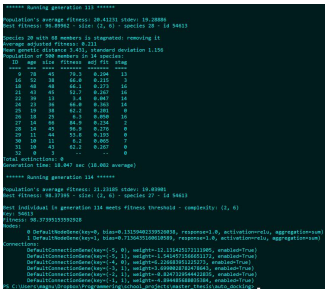
The NEAT algorithm appeared to find a solution to this auto docking scenario as well, also in a relatively short time. In the most successful run, it took the algorithm 114 generations to arrive at a solution that received a fitness score of over 98 (98.37395). The output of the test can be seen in figure 7.5a.

Another run was done where the initial population started with no connections. This simulation took 1046 generations to reach a solution, where it achieved a fitness of 98.48924. The output of this simulation can be seen in figure 7.6a

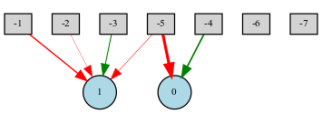
Discussion

First, it can be observed that the structure of the neural network is very simple. In the topology of the network from the first model, shown in figure 7.5b, there are no hidden nodes, as the input nodes (nodes -1 to -7 inclusive) are directly connected to the output nodes (nodes 1 and 0). This is beneficial as it makes it easier to analyze the inputs and how they affect the outputs. It also has the added benefit of being computationally cheap, which means it can be deployed on low cost computers like microcontrollers.

It can also be observed that the network structure does not include all possible connections. For example, it can be seen that nodes -1, -2, and -3 are only connected to node 1, which is the right thruster of the Otter. The same

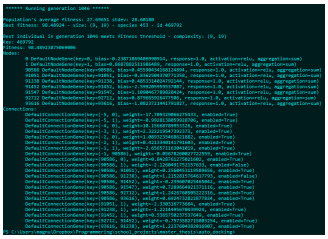


(a) Final output

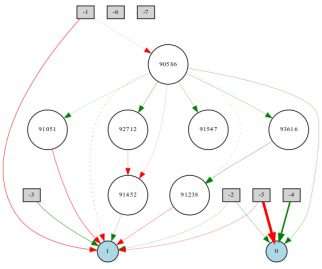


(b) Network structure

Figure 7.5: First NEAT simulation results



(a) Final output



(b) Network structure

Figure 7.6: Second NEAT simulation results

phenomenon is also observed in the network structure of the second model, figure 7.6b. Here, nodes -3 and -4 are only connected to one output node each. While this may lead to a working solution, as you can control the yaw of the Otter with only one thruster, this is somewhat inefficient and leads to the Otter driving in somewhat of a zig-zag pattern. This will in turn cause unnecessary fuel consumption, take longer time, and be uncomfortable for passengers.

While some of the inputs are only used for one output node, the most interesting part of the network topology is possibly the fact that the inputs related to obstacles are not used in either of the models. It was expected that an optimal solution would consider the obstacle in some way, but this model seems to completely ignore it. This makes sense in the sense that if you want to get to a waypoint, the only information you really need is which direction it is. The potential downside is that it may not be possible to use this model for collision avoidance, as discussed in section 5.5.

In sum, the Otter seems to arrive at its destination and at a good angle. The main drawback is that the Otter drives in somewhat of a zig-zag pattern, which is both inefficient and disturbing (for potential passengers). The next step is to see what happens if the NEAT algorithm is given a set of waypoints generated by the A* algorithm. Additionally, the inputs related to obstacles will be removed, and fuse the collision avoidance algorithm with path planning instead. The main reason is that the inputs aren't used in either of the NEAT generated networks, and therefore seem to be superfluous.

Chapter 8

Combining the methods

In this chapter, the procedures related to combining the methods will be presented.

First, the A* method is combined with auto docking, to give the Otter a path to follow to the destination. At this point, the Otter has a single model that is used for both path following and auto docking. It will be seen that it was better to split this into two models, where one is for path following and one is for auto docking.

After this, an overview of a method for adding obstacles is given, and how this can be used for collision avoidance. How this can be done will be discussed, but the system will not be implemented because of lacking components.

At this stage, the Otter has a system for planning a path using the A* algorithm, follow this path using a NEAT model, detect obstacles as it goes, and dock autonomously using another NEAT model.

8.1 Path planning and auto docking

The first fusion of methods was combining path planning with auto docking. In the initial simulations, the Otter was simply given the distance and angle to the position of the dock. To help it along the way, the Otter was given a path generated by the A* algorithm. Note that at this stage, path following and auto docking was treated as one holistic solution, where a common model was used for doing both tasks.

At this stage it was decided to change the simulation area (the map) to a binary map of the Aker Brygge area. There are two reasons for doing this. Firstly, this is the area where where the Otter will be live tested in the water, so it was more fitting to train it there. The other reason is that it was desirable to have a binary map that could be used for more realistic path planning. This means that the A* algorithm would have to find a path based on the environment with obstacles (mainly land and docks), and not just in a completely open world. The position of the dock (and its entrance) was found using a Google Maps

satellite image.

When testing in the new environment (the Filipstad harbor area), it was observed that the Otter had a difficult time doing two things with a single model. This means that it would either perform well during path following and not during auto docking, or have very little to no path following at all. Therefore, it was decided to split the model into two. A model for path following and one for auto docking were trained separately. This lead to both better performance and reduced training time, as two separate computers could be used for the task.

One important difference here is that the inputs related to obstacles were removed. The main reason for this is that collision avoidance will be implemented as a part of path planning, which will be elaborated further in the next section. Additionally, there are some practical implications. For example, what would the input be if no obstacles are detected? Or if there are obstacles on either side of the vessel?

It was hoped that adding an A* path would reduce the "zig-zag" driving that the Otter was doing. Implementing the A* path did not improve the shaky driving in itself, but a penalty for high yaw rates were added. Mathematically:

$$r = -0.1(e^{\frac{-|\omega|}{0.326}})^2 \tag{8.1}$$

Here, 0.326 is the maximum yaw rate of the vessel, and ω is the yaw rate. After training with this reward, the Otter began driving straighter.

8.2 Path planning and collision avoidance

After creating a new simulation area and adding a path generated by A*, the ability to add obstacles to the map was implemented. The binary map consists of zeros and ones, which indicate whether that point is available as a path. When an obstacle is detected, the location of it is marked as ones, along with all points around it in a set range (in a square or circle pattern).

Following, a new path is generated based on the A* algorithm. This is the main advantage of having a fast algorithm for calculating a path, as this makes it feasible to continuously update the path based on the dynamic environment. The Otter will then follow the new path generated, and (hopefully) avoid the obstacle. This has been tested in simulation, where an obstacle is added along the Otter's path after a certain amount of time, and the Otter is forced to calculate a new path around it.

The main drawback here is that there is no method implemented to accurately locate the obstacle. It is possible to tell which direction it is based on the image frame, but it is not possible to accurately know the distance, unless another camera or a lidar is added. One proposition was to use another camera to cross-reference the position of the obstacle based on the two images. But after some analysis it was found that this will give as many false positives as true negatives when the number of obstacles in the camera frames are greater than 1. An illustration of this is sees in figure 8.1.

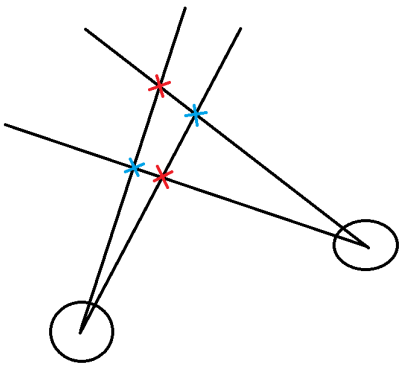


Figure 8.1: The cameras (represented as circles) can tell which direction an obstacle (blue crosses) is, and cross-reference to determine position

Chapter 9

Live testing

In this chapter the preparations, observations, and results related to the live tests of the Otter in the water are covered.

First, the preparations needed for live testing the boat is presented. Then, some observations that was made that may have had an effect on the performance of the vessel is covered. The performance of the vessel is then evaluated, and the results reflected upon.

9.1 Preparations

Before live testing in the water, some preparations needed to be made. In this section,the most important ones are covered.

Getting observations

One of the first things to do was to create a method to get the current state or observation from the Otter. One for path following, and one for auto docking. To do this, it was necessary to get the relevant sensor readings, and perform some calculations. The observations are then arranged in an array, so that they can be used as inputs to the neural network produced by the NEAT algorithm.

The way this was done was by reading the broadcast messages from the Otter, and saving this information. Then, a method was written that uses this information to create the needed observations. Finally, the observations are returned as an array.

Real life coordinates to local coordinates

Before testing live, a system for converting global GPS-coordinates to local coordinates in the testing area was needed. This is because the binary map of the Aker Brygge consists of a 3263x1858 grid. The lower left corner of the map has the coordinates $59°54'N, 10°41'E$, and the upper right corner of the map has the coordinates $59°55'N, 10°44.5'E$. 3263 is the distance in meters from the

south-west corner of the map to the south-east corner of the map. Likewise, 1858 is the distance in meters from the south-west corner and the north-west corner. This gives us a grid where each point has a size of 1x1 meters.

To convert the Otter's position from GPS coordinates to local coordinates, the distance from the south-west corner of the map to the Otter's position is calculated. This is easily done by the pymap3d python library. The same method is applied for finding the position and entrance of the dock. This method is also used for finding the distance between the Otter and the dock, in meters.

Converting thruster inputs to forces

The Otter command for allocating thruster force only accepts X-force and Z-torque, so it was necessary to convert the inputs to forces in the X-direction (along the boat from aft to bow), and torque around the Z axis (the yaw of the boat).

Calculating the force applied by the thrusters is given by

$$F_x = c(u_1 cos(\alpha) + u_2 cos(-\alpha)) \tag{9.1}$$

where α is the angle between the thruster and the center of gravity of the Otter, relative to a straight line running along the X-axis. Calculating the coefficient is done by setting both thrusters to 1 (100%), and seeing the result. This gives

$$c_x = \frac{1}{cos(\alpha) + cos(-\alpha)} \tag{9.2}$$

The same method is used for calculating the torque, except that one must calculate the force applied in a rotational motion, meaning that one can simply add $\frac{\pi}{2}$ to the angle. The same principle also goes for finding the maximum rotation on the thrusters, and calculating for z:

$$c_z = \frac{1}{cos(\beta) - cos(-\beta)} \tag{9.3}$$

where $\beta = \frac{\pi}{2} - \alpha$.

Setting a speed limit

For safety reasons, a coefficient of 0.4 was implemented for all actions the Otter performs. This effectively reduces the maximum speed, which reduces the consequences of a collision. Apart from the lower velocity itself, it is unclear how this will affect the performance, but it was deemed too undesirable to risk driving the Otter at full speed into something.

Defect velocity sensor

The sensor that reports the speed over ground was malfunctioning on the day of testing. There was a major drift in the sensor, that made the reported value increase infinitely.

To fix this, the GPS-position of the vessel was taken with 0.2s intervals, the distance between those two points were calculated, and divided by 0.2. This is not a great way to measure velocity as it is very inaccurate, and doesn't indicate which direction you are driving. However, it was the best estimate that was available at the time.

9.2 First test

9.2.1 Considerations

Wind

At the day of testing, yr.no reported winds of up to 8.8m/s. This is something very different from the simulations, which did not include any wind (or waves, which are generated by wind). The heading of the wind was west/south-west at our position, which means the Otter had headwind when approaching the dock.

9.2.2 Performance

Low speed

The first thing that was noticed was that the Otter drove much slower than it did in the simulations. There are two main reasons for this. First, a speed limit was added that made the Otter only use 40% force. Secondly, there was a very strong headwind, which made it drive even slower. The combination of these two factors made the Otter move very slowly during the path following stage.

When auto docking, the low speed combined with the strong headwind made the Otter not only stop, but drift backwards, away from the dock. As the Otter was attempting to stay in the same spot or approach slowly, the thruster force was close to 0 on both thrusters, which means the main force acting on the Otter is the headwind. The result was an unsuccessful attempt to enter the dock. However, there is another reason the Otter may have been unable to dock.

Inaccurate GPS

When comparing the GPS position of the Otter with its position in the water, it could be observed that the GPS was becoming increasingly inaccurate. The GPS position drifted over time, and at the later stages of the testing procedure the GPS reported the Otter to be ca 10m ahead of where it actually was. This was checked by displaying the Otter position (based on GPS) on a live map, and visually observing where the Otter was in the water at our testing area.

The consequence of this is that the Otter believed it was either very close to or in the dock, and therefore lowered the thruster force. Because of this, the Otter ended up a few meters away from the dock entrance, and turned off its thrusters. Combined with the headwind, the Otter was unable to reach the dock.

Figure 9.1: The path driven by the Otter is marked in green. A red supporting line is drawn to highlight the resemblance to the A* path found in figure 6.1. The Otter is symbolised as a bullseye

Two things are suspected to cause this. The first one is a defect sensor, and the other is interference. Maritime Robotics (the producer of the Otter) was contacted, and they responded that there has been some issues with the GPS drifting when the Otter is moving at low velocities. For this reason, it was assumed that GPS was faulty and the second test would be performed with improved firmware.

9.2.3 Reflection

Evaluation

The Otter showed signs of "intelligent behavior" as it drove towards the dock. This can be confirmed by looking at the path it followed when heading towards the dock. The path it took looked very similar to the path drawn by the A* algorithm, as pictured in figure 6.1. The path taken by the Otter can be seen in figure 9.1.

For this reason, the results of this test appear promising. While the behavior is far from optimal, a few things have been identified that can potentially improve the performance for the second test, outlined below.

Improvements

The first improvement to implement is increasing the "speed limit" on the Otter. It is suspected that the low velocity could make the Otter drive so slowly that it alters the behavior too much compared to the simulations, where it was allowed to use full force.

The second thing to improve is the GPS sensor. As the simulations are based on knowing the position of the Otter and the position relative to the dock, the algorithm is completely dependent on a functioning GPS.

As the IMU sensor is unable to report a velocity, it will be very desirable to have this fixed as the surge of the Otter is one of the inputs for the NEAT models.

The final thing to do is to test the Otter on a day with less wind, to see what effect this has on its performance.

9.3 Second test

9.3.1 Considerations

The first thing to be observed was that it was much less windy. According to yr.no, the wind was approximately 3m/s. This is good as it more closely resembles the simulation environment.

This time the speed limit coefficient was changed to 0.6. While this may have been somewhat of an overkill because of the calmer wind, this decision was made based on the argumentation that it would be better for the Otter to be able to act closer to how it was allowed in the simulation. Ideally the Otter should have been given 100% force, but this was not done for safety reasons.

The GPS was reported to be fixed by Maritime Robotics. This meant that it should be possible to rely on the GPS for this test.

The IMU was still not fixed, which meant that it was not possible to get a reading on the velocity. Therefore, the GPS position read with 0.2s interval was still the method used for determining the velocity of the vessel.

9.3.2 Performance

For this round of testing, the Otter showed many of the same symptoms as it did in the first test. It follows something that looks similar to the path generated by the A* algorithm, and starts approaching the dock. The issue still remains though, where it slows down as it approaches the dock but never quite reaches the entrance. This time information on which model it was currently using was reported, and it was observed that the Otter never switched from the path following model to the auto docking model. This means that the auto docking procedure never began.

Another thing that was observed was that the Otter started "circling" around the entrance of the dock. This can be seen in figure 9.2. This makes us suspect that we have a poor protocol for switching from path following to auto docking. Our current theory is that the final waypoint of the path generated (to the entrance of the dock) lies too far outside of the area where the Otter is supposed to begin the auto docking routine.

As with the first test, the GPS seemed to be drifting during this test as well. A screenshot from the Vehicle Control Station is seen in figure 9.3. Both from the camera frame and visual observations from land, it can be seen that the Otter lies very close to the centre of the water, while the GPS reports the Otter

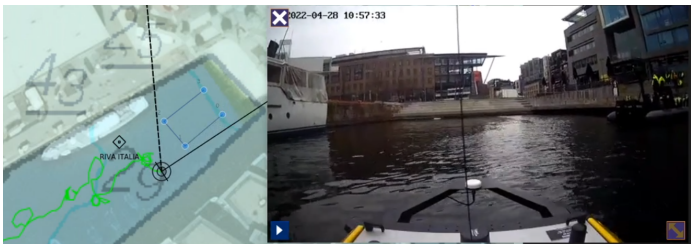


Figure 9.2: Left: The GPS position reported from the Otter. Right: An image frame taken from the Otter camera at the same time.

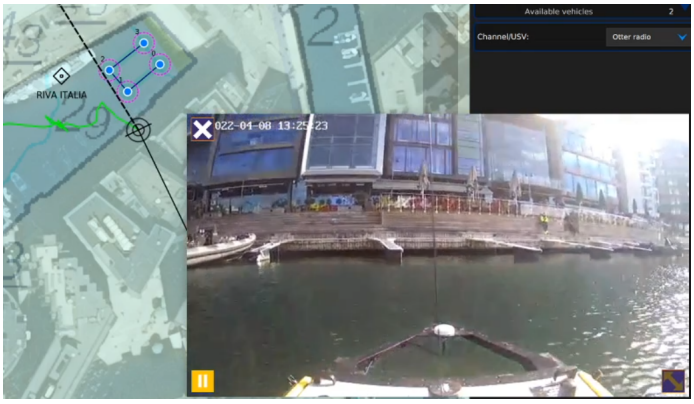


Figure 9.3: Left: The GPS position reported from the Otter. Right: An image frame taken from the Otter camera at the same time.

being on land. The image frame in figure 9.3 is taken at a later stage than the second NEAT test.

9.3.3 Reflection

The initial thought is that since the GPS was still malfunctioning, and the vessel is completely dependent on it for the NEAT model, then this could be the main cause of the Otter under-performing. There can of course be more reasons to why this is, but this is something that should be tested and explored further. The biggest question is probably what the source of the disturbance is. As the Otter operates in a densely populated and urban area, there is a real chance of interference. However, interference usually causes a bias or a high variation in the measurements, while in this particular case there is a constant drift, which is always in the same direction (both in the first and second test the GPS drifted south-east over time). Additionally, the producer of the vehicle has admitted to having issues with the GPS sensor at low velocities, which gives further reason to suspect that the sensor may be faulty.

In addition to the GPS, the IMU sensor was also malfunctioning during the

second test. It is difficult to tell the direct consequence of having an inaccurate reading on the velocity, especially because is it very difficult to tell exactly how fast the Otter is really traveling. In addition to the method of determining the velocity itself is not very good, when combined with a poor GPS reading it becomes even worse. This means that it will be necessary to test the Otter again with a different IMU sensor to check if this is a contributor to the poorer performance.

Apart from the sensors, there is of course the possibility that the model is simply not good enough. The NEAT algorithm is great for solving simple problems, and it seemed to perform quite well in the simulation environments. But the case may be that the simulation environment it trained in is a very simplified version of the real world, meaning that the NEAT algorithm is too simple for this kind of task in the real applications. On the other hand, the method itself may be a good fit, but the model used in these tests is simply not trained well enough. This could be because of a sub-par simulation environment, our bad hyperparameters, or simply not enough training time. One thing that can be said with a high degree of certainty is that if the model is not good enough, it is the path following model, as this is the only one the Otter used during the live test.

In relation to re-training models, one could include a speed limit during training. The Otter may rely on being able to control the thrusters completely, but in reality this was not the case. It is believed that the training environment should resemble the real world as closely as possible. This does not only include physics, but also rules and other non-physical considerations like speed limits.

Finally, there is the possibility that the dynamic environment the Otter is tested in is simply too different from the simulation environment. The two biggest factors here are waves and wind. As mentioned earlier, the simulation environment does not include wind or waves, but this is usually present to some degree in the real world. It is difficult to say something about exactly how this affects the Otter's decision making.

Chapter 10

Discussion

In this section, some essential questions related to the project are discussed. An attempt is made to give an evaluation of the project as a whole, and the learning outcomes are reflected upon, along with some suggestions to what can be improved.

10.1 Project scope

This project set out with the goal of creating an Autonomous Surface Vehicle that could be used to transport humans and goods. The physical vessel used is not fit to transport neither humans nor goods, but its behavior is designed to resemble that of a vehicle used with this intention. This can for example be either a passenger ferry or a shipping container carrying fragile goods.

The scope of this project started out quite grand, but this was intentional. The goal was to get as far as possible in making an ASV, which included all the five key areas. This includes planning a path to a destination, following that path, detecting obstacles along the way, avoiding collisions, and docking at the destination. For this reason, the methods used have been somewhat basic and simple, which was also intentional. It is assumed to be too ambitious for a single person to make a full scale completely autonomous passenger ferry in the time frame given for this project, but the ASV developed in this project can be used as a platform for further development and research. In this sense, it is the impression of the author that the scope of the project has been just right considering the approach of starting with a very minimalist approach, and adding on the parts that were needed as the project went on.

10.2 Learning outcomes

Reward functions are intricate

The reward functions given to the agent during training received a lot of attention. It can be argued that this is the most important part of training an AI model, specifically in EA algorithms. This is because the reward directly influences the way the agent "thinks" and learns during its training. For the NEAT algorithm specifically, it is almost the only thing that needs to be specifically constructed by the developer.

It was noticed that the rewards require a high degree of specification, as they are very intricate. This complicates the development process, and requires more time than anticipated. For example, when the Otter was given a reward based on how close it was to the dock at a certain time, it rushed to approach the dock quickly. Not only can this be dangerous and uncomfortable for the passengers, but it ended up crashing into the dock a lot. Therefore, the Otter was given a negative reward for crashing, but this only led to the Otter not daring to even approach the dock. The amplitude of the negative crashing reward therefore needed to be adjusted.

Another issue with testing different reward functions is that it can take a long time to see the effect. If there is an error in the simulation environment, it can often be discovered quickly. But several generations need to pass in the training to be able to observe the learned behavior of the AI agent based on its reward function.

A good environment is important, and takes the most time

While there are readily available libraries and implementations for most AI methods, the simulation environment used to train the model needs to be created for each specific case. There was no simulation software available for the specific purpose of this project, so it was necessary to create it from scratch. This took more time than anticipated.

The main reason this took a long time to develop is because it was necessary to create a universal solution that could be used with many different types of AI methods. This means that re-usability is of the essence, as it was the intention to design the environment in such a way that it could be used by others to test out their methods at a later stage. This means that the environment needs to be flexible in terms of customization.

It was experienced many times that when something didn't work, it was a bug in the simulation environment that caused it. This could be anything from poor reward functions to a calculation error for the position/angle of the dock. This made it challenging because it was often difficult to tell whether the error lied with the AI model or the simulation environment.

Training AI models takes time

Training AI models took a lot longer than anticipated. As mentioned earlier, training a DRL method for auto docking took around 6 days, and would probably have benefited from being trained even longer. One reason why the model takes a long time to train is because it has a temporal simulation. This means that timestep 4 needs to be simulated before timestep 5, etc. When training on a static dataset it is possible to train on every data in parallel, but this is not possible when one data point is dependent on the previous. It needs to be generated serially.

This is an issue when one wants to test different methods. Even with a perfect simulation environment, it could take days or even weeks to determine whether a method seems promising or not. Because of the limited computational resources available in this project as well, it is too time consuming to train several models of different methods simultaneously.

Sensors are unreliable

In a simulation environment, it is possible to get perfect readings from sensors. In the real world, not so much. This means that while a really well trained model can perform very well in simulations, the performance can drop severely when deployed in the real world. It is very difficult to address this in the simulations, but one option is to always add a bit of distortion to the measurements. For example, adding a random offset with a small standard deviation for each input.

Chapter 11

Future work

In this chapter, some suggestions for future work are presented. The chapter is divided into two parts, namely "short term" and "long term". The short term suggestions are things that can be implemented for the Otter USV itself, while the long term suggestions are aimed at using an ASV for transportation as a whole.

Note that this is merely a short list of suggestions among the vast ocean of opportunities related to Autonomous Surface Vessels. The chapter covers general suggestions and ones specific to transportation of humans. This means that suggestions related to other professions like fishing and petroleum services will not be covered.

11.1 Short term

This section covers a list of suggestions for future work related to improving the performance of the Otter USV.

- Make a method for easily creating maps for different docks
- Test at multiple docks
- Accurately determine position of obstacles
- Avoid dynamic obstacles
- Include weather in simulation training
- Determine whether it is safe to travel a path based on weather

Make a method for easily creating maps for different docks

As of now, only two maps are made for training the Otter. One "dummy dock" only used for testing, and one that represents the Aker Brygge area. This is fine if one wants to make a local ferry that travels from Filipstadkaia to Aker

Brygge, or to Bygdøynes for example. But to be able to deploy the Otter at an arbitrary area, it will be necessary to find a simple and efficient way to generate a map for that area.

Alternatively, one could have one large map of Norway for example, but this will be very memory heavy, even with a binary representation. A variant of this is to make a map of Norway and then crop out the part that is needed. So if you would like to deploy and test the Otter in the Trondheim bay, you would crop out this area and feed that local map to the Otter.

Test at multiple docks

It is essential that the Otter is able to dock at a number of different docks to be able to work as a ferry or water-taxi. Therefore, it needs to add new docks to the repertoire. In theory, one should be able to just update the "dock position" and "dock entrance" info for the Otter, and it should dock at that destination. But for safety and temporal reasons, the Otter in this project was only tested on a single dock.

In future, it will be necessary to test its ability to take in an arbitrary position of a dock, and dock there. This will be a huge benefit if used as a water-taxi, as it may allow the vessel to use private and personal docks as pick-up and drop-off points.

Accurately determine position of obstacles

While the Otter at this stage is able to detect ships, it lacks the ability to (accurately) determine their positions. This is absolutely essential for making an ASV that will operate in areas with other ships, like in harbors. There are several methods for accomplishing this, where two honorable mentions are using a lidar or a drone assistant.

Avoid dynamic obstacles

At this stage of the project, the Otter can add an obstacle to a map and plan a route around it. This works for static obstacles, but is inefficient for moving obstacles. It would therefore be great to see implemented methods for both estimating the movement of an obstacle and methods for avoiding it.

Include weather in simulation training

As seen from the live tests, wind and current can have a major impact on the vessel. Therefore, it can be very beneficial to include these situations when training our models. For this reason it is believe that it would be beneficial for the Otter to train in a simulation environment that includes wind and currents of varying strength and direction. It is expected that this can make the Otter more robust by becoming familiar with more complex situations.

Determine whether it is safe to travel a path based on weather

As mentioned, the ASV should be able to adapt to different weather conditions such as wind and current. But in addition to this, it would be a neat feature for the vessel to be able to decide whether the trip is too risky or not. This isn't restricted to disasters like tsunamis, but also if there are waves large enough to risk injury for passengers or the goods for example. This can be done by planning a path to a destination, and then checking the weather forecast of major nodes along the way at certain intervals (for example every hour), and decide if the weather conditions are acceptable.

11.2 Long term

In this section, suggestions for future work that should be implemented in an autonomous transport fleet are presented. These are ideas that can be done as standalone projects, or as part of a bigger project where one has several vessels available that perform at an acceptable level individually.

- Autonomous charging
- Automatic loading and unloading of goods
- Using swarm technology to optimize many vessels
- Add a safety feature for the passengers in case they fall over board

Autonomous charging

For a vessel to be able to operate autonomously over time, it would need to be charged regularly (unless you're using nuclear power, but that is another topic entirely). The most obvious place to do this is when the vessel is docked. We therefore suggest developing a mechanism for automatically charging a vessel while it is docked. This could be done either cabled, which will provide the fastest charging, or by using induction, which is somewhat slower but is easier to implement. In addition to this, it could be quite useful to place solar panels on the roof of the vessel, to further increase charging potential.

Automatic loading and unloading of goods

For vessels designed to transport goods, an important step in reaching full autonomy is the automatic loading and unloading of goods. There are two main components to this. First, the goods need to be organised in a way that the ones that are next to be unloaded are accessible (they are not placed at the bottom for example). Secondly, the correct goods needs to be physically moved onto a safe spot at the harbor.

One suggestion for this is that the transport vessel can be designed in such a way that it has two robot arms that will load and unload items when docked.

If the ship is docked with its side towards the dock, one robot arm could be mounted on each side of the vessel (one forward and one astern), with a container room between them. One of the arms could be responsible for loading goods onto the vessel at one side, and the other will unload from the other side. The container space in the middle could include a conveyor belt that would move the goods from one side to the other.

This is a quite trivial example as logistics are often much more complex than this, since items come in all shapes and sizes and have different destinations. Also, depending on the goods that are handled, this also requires some important calculations regarding weight distribution and torque. This makes it a somewhat cross-disciplinary topic, including both electronic, robotic, and mechanical engineering.

Using swarm technology to optimize many vessels

In the future, it will be beneficial to have several vessels working together. It will therefore be necessary to have an optimization algorithm that can coordinate the actions of these vessels. There are some very interesting swarm optimization algorithms, also known as multi-agent systems, that may prove to be useful in this situation.

Add a safety feature for the passengers in case they fall over board

Obviously, the safety of the passengers is the top priority. It is therefore necessary to have a good routine for what happens if someone falls overboard. The most basic equipment along with personal flotation devices, are lifebuoys. These should be present and easily accessible on every vessel that transports people.

But when it comes to autonomy, it is very difficult to detect when a person goes overboard with 100% confidence. There should therefore be an option for the passengers to press an emergency button that stops the vessel and prepares for rescue. One additional option is to include personal flotation devices with built-in beacons that will activate automatically if it either gets wet or is at a certain distance away from the vessel. An expansion to this is that when an emergency alarm goes off, a life buoy could be deployed automatically to where the person is detected in the water.

Chapter 12

Summary

In this report, the project of creating an Autonomous Surface Vehicle that can be used to transport humans and goods using Artificial Intelligence methods is presented.

The report begins with the motivation behind the project, and its objective. Following, the reader is given an introduction to ASVs, along with its basic components and functionality. Theoretical background on the five key areas that make up this project are then provided, along with some examples of different methods that are used in these areas. The five key areas are path planning, path following, obstacle detection, collision avoidance, and auto docking. After the theoretical introduction, the timeline for the project is presented.

Initial methods are chosen for each of the five key areas, along with the reasoning for choosing them. Afterwards, the setup along with the relevant technologies needed to implement the chosen methods are presented. After testing the initial methods, the results of the simulations done with them are presented and discussed. Then, the methods are re-evaluated and replaced where needed. Finally, the methods are combined to attempt a holistic solution.

The final stage was to test the Otter live in the water. The necessary preparations were made, and some important considerations on the day of testing were reflected upon. The performance of the Otter is then reviewed, and some suggestions for improvement are reflected upon.

The final part of this report contains a discussion of the project, and a section with suggestions for future work. The report ends with this summary.

I started this project with the hope of creating an ASV that could navigate autonomously, which in turn could be used for transporting people and goods. Unfortunately, I was not able to test the models on maritime vessels that are able to carry humans. However, I do believe I did manage to create an ASV that appears to behave in a manner that can remind of intelligent behavior. The ASV planned a path, followed it, detected ships along the way, and attempted to dock at the destination. The ASV did not perform all its tasks optimally, but this is not to be expected from a project with such relatively limited time and resources. Still, I do believe that my work here can provide a great starting

point for the further development of maritime vessels that one day can provide transportation services for both humans and goods.

Finally, I would like to thank you again for taking the time to read my report on this project (unless you just skipped to the end). It has been a great pleasure to do this project, and I believe with my whole heart that we some day soon will be able to take autonomous ferries across the ocean.

Bibliography

Asvadi, A., Premebida, C., Peixoto, P., & Nunes, U. (2016). *3d lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes*. https://www.sciencedirect.com/science/article/pii/S0921889016300483

BBC. (2018). *The ferry using rolls-royce technology that sails itself*. https://www.bbc.com/news/av/technology-46350188

Bevilacqua, F. (2013). *Understanding steering behaviors: Path following*. https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-path-following--gamedev-8769

Bitar, G., Martinsen, A. B., Lekkas, A. M., & Breivik, M. (2020). *Trajectory planning and control for automatic docking of asvs with full-scale experiments*. https://arxiv.org/abs/2004.07793

Council, N. R. (2005). Autonomous vehicles in support of naval operations. National Academies Press.

Dancuk, M. (2021). *Pytorch vs tensorflow: In-depth comparison*. https://phoenixnap.com/blog/pytorch-vs-tensorflow

Dunbabin, M., Grinham, A., & Udy, J. (2009). *An autonomous surface vehicle for water quality monitoring*. https://www.araa.asn.au/acra/acra2009/papers/pap155s1.pdf

Falson, A. (2016). *Tesla autopilot vulnerable to hacking?* https://performancedrive.com.au/tesla-autopilot-vulnerable-hacking-1113/

Faltinsen, O. M. (2005).

Fossen, T. I. (2011a). *Handbook of marine craft hydrodynamics and motion control*.

Fossen, T. I. (2011b). Handbook of marine craft hydrodynamics and motion control.

Fossen, T. I. (2011c). Handbook of marine craft hydrodynamics and motion control.

Fossen, T. I., Breivik, M., & Skjetne, R. (2003). *Line-of-sight path following of underactuated marine craft*.

Guo, S., Zhang, X., Zheng, Y., & Du, Y. (2020). *An autonomous path planning model for unmanned ships based on deep reinforcement learning*. https://www.mdpi.com/1424-8220/20/2/426/htm

Huang, B.-Q., Cao, G.-Y., & Guo, M. (2005). *Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance.* https://ieeexplore.ieee.org/abstract/document/1526924?casa_token=JBXgsVHDm-8AAAAA:SoXs_yfLlxa8BFjh8-K4bcnjPOYZMK41o_Y69xZnV3IfYDHlyHn_F3SJ_lUm34jy3kUk-Fg9Pw

Kuwata, Y., Wolf, M. T., Zarzhitsky, D., & Huntsberger, T. L. (2013). *Safe maritime autonomous navigation with colregs, using velocity obstacles.* https://ieeexplore.ieee.org/document/6519944

Lidar, V. (2021). *A guide to lidar wavelengths for autonomous vehicles and driver assistance.* https://velodynelidar.com/blog/guide-to-lidar-wavelengths/

Manley, J. E. (2008). *Unmanned surface vehicles, 15 years of deployment.* https://ieeexplore.ieee.org/abstract/document/5152052?casa_token=9S6k-ykVWsEAAAAA:Gr-3_oAn2gY73QgRJRohJo2Tjv0FhrasLO_vQHQP_ldyurZ6yheCE9_YbrBoc9YDYKZwiEGqh9o

Martinsen, A. B., Lekkas, A. M., & SebastienGros. (2019). *Autonomous docking using direct optimal control.* https://www.sciencedirect.com/science/article/pii/S2405896319321755

Matthies, L. (2014). Obstacle detection. In K. Ikeuchi (Ed.), *Computer vision: A reference guide* (pp. 543–549). Springer US. https://doi.org/10.1007/978-0-387-31439-6_52

Meyer, E., Robinson, H., Rasheed, A., & San, O. (2020). *Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning.* https://ieeexplore.ieee.org/abstract/document/9016254

Mirriam-Webster. (n.d.). *Waypoint definition.* https://www.merriam-webster.com/dictionary/waypoint

Moore, S. K. (2017). *Superaccurate gps chips coming to smartphones in 2018.* https://spectrum.ieee.org/tech-talk/semiconductors/design/superaccurate-gps-chips-coming-to-smartphones-in-2018

Navone, E. C. (2020). *Dijkstra's shortest path algorithm - a detailed and visual introduction.* https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

Nikel, D. (2017). *Autonomous boat tests in trondheim.* https://www.lifeinnorway.net/unmanned-boats/

OpenAI. (2017). *Proximal policy optimization.*

OpenAI. (2018). *Deep deterministic policy gradient.*

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2015). *You only look once: Unified, real-time object detection.* https://arxiv.org/pdf/1506.02640v5.pdf

Robots, S. (2019). *Motion planning algorithms - a* vs dijkstra's.* https://www.youtube.com/watch?v=RYdBcnSiwag

Schuster, M., Blaich, M., & Reuter, J. (2014). *Collision avoidance for vessels using a low-cost radar sensor.* https://www.sciencedirect.com/science/article/pii/S1474667016431447?via=ihub

StackOverflow. (2020). *2020 developer survey*. https://insights.stackoverflow.com/survey/2020/#technology-most-loved-dreaded-and-wanted-languages-loved

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127. https://doi.org/10.1162/106365602320169811

Swift, N. (2017). *Easy a* (star) pathfinding*. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

Ulrich, I., & Nourbakhsh, I. (2000). *Appearance-based obstacle detection with monocular color vision*. https://www.aaai.org/Papers/AAAI/2000/AAAI00-133.pdf

Wang, H., Wei, Z., Wang, S., Ow, C. S., Ho, K. T., & Feng, B. (2011). A vision-based obstacle detection system for unmanned surface vehicle. *2011 IEEE 5th International Conference on Robotics, Automation and Mechatronics (RAM)*, 364–369. https://doi.org/10.1109/RAMECH.2011.6070512

Wang, H., Yu, Y., & Yuan, Q. (2011). *Application of dijkstra algorithm in robot path-planning*. https://ieeexplore.ieee.org/document/5987118

Wevolver. (n.d.). *Photo of the otter usv*. https://www.wevolver.com/wevolver.staff/otter/

Wikipedia. (n.d.-a). https://en.wikipedia.org/wiki/Euler_angles

Wikipedia. (n.d.-b). *A* search algorithm*. https://en.wikipedia.org/wiki/A*_search_algorithm

Wikipedia. (n.d.-c). *Motion planning*. https://en.wikipedia.org/wiki/Motion_planning

Wikipedia. (n.d.-d). *Propeller and rudder*. https://upload.wikimedia.org/wikipedia/commons/d/db/Brosen_propelersterntychy.jpg

Wikipedia. (2018). *Frognerkilen*. https://no.wikipedia.org/wiki/Frognerkilen

Wooa, J., & Kim, N. (2020). *Collision avoidance for an unmanned surface vehicle using deep reinforcement learning*. https://www.sciencedirect.com/science/article/pii/S0029801820300792

Yan, R.-j., Pang, S., Sun, H.-b., & Pang, Y.-j. (2010). *Development and missions of unmanned surface vehicle*. https://link.springer.com/article/10.1007/s11804-010-1033-2

Yara. (2018). *The first ever zero emission, autonomous ship*. https://www.yara.com/knowledge-grows/game-changer-for-the-environment/