Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of ICT and Natural Sciences

Arild Valderhaug
Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy
Ola Sæterøy

Autonomous Ships: A Velocity Obstacle Particle Swarm Optimization Hybrid Algorithm

Bachelor's project in Bachelor of Science in Engineering - Automation
Supervisor: Anete Vagale, Aleksander L. Skrede
Co-supervisor: Robin T. Bye, Ottar L. Osen
May 2021

NTNU
Norwegian University of
Science and Technology

Arild Valderhaug
Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy
Ola Sæterøy

Autonomous Ships: A Velocity Obstacle Particle Swarm Optimization Hybrid Algorithm

Bachelor’s project in Bachelor of Science in Engineering - Automation
Supervisor: Anete Vagale, Aleksander L. Skrede
Co-supervisor: Robin T. Bye, Ottar L. Osen
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of ICT and Natural Sciences

NTNU
Norwegian University of Science and Technology

Tittel:

Autonomous Ships: A Velocity Obstacle Particle Swarm Optimization Hybrid Algorithm

Candidate name (number):
Arild Valderhaug (10036)
Karl-Oskar Norheim Molvær (10069)
Magnus Heggdal Sandøy (10097)
Ola Sæterøy (10008)

Date: 19/05/2021	Sub. Code: IE303612	Subject Name: Bachelor Thesis	Document Access:

Study Program: Automation Engineering	Nr. pages/Attachments: 103 / 11	Bibl. nr:

Advisers:
Anete Vagale
Aleksander Skrede
Robin T. Bye
Ottar L. Osen

Summary:

The main goal of this thesis is to design a hybrid collision avoidance algorithm from components of the Velocity Obstacle and Multi Objective Particle Swarm Optimization algorithms. The purpose was to improve the performance in areas that were weaknesses in the original algorithms. The original and hybrid algorithms were run to a selection of tests, which found that the hybrid algorithm improved performance in certain areas compared to the algorithms it was based on.

Preface

This project is the final thesis of a bachelors degree in engineering at Department of ICT and Natural Sciences (IIR) at the Norwegian University of Science and Technology (NTNU).

Acknowledgements

We would like to thank our supervisors Anete Vagale (NTNU), Aleksander L. Skrede (NTNU), Robin T. Bye (NTNU) and Ottar L. Osen (NTNU).

Arild Valderhaug
Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy
Ola Sæterøy
Ålesund, May 2021

NTNU
Norwegian University of Science and Technology

Contents

List of Figures

List of Tables

Abbreviation

AIS	Automatic Identification System
ASV	Autonomous Surface Vehicle
AWSA	Adaptive Wolf Colony Search Algorithm
COLREG	Convention on the International Regulations for Preventing Collision at Sea
DCPA	Distance of Closest Point of Approach
DWA	Dynamic Window Approach
GNC	Guidance, Navigation and Control
GPF	Generalized Potential Field
GPS	Global Position System
IDE	Integrated Development Environment
IMO	International Maritime Organization
LIDAR	Light Detection And Ranging
LOS	Light Of Sight
MOPSO	Multi Object Particle Swarm Optimisation
MPC	Model Predictive Control
MSS	Marine Systems Simulator
NTNU	Norwegian University of Science and Technology
RADAR	Radio Detection And Ranging
ROS	Robot Operative System
TCPA	Time to the Closest Point of Approach
USV	Unmanned Surface Vehicles
VFF	Virtual Force Field

- **VD** Voronoi Diagram
- **VO** Velocity Obstacle
- **VOPSO** Velocity Obstacle Particle Swarm Optimization
- **WSA** Wolf Colony Search Algorithm

NTNU
Norwegian University of Science and Technology

Concepts

Algorithms is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.

AIS Radio detection system used for collision avoidance.

ASV Boats or ships that operate on the surface of water without a crew.

COLREG is the equivalent of traffic rules for boats.

Gazebo is an open-source 3D robotics simulator.

Heuristics is a technique designed for solving a problem more quickly when classic methods are too slow, or to find an approximate solution when classical methods fail to find ant exact solution.

Hyper-V Software for Virtual machine servers.

LIDAR A method for determining ranges by targeting an object with a laser and measuring the time for the reflected light to return to the receiver. Used to make digital 3-D representations of areas or surroundings.

MATLAB A numeric computing environment which allows execution of complex mathematical operations, including algorithms.

PyCharm An IDE developed by Jetbrains for writing programs in Python.

ReSharper An IDE developed by Jetbrains for writing programs in C++.

ROS An open source collection of frameworks for robot software development.

Simulation is an approximate imitation of a process or system representing its operation over time.

Simulink is a block diagram environment for Model-Based Design.

Ubuntu Linux based PC Operative system.

Notations

- v_i - Velocity of the i^{th} particle.
- x_i - Position of the i^{th} particle.
- r_1 and r_2 Random numbers.
- $P_{best,i}$ - Personal best of the i^{th} particle.
- g_{best} - Global best.
- c_1 and c_2 Acceleration Coefficients.
- W - Inertia weight

1 Introduction

Maritime transportation is a large part of the Norwegian economy, seeing as it has the second longest coastline in the world. An ambitious goal for the future is to implement a network of autonomous ships transporting goods and people along this coastline. This switch to autonomy introduces a great many challenges, and requires efficient algorithms which can respond to a wide array of situations.

One such situation is the interaction between large transport ships and smaller vessels, which can often act in an unpredictable manner. Even though there exists a set of rules for collision prevention (COLREGs), there is always the risk of human error that results in rules being broken or misinterpreted. An interaction between an autonomous ship and a human operated vessel is a realistic scenario that may happen often in the near future, and for that reason it is important that autonomous vessels have effective collision avoidance algorithms.

The project is focused on the simulation of several collision avoidance algorithms. The subject of interest is the performance of the algorithms themselves, which means an existing simulator will be used in their implementation. The algorithms in focus are Velocity Obstacle and Multi Objective Particle Swarm Optimization (MOPSO), and the hybrid algorithm proposed will be a combination of these. These are local algorithms with collision avoidance as their primary purpose, and the tests they are run through will focus on this. The goal is to have produced a hybrid algorithm that performs better in these tests than the original algorithms it is based on.

1.1 Report Structure

Chapter 1 - Introduction contains the background of the assignment, and the project structure.

Chapter 2 - Theory contains the theoretical basis that is needed for decisions throughout the project.

Chapter 3 - Development Process contains information on the process that was followed to complete the bachelor.

Chapter 4 - Requirements describes the required foundation for the simulator and algorithms.

Chapter 5 - Technical Design describes the foundation of the chosen solution the group are to utilize.

Chapter 6 - Implementation contains the more technical details of the solution that were implemented and its implementation.

Chapter 7 - Results contains the results from the algorithm tests.

Chapter 8 - Discussion contains an evaluation of the groups methods and results, with limitations, changes, improvements and possibly errors that can occur.

Chapter 9 - Conclusion contains the final summarize from the findings and discussion section.

Appendix A - COLREGs contains a excerpt of the COLREGs rules relevant to the thesis.

Appendix B - Progress Plans contains several progress plans throughout the project.

Appendix C - Pre-project Report contains the Pre-project report from the start of the project.

Appendix E - Source Code contains the source code of the simulator and the algorithms.

2 Theory

This section presents the definition of an autonomous ship and how to a achieve autonomy, as well as path planning and collision avoidance algorithms that have been researched throughout this thesis. Further on it presents different simulators used for simulating path planning and collision avoidance algorithms.

2.1 Autonomous Ship

Autonomous ships are based around the concept of a ship utilising different information to automatically locate, detect and predict the situation around a vessel, to be able to make a decision about how the ship must orient itself at sea to reach its destination.

2.1.1 Requirements to Achieve Autonomy

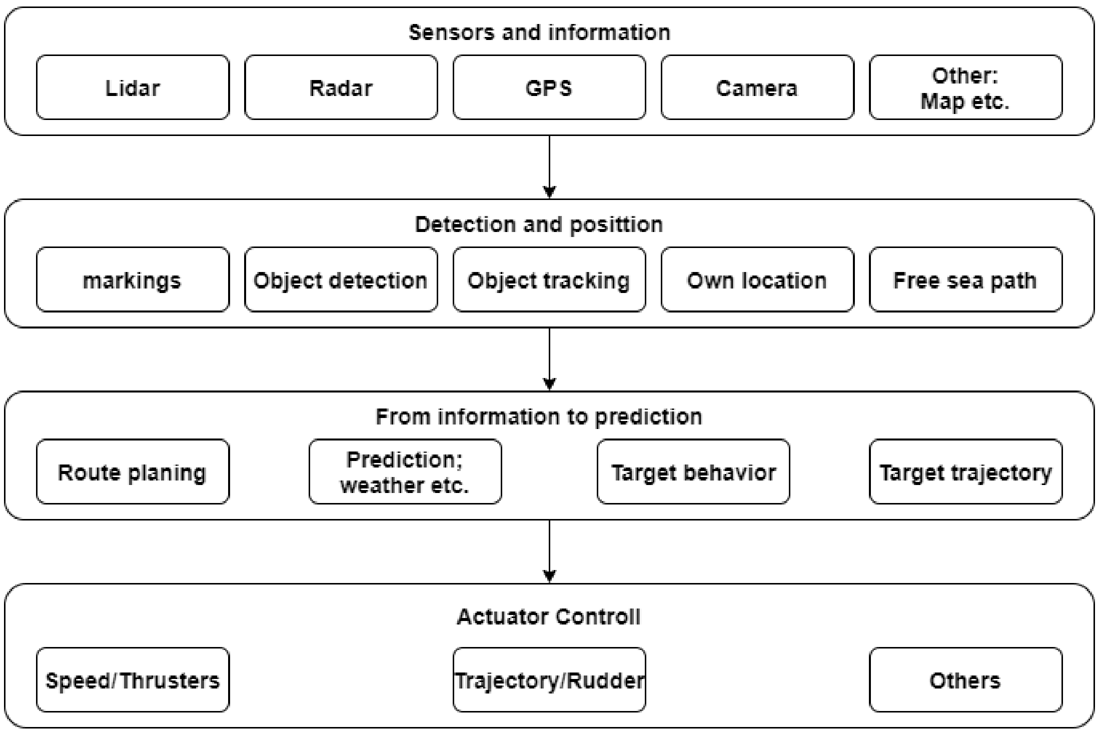


Figure 1: Autonomous ship architecture components

The first requirement for achieving autonomy is information. A constant stream of information is gathered by using technologies such as LIDAR, Radar, and GPS, as shown in Figure 1.

The third step is utilizing the information to calculations, predictions and make decision. This step is highly technical with many algorithms and rules that need to be followed, and is the reason why there still are no fully automated ship traveling up and down the Norwegian shoreline.

The last step is actuate the decisions to control the ship by implementing new directives.

2.1.2 Autonomy Levels

There is no specific method or rule for how to identify a vessel as an autonomous vessel. But there are a countless number of articles about how to identify different levels of autonomy. The main points of identifying the level of autonomy a vessel has, is to know how great of a role the vessel is in the decision-making process, and how much a human must intervene.

Generally autonomy levels can be categorised as shown in Table 1.

Level	Short description	Description
1	Manual operation or Remote control	Operator is in control of gathering information and taking decisions
2	Operator assisted	System recommend decisions for the operator
3	Partly autonomous	System take decisions, but the operator can override and change decisions
4	Autonomous	The system is planing, predicting and taking decisions by itself.

Table 1: General Autonomy Levels [21]

2.1.3 Autonomy Modes During Voyage

In order to make a vessel autonomous, there is a need to understand what environment the vessel is supposed to travel in. There is a big difference in the information one needs when operating that ship inside a port, versus out at sea. Figure 2 shows the autonomy modes during a voyage.

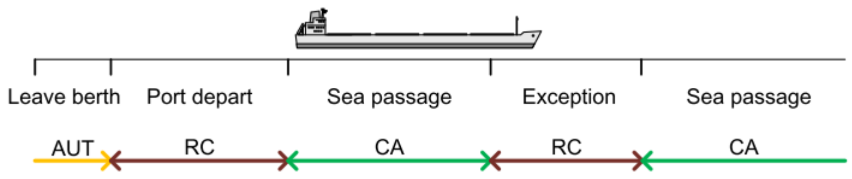


Figure 2: Changes in autonomy modes during voyage [21]

Port Arrival/departure: To manoeuvre a ship in a port requires very high situational awareness, and knowledge about how the ship reacts with limited room to maneuver.

There is also a need to know where to dock, and the area to moor the ship to the dock. This poses great difficulties for autonomous ships without a dedicated port.

Sea passage: Main challenges when the ship is at sea are weather, unpredictable vessels, and keeping the ship on course. If traveling along the coast it becomes a higher priority to avoid shallow waters.

At sea exception: Areas with limited area to evade other ships. Artificial waterways. Drawbridge that is open a limited time. Areas only open when high tide or low tide. Are some areas where autonomous ships need human intervention.

2.2 Algorithms

An algorithm is a list of instructions and rules designed to perform a specific task [29]. Algorithms are used to process data and perform calculations or actions in various ways, such as multiplying two numbers, or a complex operation, such as playing a compressed video.

In computer programming, algorithms are often created as functions to serve larger programs. For example, a simulator uses an algorithm to run a scenario while the simulator program displays it.

Since there are multiple ways to perform a specific operation within a software program, programmers strive to create the most efficient algorithms possible [13]. An algorithm should be so well defined throughout all aspects that it only leads to one unambiguous result. The benefit of efficient algorithmsis that program use as little system resources to quickly and efficiently produce reliable results.

2.2.1 Global and Local Path-planning Algorithms

Path-planning algorithms can be classified into either global path-planning algorithms [18] or local path-planning algorithms [30]. Global path-planning is often performed before the actual journey begins. It evaluates the whole set of available information for a specified area, and based on this information a safe path is generated from the point of departure and the destination. The generated path must satisfy certain constrains, such as the dynamic constraints of the vehicle and environment-related constraints. These constraints can be weather conditions and minimum distance from obstacles or land. Some known algorithms that are using global path-planning is the Voronoi Diagram (VD) [7], the Potential Field Method [4], and the A* algorithm [1].

The Local path-planning on the other hand is most often performed in real-time, and it takes the surrounding space around the vessel into account [18]. This leads to local path-planning algorithms being more fitted for dynamic obstacle detection and other unpredictable factors. Some known algorithms that are using local path-planning is the Dynamic Window algorithm [8], the Virtual Force Field algorithm [6], and the Velocity Obstacle algorithm [9].

2.2.2 Pathfinding

Pathfinding is a complex operation, but there are many reasons for implementing it [28]. Consider a vessel on one end of an area attempting to reach the other end. Without pathfinding the vessel scans a small area around the vessel to check for obstacles (shown in pink in Figure 3), afterwards it will continue its scan in small areas while simultaneously going straight towards the goal until a obstacle is discovered. Once detected the algorithm will change direction to avoid the obstacle, following the red path in 3. A path finding algorithm however would scan a larger area(highlighted in light blue), and with that search discover a shorter path and avoiding getting stuck in the obstacle (Blue line in Figure 3).

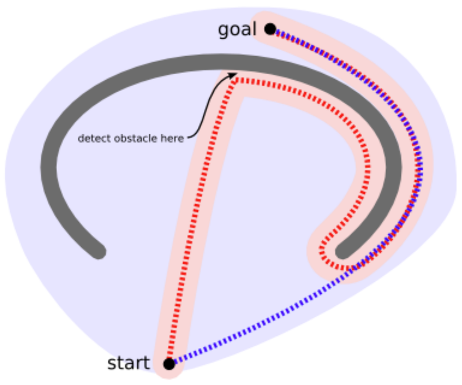


Figure 3: Obstacle Pathfinding [28]

Pathfinding has the advantage of letting the vessel plan ahead, rather than reacting once a obstacle is discovered [28]. There is however a trade-off between pathfinding and reactionary algorithm in that pathplanning is generally slower, but gives a better result by not having as high a risk of getting stuck. It is recommended to use pathfinding algorithms for longer distances, where obstacles are relatively static. On the opposite end of the scale a reactionary algorithm will generally do better in smaller local areas like docking or going into harbour where obstacles are more varied and the distance is shorter.

2.2.3 Graph-based Pathfinding

The A* algorithm and its predecessor the Dijkstra's Algorithm operates and takes decisions based on weighted graphs [24]. The graphs consist of vertices also known as nodes which are connected by numeric weighted edges. The graph-based search algorithms needs to know the location and which of the nodes that are connected and how they are weighted.

A mathematical graph consist of a set of nodes and edges, see Figure 4.

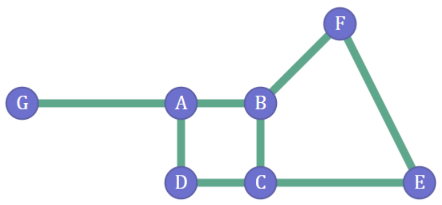


Figure 4: A Mathematical Graph [24]

For any graph two things need to be know:
1: Set of nodes in the graph
2: Set of edges from each node

A: A→BA→DA→G
B: B→AB→CB→F
C: C→BC→DC→E
D: D→CD→A
E: E→CE→F
F: F→BF→E
G: G→A

2.2.4 Dijkstra's Algorithm

The graph-based Dijkstra algorithm uses the cost from the graphs to find the shortest path from starting point of the object to the finishing point [28]. From the starting point of the object the Dijkstra visits the closest vertices in the graph, and then starts to build a set of nodes with the shortest distance from the starting point. This search continues until the algorithm reaches the finishing point, and as long as the edges do not have any negative cost it will provide the shortest path possible. In the following Figure 5, the pink square is the starting point and the purple square is the goal. The cyan squares is where the Dijkstra's Algorithm performed the scanning. Farther on, the lightest colored squares is farthest from the starting point while the darker is closest.

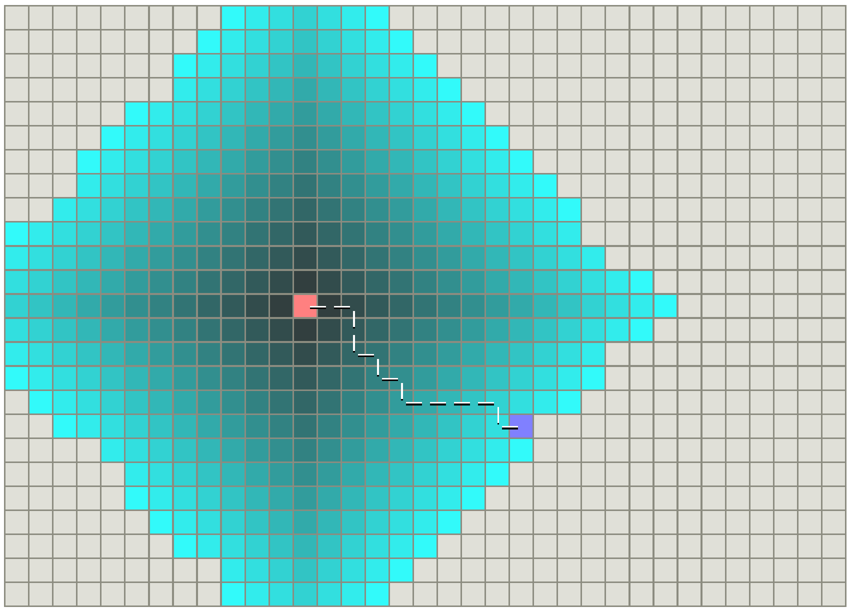


Figure 5: Dijkstra's Algorithm [28]

2.2.5 Best-First-Search

The Best-First-Search algorithm is more of a greedy algorithm as it has some estimate of how far from the goal any vertex is [28]. It works by finding the vertex closest to the goal instead of the starting point. This algorithm however is not guaranteed to find the shortest path. Best-First-Search has the advantage of running much quicker than Dijkstra's algorithm, this because it takes advantage of heuristic function to create a path towards the goal. In the following Figure 6, the pink square is the starting point and the purple one is the goal, while the yellow squares indicates nodes with high heuristic value and the dark squares represents nodes with low heuristics value.

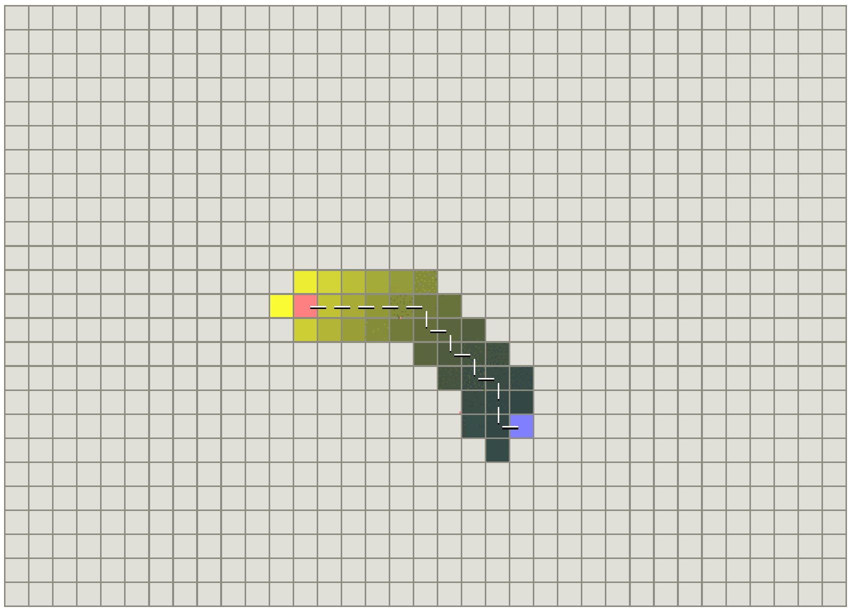


Figure 6: Best-First-Search [28]

As seen in Figure 3, the red path represents a Greedy Best-First-Search path, this can be seen in that it tries to move towards the goal. This happens because the algorithm ignores the cost of the path, and instead only focuses on the cost to get to goal. This is turn means that it could possibly create a path with high cost before reaching its goal.

2.2.6 A* Search Algorithm

The A* search algorithm is a graph traversal and path search algorithm, and it is the algorithm used for path planning in this project. One if the major drawback of a* is its memory requirement, as it stores all generated nodes in memory [31, 1]. The A* algorithm was first published at Stanford Research institute in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael, as a result of a path planning algorithm for the Shakey project [31]. The algorithm can be considered an extension of Dijkstra's algorithm, where A* is using trial-and-error to guide its search.

The A* search algorithm is a best-first search, meaning that it is formulated in terms of weighted graphs [31, 26]. The algorithm starts at a given starting node in a graph, it attempts to create a path to the goal taking cost of the path into consideration as seen in Figure 7. A* decides which of it's path to extend by calculating the cost of all existing paths to the target. It's important to note that A* does not attempt to create the shortest path to the goal, but rather the least costly path, determined by the cost of each section of the grid.

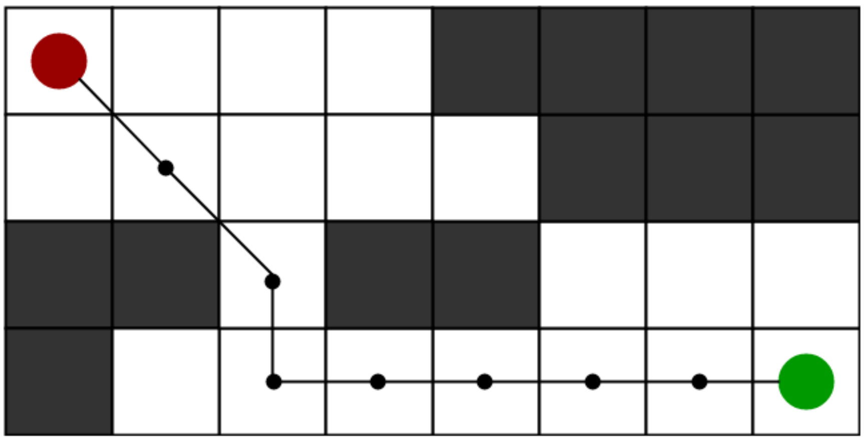


Figure 7: A* Algorithm [26]

One can say that the A* algorithm is the best of both world from both the Dijkstra's algorithm and the Greedy Best-First-Search algorithm. It can be used to find the shortest path as the Dijkstra, while using the heuristic from the Greedy Best-First-Search algorithm [28]. In simple cases it can be as fast as the Greedy Best-First-Search algorithm. It uses the information of which vertices that are close to the starting point from the Dijkstra's and the information of which vertices that are closes to the goal from the Greedy Best-First-Search algorithm.

In A* terminology, $g(n)$ represents the exact cost of the path from the point of origin to a vertex n, and $h(n)$ represents the estimated cost from any vertex n to the pre-defined goal [28]. In Figure 8, the vertices are more yellow the further they are from the goal (h), and more teal the further they are from the starting point (g). Each repetition through the main loop of the algorithm the A* balances the two as it moves from the starting point to the goal. It examines the vertex n that has the lowest sum of equation 1.

$$f(n) = g(n) + h(n) \tag{1}$$

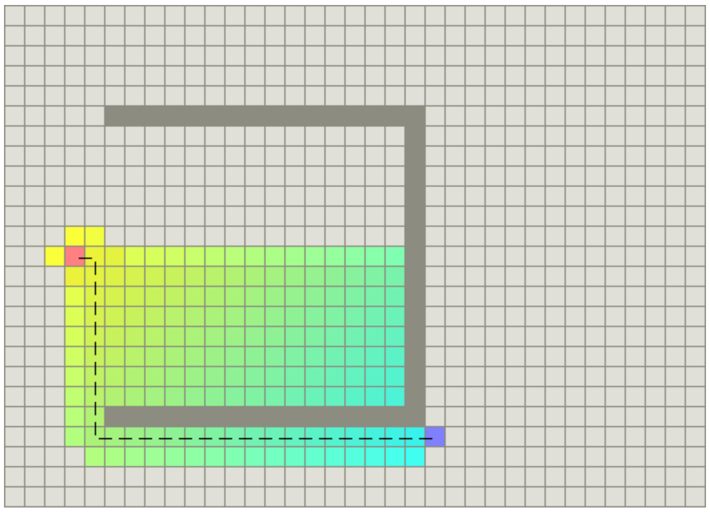


Figure 8: A* Algorithm Trap [28]

2.2.7 The Virtual Force Field Method

The Virtual Force Field Method (VFF) is a combination of using certainty grids for obstacle representation, and potential fields for navigation [6]. This combination is especially capable of accounting for inaccurate sensor data, and enables the robot to move continuously, as opposed to a staggered movement where it stops in front of obstacles before recalculating its path. Krogh and Thorpe [5] suggested a combined method for global and local path planning, which uses Krogh's Generalized Potential Field (GPF) approach. These methods however assume a know and prescribed world model of the obstacle. The combination of the potential field method with a certainty grid creates a reliable control method for autonomous vessels.

2.2.8 The Dynamic Window Approach

In the Dynamic Window Approach (DWA) approach, the algorithm searches for velocities that are can be reached within the dynamic constraints of the vessel [8]. Additionally the search area is limited with respect to obstacles in the travel path. Defining the search space by these criteria is done in the first step of the algorithm. In the second step the optimal velocity is chosen from the velocities considered valid. The vehicle dynamics are taken further into consideration as the dynamic window is reduced to velocities which can be reached within a given time interval. All maneuvers outside the dynamic window are not considered when calculating the obstacle avoidance.

2.2.9 Line Of Sight

The simulator contains a simple Line Of Sight (LOS) algorithm. This algorithm is based on a 2D field of view from the vessel [11]. More complex 3D LOS algorithms uses the elevation observer points and the points on the terrain of interest as the criterion to perform visibility analysis. Some of them also takes the distance between the observer and target point, and the velocity and direction of moving objects into consideration. The LOS algorithms are still very simple algorithms that can not easily be modified to consider additional criteria, such as the need to deviate from the original route entirely.

The LOS algorithm steers the vessel onto the line between the waypoints as shown in Figure 9.

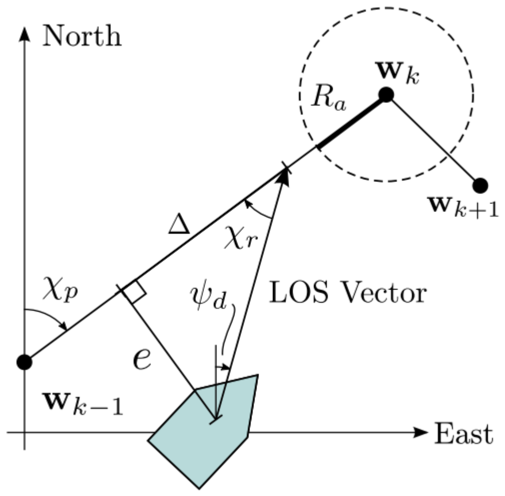


Figure 9: LOS guidance [16]

The two criteria used by the LOS algorithm in the simulator is the circle of acceptance and progress along path criteria [16]. Since the ship is following a path described by waypoints, it uses one of these two methods to know when to switch to the next waypoint. The difference between these two methods is that the circle of acceptance makes sure that all of the waypoints are visited, while the progress along path, also known as along-track, switches waypoints whenever the vessel crosses the line normal to the waypoint line as shown in Figure 10.

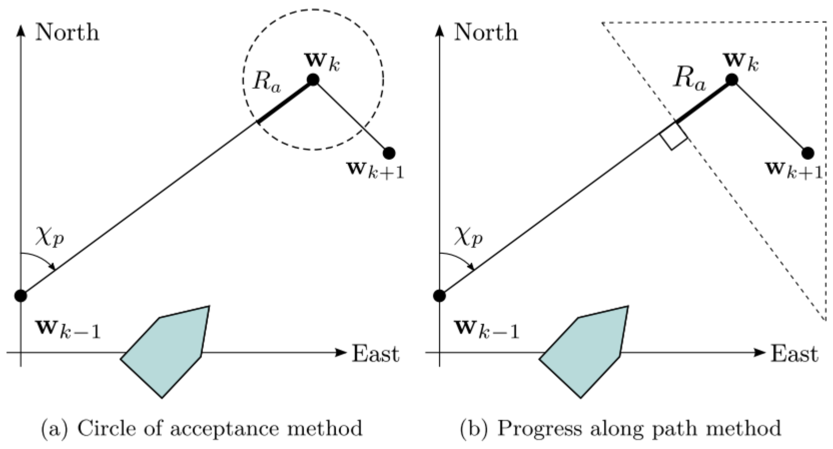


Figure 10: Circle of acceptance and Progress Along Path methods [16]

2.3 Collision Avoidance

There has been a rapid growth in the control methods for ship collision avoidance these past years. Most of then can be divided into two classes: model-based and model-free methods [22]. In 2017 Xin Wang, Zhengjiang Liu and Yao Cai [20] proposed a collision avoidance support system that quantified the distance at the closest point of approach (DCPA) and the time to the closest point of approach (TCPA) to assess the collision risk between two encountered vessels, and subsequently employed a proportional-integral-differential control method to calculate the vessel maneuvering motion. The simulation indicates that the own vessel can avoid only one target vessel, and the only maneuvering party is the own vessel while the target vessels remain at the detected bearing. Therefore, this research was confined to one ship-one ship situation.

He (et al) [19] presented a quantitative analysis system for COLREGs that included a series of judging models based on the collision risk with encountered target ships. This method can handle one ship - multiple ships situations where one ship can avoid multiple target ships simultaneously. Zhang (et al) [17] proposed a distributed anti-collision decision support formulation using a decision tree in multi-ship situations under COLREGs. This approach can manage complex scenarios with multiple target ships. Model predicitive control (MPC), a popular mode-based method, can compute an optimal trajectory based on the target ships motion prediction while considering its uncertainty. Within this approach it is possible to incorporate models of the target ships motion, the evolution of the dynamic environment, and different operational constraints [17].

A reliable and robust collision avoidance system is crucial for an Autonomous Surface Vessel (ASV) [16]. To operate and behave at sea it needs to adhere the International Regulations for Avoiding Collision at Sea (COLREGs). The Velocity Obstacle method can be used as the basis for collision avoidance, where it takes on the main COLREGs scenarios: overtaking, head-on, and crossing.

2.3.1 Velocity Obstacle

The Velocity Obstacle algorithm (VO) creates an approximation of the velocities that would cause a collision with an obstacle at some point in the future [9]. The Velocity Obstacle represents the velocities that will eventually cause a collision, and the vessel can adjust well in advance of the collision by simply selecting a safe velocity. It is one of the simplest methods for collision detection, but it still requires obstacle velocities data.

It is possible to use the velocity obstacle algorithm for single and multiple obstacles [9]. When it is used for multiple obstacles it may be useful to prioritize the obstacles so that those with imminent collision will take precedence over those with long time to collision. The algorithm is not fit to predict remote collision as it may be inaccurate if the obstacle does not move along a straight line.

2.3.2 Adaptive Wolf Colony

The Wolf Colony Search Algorithm (WSA) [15] is based around the hunting tactics where communication the leading wolf, detective wolves, and fierce wolves is used to imitate the hunting behavior of a wolf colony. This algorithm searches the solution space by taking advantage of the moving behavior of detective wolves; the leading wolf will use his summoning power to move the wolf colony toward promising regions where the it is most likely to find prey; the combination of detective and fierce wolves gives WSA a fine search ability among the promising solutions.

The Adaptive Wolf Colony Algorithm (AWSA) [15] is a Search algorithm based on WSA. AWSA can adjust the step lengths in real-time, based on the current search space and directions. The AWSA addresses the problems associated with low convergence rate caused by boundary overstepping of WSA in the encircling process.

Figure 11 is showing the flowchart of how to Adaptive Wolf Colony Algorithm is operating.

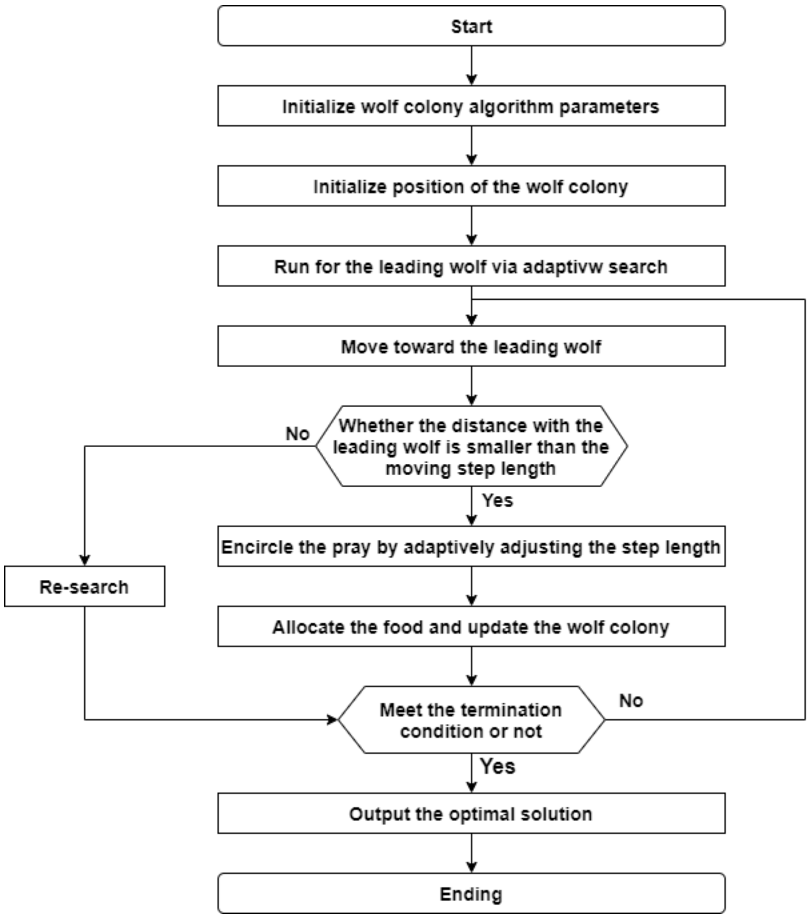

Start
Initialize wolf colony algorithm parameters
Initialize position of the wolf colony
Run for the leading wolf via adaptivw search
Move toward the leading wolf
Whether the distance with the leading wolf is smaller than the moving step length
No
Yes
Encircle the pray by adaptively adjusting the step length
Re-search
Allocate the food and update the wolf colony
Meet the termination condition or not
No
Yes
Output the optimal solution
Ending

Figure 11: Adaptive Wolf Colony Search Algorithm flowchart [15]

2.3.3 Multi Objective Particle Swarm Optimization (MOPSO)

Particle Swarm Optimization (PSO) proposed by Kennedy & Eberhart [33] is inspired by the social behavior of some biological organisms, like the choreography of a bird flock who synchronously often change direction suddenly, scattering and then regrouping. Coello Coello & Lechuga describe it as a distributed behavioral algorithm that performs multidimensional search [10]. In the simulation, the behavior of each individual is affected by either the best local or the best global individual. The PSO also differs from an evolutionary algorithm in that it allows individuals to benefit from their past experience. In evolutionary algorithms, on the other hand, data obtained from the current population is the only past knowledge of the individual.

Coello & Lechuga proposed an idea of using the PSO algorithm with a global repository in which every particle will deposit its information after each cycle [10]. In this way the PSO algorithm is capable of handling multiple objectives at the same time, instead of only a singular approach. The MOPSO also implements a geographical-based approach to maintain diversity.

2.4 COLREGs

COLREGs is the International Regulations for Avoiding Collison at Sea and was established by The International Maritime Organization (IMO) in 1972 [2, 3]. The COLREGs are divided into five parts:

- **Part A** General (Rules 1-3).
- **Part B** Steering and Sailing (Rules 4-19).
- **Part C** Lights and Shapes (Rules 20-31).
- **Part D** Sound and Light Signals (Rules 32-37).
- **Part E** Exemptions (Rule 38).
- **Part F** Verification of compliance with the provision of the Convention (Rules 39-41).

Since the algorithms that have been used during this thesis do not compile for either light or sound, the main focus will be around part B of the COLREGs. The rules in Part B that are relevant to this thesis are rule 8, 13, 14, 15, 16 and 17. The scenarios to be considered in this thesis are: head-on, overtaking, crossing and avoiding a vessel with irregular maneuvering. The relevant rules are listed in Appendix A.

2.5 Simulation

A simulation is an imitation or recreation of a real-world situation, a process, or a course of events. Simulation are used for performance optimization, safety engineering, testing and training. By utilising a simulator it is possible to simulate what would happen in a real life scenario without investing in expensive equipment and put materials or human lives in danger. Simulation can be graded regarding authenticity, which deals with the simulation's credibility and accuracy.

2.5.1 Gemini

The Gemini simulator [38] is a Unity-based visual simulator originally developed by graduate students at NTNU in Trondheim, Norway. The simulator began as a project for the Milliamperè Autonomous ferry. The purpose was to provide a foundation for Electromagnetic Radiation (EMR) based sensor, such as optical cameras, LIDAR and Radar for use in development and testing of autonomous applications. In addition to providing the simulated sensor data from inside the simulation environment, Gemini will expose an API that allows developers to interface and communicate with the simulated vessel(s) and environment.

2.5.2 ROS and The ASV System Package (ROS package)

The Robot Operating System (ROS) [36] is a flexible framework for writing robot software. It is a collection of tools, libraries and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. The ROS was built from the ground and up to encourage collaborative robotics software development. It has a wide variety of usage like mapping indoor environments, and could contribute a world-class system for producing maps. Further on another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter.

The Autonomous Surface Vehicle (ASV) system package [39] is a collection of ROS packets developed by Thomas Stenersen as a part of his master's thesis. The ASV system package is mix of C++, Python and CMake.

2.5.3 Marine System Simulator (Matlab/ Simulink Module)

The Marine Systems Simulator (MSS) [12] is a Matlab and Simulink library for marine systems. It includes models for ships, underwater vehicles, unmanned surface vehicles (USV) and floating structures. The library also contains guidance, navigation and control (GNC) blocks for real-time simulation.

2.5.4 Project Thesis Simulator: Stenersen (Python)

This simulator is developed by Thomas Stenersen in 2014 during his project thesis [14, 40]. It makes the foundation in his master's thesis the following year. The simulator is a path planning simulator which contains a several path-planning algorithms, including the A* algorithm used in this project. It is organized in a hierarchical manner as shown in Figure 12 By implementing core modules as separate classes, it is believed that through this strict separation the simulator is easily modified.

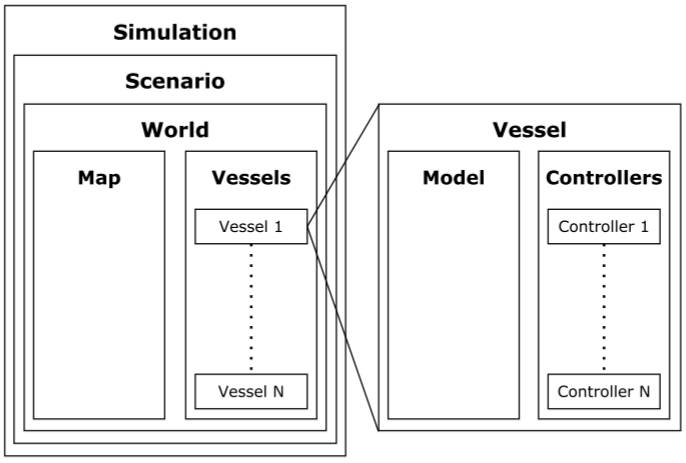


Figure 12: Simulator structure in hierarchical manner [14]

At the top of the simulator structure lies the Simulation and Scenario classes [14]. The Scenario class defines the scenario parameters such as starting positions and landmasses, and the Simulation class performs the mathematical aspects of the simulation. The Simulation class then updates the World class, which implements these calculations in the Map and Vessel classes, and checks if any of the vessels have collided. Information about the path travelled by the vessels is stored in the Simulation class, which is later used to draw the visual representation of the simulation.

The Map Class implements a map which uses sets of polygons to represent land. The Vessel Class is an implementation of a surface vessel. It instantiates the VesselModel class, as well as information required to draw the visual representation of the vessel in the simulator. The VesselModel class simulates the vessel motion by implementing a 3 degree-of-freedom vessel model.

Finally each algorithm, both local and global, are implemented as their own controller classes. These are implemented as a subclass of the Controller superclass.

2.6 Simulation Map Input

Figure 1 is a good indication for requirements for what inputs a ship simulator needs, and what tasks a complete simulator must know how to process. A big obstacle is generating the needed data and information needed to run a simulation. There is a big gap in complexity between different information a simulation utilize. More lifelike simulations require information with higher degree of complexity.

2.6.1 Map Data

Figure 13 shows how most of the free map applications like Google Maps [32] or Marine Traffic [35] is displayed. When using these kind of maps in a simulator, it will be easy to convert data to distinguish what is land and what is sea. If one use these maps to orient oneself at sea, there will be a major chance that accidents will occur.

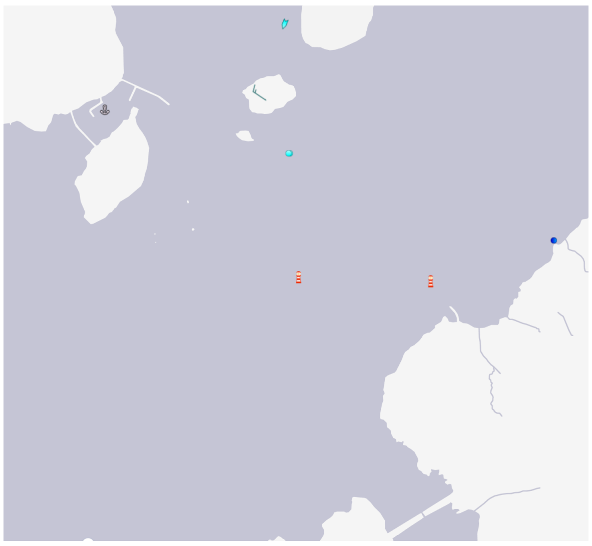


Figure 13: General Map [34]

Instead of using a general map in the simulator, implementation of a nautical chart will significantly increase the quality of the simulation. Further on will the nautical chart implementation make the simulation more similar to the reality, and therefore increase the quality and security of a possible implementation of the algorithm in an autonomous vessel. By adding a map from the Norwegian Mapping Authority as an overlay you can accomplish a much better insight into how the waters are below the surface.

A problem at sea is that it is not possible to directly see what is below the surface at first glance. When leading a boat with a deep draft, it is easy to run the boat aground when not pay attention. Sea charts contains information on where vessels can traverse in terms of depth, which are displayed as different icons, like in Figure 14. In reality these icons can be seen as lighthouses, red/ green lanterns, buoys, beacon etc., and the captain needs to investigate if his vessel could traverse in these waters.

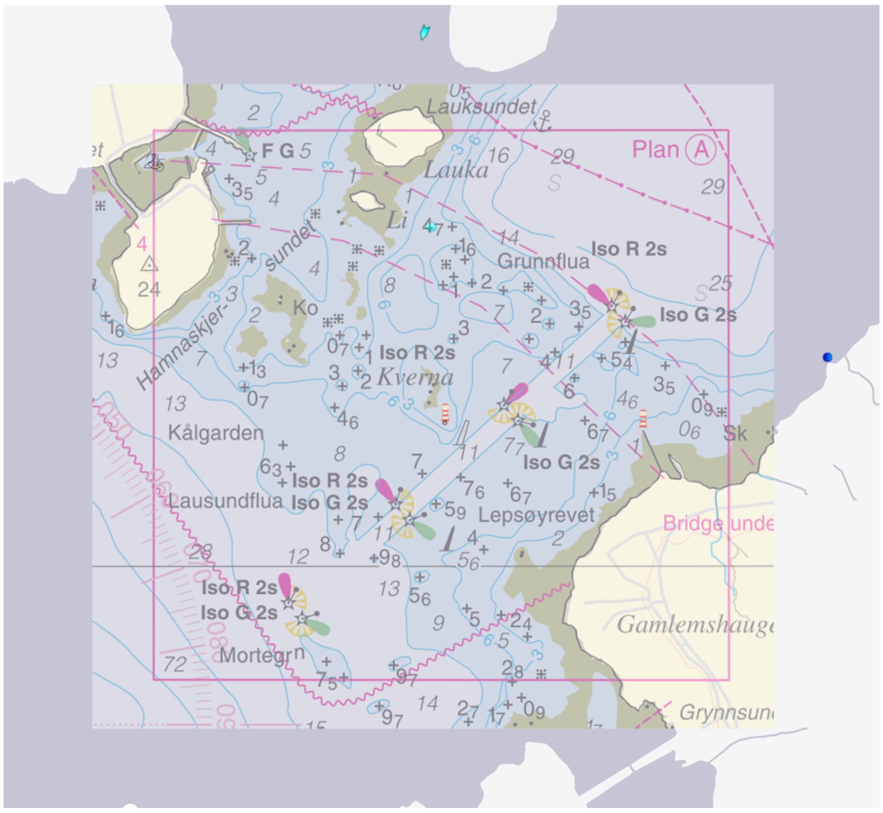


Figure 14: Map with symbols [23]

A final issue that arises when traveling by sea is the issue of tides. A map that is static will not show exactly how deep it is in its area at any given time, as the water level is determined by the sun and the moon's location in the sky. This can create problems when traveling in a ship that is deep in the waters. Even when sailing in an area at high tide, it may not be possible to when sailing in that area at low tide, like shown in Figure 15. In a simulator, this will mean that a restriction of traffic should be added for where autonomous ships should travel, which will help to reduce the chances that any ships will run aground.

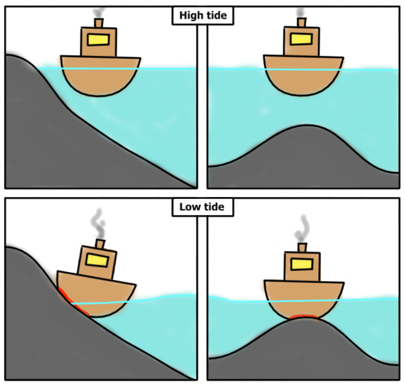


Figure 15: The difference in water level at high tide and low tide

Map data is constantly evolving, but since the ocean covers 71% of the Earth's surface [42], it is a challenge to have all sea charts up to date at all times. Through the hundreds of years humans have mapped the ocean, technology has changed drastically. As a result, there may be inaccuracies in terms of depth and how accurate the map itself is.

3 Development Process

This section describes the procedure in the work regarding the algorithms, and how the group chose to manage the given time of the thesis.

A progress plan was prepared in form of a Gantt chart to visualize timelines and deadlines for all tasks, to easily display when each section of the project should be finished. A continuous update was performed every two weeks. To get a insight into the progress of the project process, see Progress Plans [B.1], [B.2], [B.3].

The project was divided into three parts: Phases 1 through 3.

3.1 Phase 1: Planning and Documentation

The group had little to no experience or knowledge regarding autonomous ships and algorithms prior to the bachelor thesis. Therefor was it important to develop a theoretical foundation. Work tasks were delegated to systematically review as much theory as possible, and the relevant knowledge was passed on to each of the group members.

The knowledge and experience acquired in phase 1 made the foundation for phase 2.

3.2 Phase 2: Implementation and Simulation

Phase 2 creates the foundation in implementing the different algorithms in the simulator. One of the more time consuming parts of phase 2 is obtaining an insight into the workings of the simulator. This is a crucial part in terms of how to implement the different algorithms in the simulator, as well as the hybrid algorithm the group will create.

Further on in phase 2, the group decided which algorithms to compare and create a hybrid from. After the algorithms are selected, the focus was on implementing the algorithms in the simulator.

3.3 Phase 3: Finalization

Phase 3 is about gathering the results from the simulations and processes them. Phase 3 will be documented with illustrations, tables, and results before they are systematically analyzed. The main focus will be to analyze results from each of the algorithms and compare them to each other, by doing this the group will be able to localize strengths and weaknesses from the different algorithms.

4 Requirements

The requirements are based on the following items from the Pre-project Report [C]:

- The project is focused on the simulation of several algorithms for collision avoidance in autonomous ships.
- The subject of interest is the performance of the algorithms themselves, which means an existing simulator will be used in their implementation.
- There is already a great deal of work done on testing algorithms in simple simulations and situations, and the goal is to put some of the more promising ones through a more challenging test.
- The scenario envisioned introduces a chaotic element to the simulation in the form of a smaller vessel with a random aspect to its path planning.
- A selection of algorithms will be run through this simulation, The results of these tests would then be used in the selection of algorithms for further use in the creation of a hybrid algorithm.
- This hybrid will then be run through the same tests and be compared to the original algorithms.

4.1 Simulation Requirements

A preexisting simulator is to be utilized to reduce development time. It had the following requirements:

- **Licence free software:** The simulator needs to be open-source software.
- **Coding language:** Every group member are experienced in either Java, Python or both, meaning the ideal simulator would operate on one of these languages.
- **Illustration of ship movement:** The simulator should preferably have a way to produce a visual output of ship movement.
- **Ability to modify simulator:** The ability to implement new algorithms without remaking the whole simulator, as well as ability to modify the simulator to get test data information output.
- **Complexity:** The simulator must not be overly complicated, implementations of algorithms are the main focus.

4.2 Algorithms

Simulation of several algorithms for collision avoidance

- **Path-finding with target detection:** Algorithms should be able to reach the set goal while simultaneously avoiding obstacles.
- **Global and local search:** The algorithms should be within the same scope as to give better grounds for comparison.

4.3 Test Requirements

- **Scenario:**
  - Random ship implementation.
  - Different directions of approach.
- **Evasion:**
  - Natural evasion of landmass and other ships.
  - Ships needs to follow the COLREGs.
- **Test result:**
  - Minimum distance from other obstacles.
  - CPU usage time.
  - Total travel distance and time.

5 Technical Design

This section describes the decision process behind selecting a simulator and the algorithms. It explains why the different simulators and algorithms where researched, and what the main factor for each of the choices made during the thesis.

5.1 Simulator

The simulator will be used to enable a virtual replication instead of testing on a physical object. A simulator can have a higher throughput of tests and validations compared to physical testing, with the possibility to recreate scenarios. It is also possible to simulate edge cases and dangerous scenarios without the danger of destroying equipment.

The simulator will contain information on the position, heading, and velocity of the vessel. Most large vessels utilize AIS, which sends information containing ship status, position, speed, course and local weather. This information can be utilised towards preexisting maps to predict collisions and adapt courses in a simulation environment.

An open source simulator is to be utilized in this thesis. There is a number of simulation tools available towards simulating algorithms. Another important factor for which simulator will be used is time and knowledge. Since the group only has a few months to work on the project and with no prior knowledge towards making simulators, the complexity and coding language of the simulator need to correspond to what the group members have prior experience with.

5.1.1 Choice of Simulator

Following is some of the most promising simulators that were tested in Phase 1 [3.1] of the project.

Matlab Toolboxes

There was a limit of simulators the group was able to find that were made for Matlab [41], the ones the group found were heavy integrated into the algorithms code. This made it problematic when trying to implement new algorithms into preexisting code. The second limitation was the time Matlab utilizes when executing code, especially as the amount of code increases.

Marine System Simulator

Marine System Simulator [12] were one of the toolbox that were tried in Matlab. This simulator was focused on measuring changes in actuator and vessel dynamics, while the group were interested in a more visual approach.

Gemini

Gemini[38] is a simulator produced by students at NTNU Trondheim, Norway, and it is a simulator for testing and verifying algorithms and systems for autonomous vessels. The main problem with the simulator are its programming language C# and C++. With no prior experience with Unity, C++ and C# the learning curve would be too high when considering the limited time before the deadline.

Gazebo and ROS

A combination of Gazebo [37] and ROS [36] was the first simulator the group tried to utilize. Gazebo was to be used as the 3D simulator, whilst ROS provides libraries and tools to develop the software and create the autonomous ship application.

One of the main motivations for utilising ROS was the open source materials that is available around the ROS framework and its use in robot competitions around the world. However, it is dependent on an installation of Ubuntu, which provided some problems during installation. For this reason it was decided to look for an alternative.

Project Thesis Simulator: Stenersen

Finally a simulator created by a former Masters level student at NTNU, Thomas Stenersen, was considered. This Stenersen simulator [14] is one that the group essentially stumbled over while researching available simulators. However, several aspects made it seem well suited to meet the needs of the project.

The first thing was that it was a visual simulator that required only Python to run. This made it especially tempting compared to ROS in that it was during troubleshooting of Linux installations when the group discovered this alternative. The visual aspect was also a positive as it produced a simple animated representation of the simulation with the option to save each animation to a file.

The second appeal of the Stenersen simulator was that it implemented a simple kinematic model for a variety of vessels. This would make the course alterations of the collision avoidance algorithms more realistic, and combined with the visualisation feature it produced a clear representation of the entire maneuver.

Ultimately it was the modular nature of the simulator that led to the decision to use it. It was neatly organized, which each type of vessel and vessel controller separated into their own classes. This meant that adding the algorithms to the simulator was as simple as creating new controller classes and adding them to the existing structure.

The Stenersen simulator had some global path finding algorithms already implemented. Thanks to the modular design of the simulator the collision avoidance algorithms could piggyback off these global algorithms and only kick in when there was a potential risk of collision.

The Stenersen simulator is of course not perfect. For one it does not implement environmental disturbances such as wind, current, or waves. Another downside is that it calculates each step of the simulation before moving on to the next, which means the simulation is not truly real-time. However, this is not considered to be detrimental to the aims of this project.

5.2 Algorithms

The simulator came with A* already implemented. This was a definite bonus, as it served as a global path planning algorithm. The simulator also came equipped with a Line-of-sight path following algorithm. This meant the only required work was to implement the local collision-avoidance algorithms.

The first algorithm the group decided upon was Velocity Obstacles. This collision avoidance algorithm was designed with moving robots in mind, and for that reason it was found to be very well suited for the purposes. In addition, it was intriguing due to its implementation of simple geometry to produce very positive results.

Adaptive Wolf Colony was originally considered for the second algorithm, but due to it being both local and global algorithm it would not be a fair comparison against Velocity Obstacles. Ideally the two implemented algorithms should be as similar as possible, so the decision was made to scrap Adaptive Wolf Colony in favour of a purely local algorithm.

Since the project is focused on collision avoidance, it was necessary that all algorithms were within the local scope. After some research the group decided upon Multi Objective PSO since it seemed interesting to implement a machine learning based algorithm for comparison. Another reason for this choice was that the group quickly visualized how an hybrid algorithm would be made with MOPSO as the second base algorithm.

Both selected algorithms also has the potential of being able to take COLREGs into consideration while avoiding obstacles. The main difference between Velocity Obstacles and multi objective PSO is the fact that MOPSO does not make changes in the ships velocity while path planning, and Velocity Obstacles does not make any consideration of the avoidance maneuver beyond avoiding collision.

Finally, the general ideas behind the two algorithms suggested that they might interact in an efficient manner. For instance, if both were search algorithms there might be some conflict arising in the different methods for calculating cost values when combining them. It appeared that Velocity Obstacles was well suited to combine with a search algorithm like MOPSO, as the geometry of VO could be used to influence the cost function of the MOPSO.

5.2.1 Global Path Planning

With the priorities on implementing local algorithms for collision avoidance, a choice was made to utilize the existing A* algorithm from the simulator as the global path planing algorithm. This provides a more stable basis for setting the algorithms at the same level at start-up. Velocity Obstacle and Multi Objective PSO are to take over from the waypoint following algorithm when a collision is detected, and return to following the A* generated path when the collision is avoided.

5.2.2 COLREGs Implementation

The goal of implementing COLREGs is to make the vessel in the simulator act as similar to a real ship in a real situation as possible. Since the COLREGs is a well defined collision regulation, it would make the simulator more reliable when running the simulations compared to how the vessel should act in the real world.

5.3 Testing

This section discusses the types of scenarios the algorithms will be tested in, as well as what data will be extracted from the tests.

5.3.1 Scenario

The scenarios utilize three different approaches of collision avoidance. In all of the scenarios the ASV starts from a southern starting point and have a northern finishing point. In one of the scenarios the ASV encounters a vessel passing from starboard. In the second scenario the ASV encounters a vessel with a head-on course. Finally the ASV will encounter a stationary vessel in its travel path.

5.3.2 Test Variables

The simulation will produce some data points from the scenarios. These include travel distance and time, CPU usage time for collision avoidance calculations, and the closest distance the vessels pass. This data, along with with a visual representation of the vessel movements, will form the base for comparing the algorithms.

6 Implementation

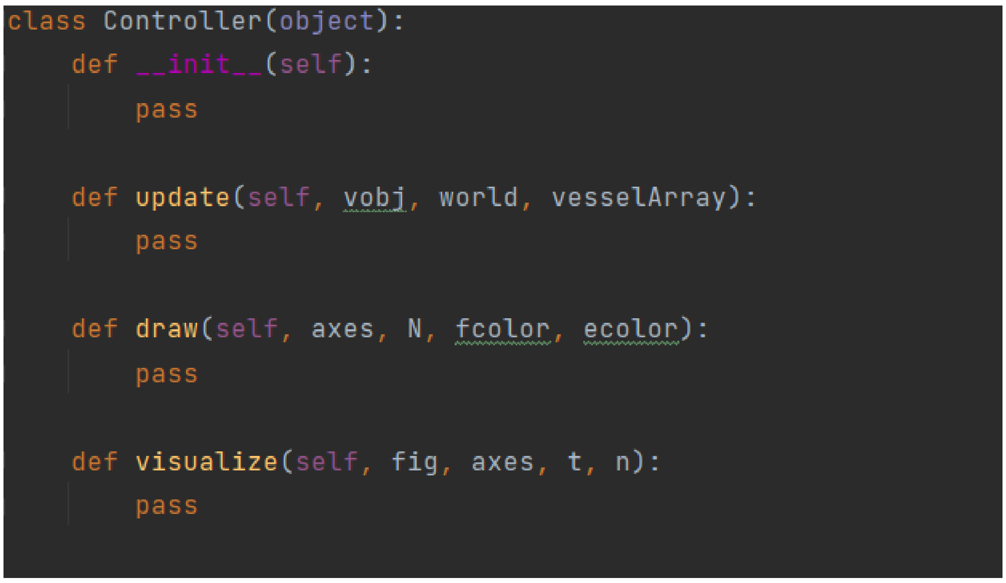
This section describes how to implementation was performed, and the challenges that occurred throughout the process.

6.1 Simulator and Algorithm

In the initial project design there were plans for implementing COLREGs and landmasses as part of the test scenarios. As the end of the project period approached, it was decided to abandon these criteria to focus on successfully implementing the algorithms. In addition, there were some difficulties combining the randomly moving vessel with the MOPSO algorithm. Because of these issues, there were no landmasses, randomly moving vessels, or adherence to COLREGs in the test scenarios.

6.1.1 Simulator

Due to the modular design of the simulator, creating new controllers was an easy process. The motion of the vessel is controlled by algorithm implemented as a subclass of a Controller superclass. Implementing new algorithms simply required them to be added as controllers extending this superclass. The structure of which is shown in Figure 16.

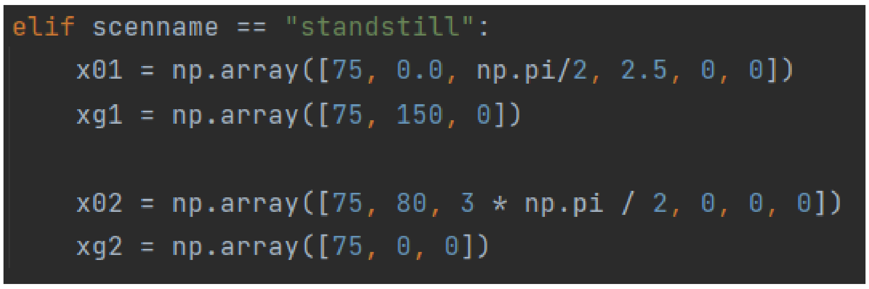


```python
class Controller(object):
    def __init__(self):
        pass

    def update(self, vobj, world, vesselArray):
        pass

    def draw(self, axes, N, fcolor, ecolor):
        pass

    def visualize(self, fig, axes, t, n):
        pass
```

Figure 16: The framework of the Controller superclass

The draw and visualize methods are used in the visualization part of the simulator, and did not need to be altered for this project. The update method is the one that impact the performance of the vessel. To make it work, two variables to be passed to this method: world, which contains information about the map, and vesselArray, which contains information about the vessels in the simulation. These new algorithms were added simply by creating new subclasses of this Controller.

The test scenarios were easily implemented in the Scenario class of the simulator. New scenarios were made simply by copying the existing scenarios, and change the parameters to suit the design requirements. An example of a scenario setup is show in Figure 17.



```
elif scenname == "standstill":
    x01 = np.array([75, 0.0, np.pi/2, 2.5, 0, 0])
    xg1 = np.array([75, 150, 0])

    x02 = np.array([75, 80, 3 * np.pi / 2, 0, 0, 0])
    xg2 = np.array([75, 0, 0])
```

Figure 17: The setup parameters for the standstill scenario

The start position array contains the x and y coordinates of the vessel, as well as the initial heading in radians followed by the initial velocities in the x, y, and z directions relative to the vessel. The goal array contains the coordinates of the goal as well as the ideal heading of the vessel when reaching the goal.

Beyond creating new controller classes and creating new scenarios in the Scenario class,only minor adjustments in the simulator code was required.

6.1.2 Random Moving Vessel

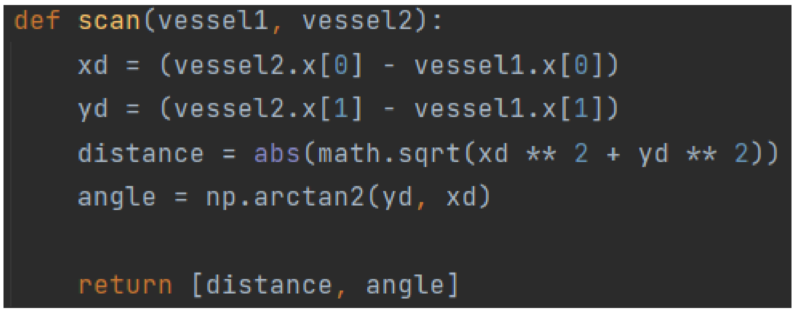
A new controller was created to create a seeded sudo-random moving ship, for testing the performance of the algorithms in terms of collision avoidance, this was done by altering a pre-existing controller from the simulator made for controlling a pursuing ship travelling the shortest possible distance towards the main ship. this was used for testing collision avoidance in a head-on collision scenario. This controller was then altered by changing the target of the controller to a randomly generate coordinate with a upper and lower limit, giving it a new coordinate every n steps decided by parameter when creating the controller object in the code. The controller can be given a seed or if no seed is given it will create a random seed itself so it's possible to rerun a scenario.

This Controller however was not used in a testing scenario due to limitations in the implementation of the MOPSO algorithm, as detailed in subsection [6.1.4].

6.1.3 Velocity Obstacle

The velocity obstacle algorithm was implemented with only minor alterations to the program. The main change that was required was to provide the main vessel with information about the location and heading of the other vessels in the simulation. In other words it was assumed that the main vessel had sensors capable of obtaining this information. Since this data is stored in an array in the World class, and the World class is responsible for running the simulation, it simply had to pass this array to the Velocity Obstacle controller.

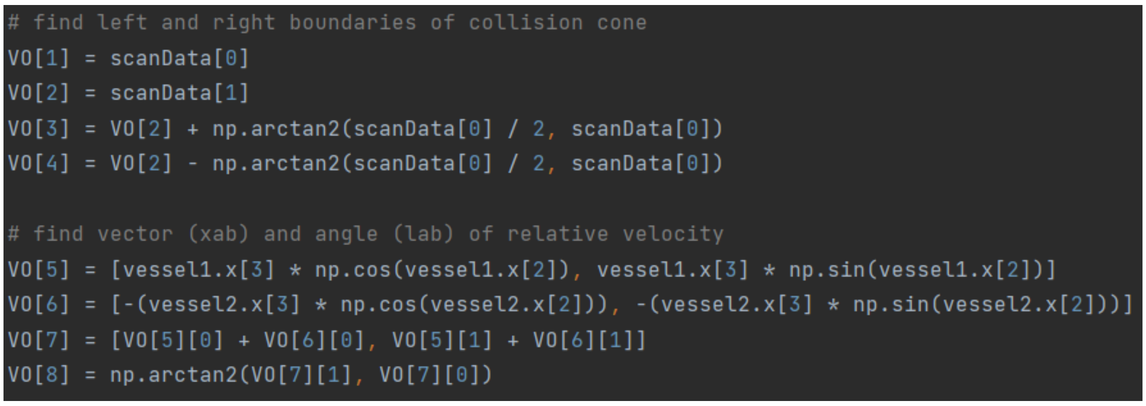
In practical terms the Velocity Obstacle algorithm is implemented almost purely using geometry and trigonometry. The first step is to obtain the distance and relative angle between the two vessels, as shown in Figure 18.



```
def scan(vessel1, vessel2):
    xd = (vessel2.x[0] - vessel1.x[0])
    yd = (vessel2.x[1] - vessel1.x[1])
    distance = abs(math.sqrt(xd ** 2 + yd ** 2))
    angle = np.arctan2(yd, xd)

    return [distance, angle]
```

Figure 18: Obtaining relative distance and angle

This data is then passed to a function which creates all required Velocity Obstacle data, represented by Figure 19.



```
# find left and right boundaries of collision cone
VO[1] = scanData[0]
VO[2] = scanData[1]
VO[3] = VO[2] + np.arctan2(scanData[0] / 2, scanData[0])
VO[4] = VO[2] - np.arctan2(scanData[0] / 2, scanData[0])

# find vector (xab) and angle (lab) of relative velocity
VO[5] = [vessel1.x[3] * np.cos(vessel1.x[2]), vessel1.x[3] * np.sin(vessel1.x[2])]
VO[6] = [-(vessel2.x[3] * np.cos(vessel2.x[2])), -(vessel2.x[3] * np.sin(vessel2.x[2]))]
VO[7] = [VO[5][0] + VO[6][0], VO[5][1] + VO[6][1]]
VO[8] = np.arctan2(VO[7][1], VO[7][0])
```

Figure 19: Calculating Velocity Obstacle data

These examples clearly show the trigonometric nature of the Velocity Obstacle implementation. By the Velocity Obstacle method, the other vessel B is transformed to a circle with diameter equal to the distance between the two vessels, and the collision cone is found by two lines with apex in the first vessel and tangential to the circle B.

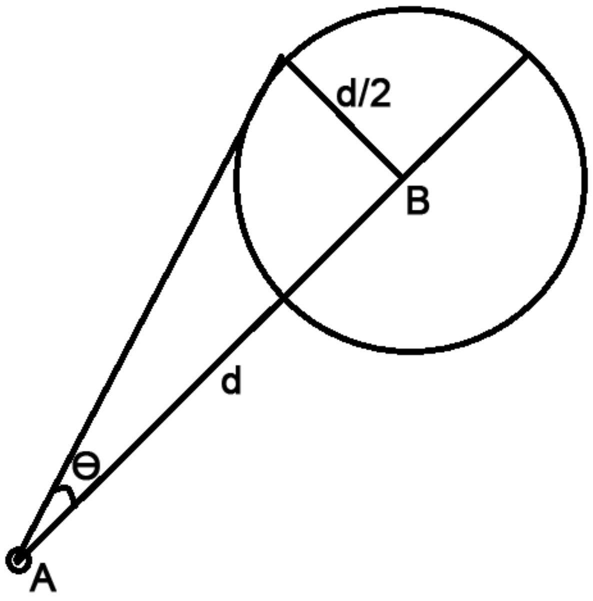


Figure 20: Calculating collision cone

The algorithm can find the angle of the collision cone by representing vessel B as a circle as seen in equation 2:

$$B_{radius} = \frac{d}{2} \tag{2}$$

It could then find the difference in angle by equation 3.

$$\theta = \arctan(\frac{d/2}{d}) \tag{3}$$

The angles in the collision cone relative to vessel A is then found to be a difference of theta on each side of the line between vessels A and B.

The next step is finding the Reachable Avoidance Velocities. These are the velocity vectors that are possible for the vessel to reach within a given time limit, and are defined by the vessels kinematics. The simulator had the values for surge and yaw acceleration predefined, so to calculate the Reachable Velocities (RV) it simply had to multiply the max acceleration with the chosen time frame. The idea here is that acceleration $\frac{m}{s^2}$ multiplied by time in seconds s produces velocity $\frac{m}{s}$. The Reachable Avoidance Velocities are then defined as the set of Reachable Velocities that do not produce a relative velocity within the collision cone.

Unfortunately the Velocity Obstacles method does not contain a method for selecting the best velocity from the Reachable Avoidance Velocities. A simple selection method was implemented, where the maximum Reachable Velocity in each direction was evaluated in order. The first velocity which was found to be an avoidance velocity would be implemented for the next time step of the simulation, after which a new collision cone would be created. The velocities were evaluated in the following order:

1. Maximum ahead starboard
2. Maximum ahead port
3. Maximum ahead
4. Maximum astern starboard
5. Maximum astern port
6. Maximum astern

If none of these velocities were evaluated as avoidance velocities, the vessel would set its velocity to 0 and hope for the best.

6.1.4 Multi Objective PSO

The Multi Objective PSO algorithm is used as collision avoidance algorithm along with A* as path finding algorithm, this means the MOPSO will only run once certain conditions are met, the condition being detection of another vessel within a set radius. Once another vessel is detected the MOPSO will start it's search and attempt to find a best possible route for the ship to avoid collision without deviating too far from it's given path. Once the ship has passed the other vessel the algorithm will do a new a* search and create a new set of waypoints towards the goal.

The multi object particle swarm optimization (PSO) algorithm was implemented by finding a PSO code-example [25] as a starting point. The code was then rewritten to fit the project and implemented into the simulation with all required alterations of the simulator already done beforehand during the implementation of Velocity Obstacle. The algorithm is describes as creating a swarm mimicking birds searching for food, each particle calculates it's new position with the use of three variables: personal best, global best, and a random variable. Once a particle has traveled to a new position a check will be done to see if the new position is better than the previous personal best as well as the global best. A new search will then be done with the new variables as explained in equation 4 and equation 5, the particles will over several iterations converge on the best coordinate calculated by it's cost function [27].

$$v_1 = Wv_1 + c_1 r_1 (P_{best,i} - x_i) + c_2 r_2 (g_{best} - x_1) \tag{4}$$

$$x_i = x_i + v_i \tag{5}$$

The cost function was created with three main variables in mind; distance from goal, avoidance of static objects, and avoidance of dynamic objects. Distance from goal is the distance of a straight line from the coordinate in question to the goal, the static object avoidance refers to avoidance of land as to prevent the ship from running ashore, this was created with a check already implemented in the simulation that checks if a coordinate is passable or not and assigning a big value to that point if the point is not passable. The final part of the cost function checks for other vessels, and was created by giving a big value to a area in front of- and around other vessels as shown in Figure 21 any point within the front zone will return a big value to the cost function, whereas the circle around the vessel will return a slight extra cost as to discourage going to close to the ship but still allowing for manoeuvres behind the ship.

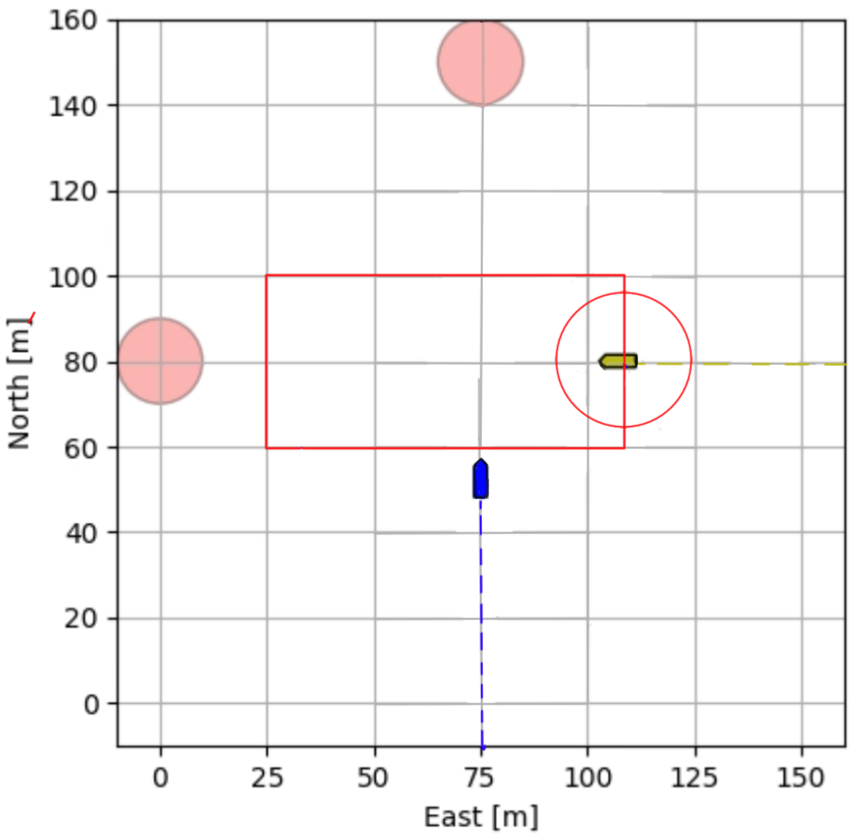


Figure 21: MOPSO dynamic object avoidance

The first version of the cost function used a cone in front of the other vessels, the main issue with this version was the PSO was able to find good point on the opposite side of the cone, or would get stuck on the incline of the cone towards the circle which in turn would lead to collision, hence it was changed to a rectangle.

6.1.5 Hybrid Algorithm - VOPSO

The hybrid algorithm proposed was a combination of the Velocity Obstacle and MOPSO algorithms, and was given the name Velocity Obstacle Particle Swarm Optimization (VOPSO). It was found to be a potentially interesting combination, it is believed that the collision cone created by the Velocity Obstacle could be used as a high cost area for the MOPSO algorithm, removing the need to implement any other ”danger zones”.

The collision cone creation was a carbon copy of the method created in the Velocity obstacle algorithm, which creates a cone representing unsafe velocities. The challenge posed by combining these algorithms was that Velocity Obstacles is a vector-based algorithm, while MOPSO is coordinate based. To account for this the hybrid algorithm would have to be altered such that either the MOPSO algorithm calculated velocities, or the VO algorithm calculated positions. The latter approach was ultimately used. This allows the swarm created by the MOPSO to search for a new best position while taking the consideration the movement of other vessels and moment an eventual collision would happen.

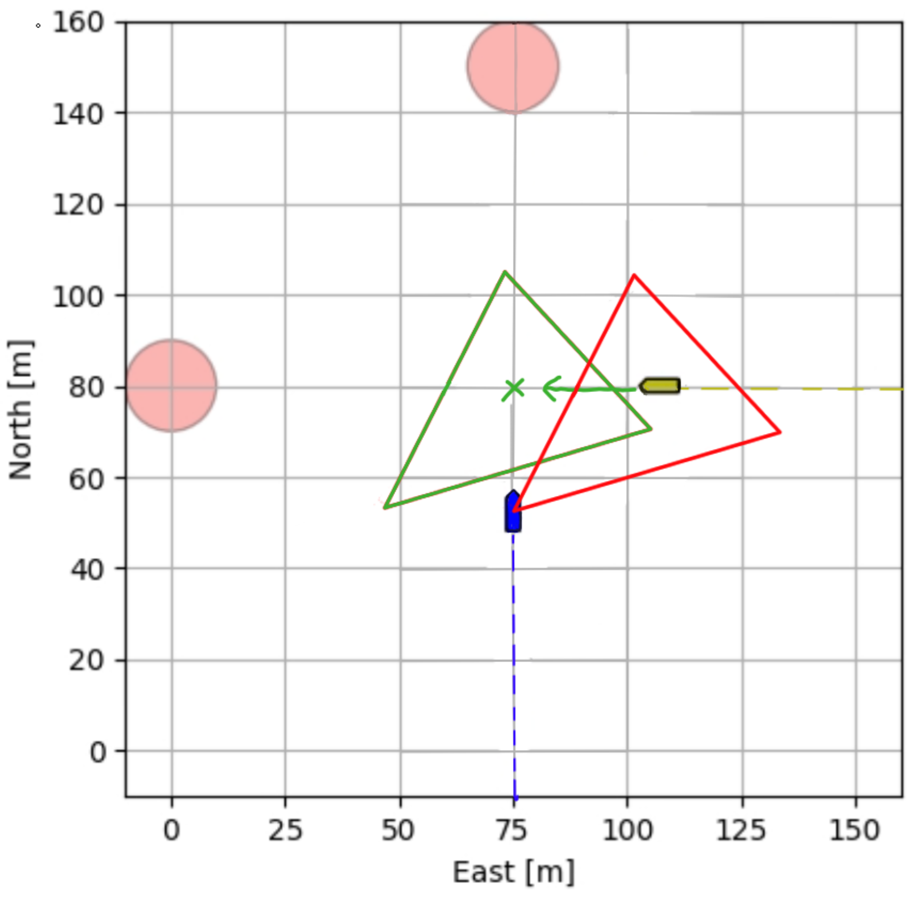


Figure 22: VOPSO Collision Cone

By calculating the time to collision, and multiplying the vertices of the collision cone with this value, the collision cone was transposed as seen in Figure 22. The idea was that multiplying the collision velocities $\frac{m}{s}$ by the time to collision s, would return the position m. The MOPSO part of the hybrid algorithm could then calculate a waypoint closest to the goal area which was not inside the transposed collision cone.

Ideally this would allow all other danger areas to be removed from the MOPSO calculations. However, the initial tests of the algorithm showed that this caused problems when the other vessel was not moving, as the relative velocity between the vessels would be equal only to the velocity of the main vessel. This combined with the angle of the cone becoming too small as the vessels were very close to each other, led the algorithm to select a waypoint just outside the cone but still so close to the vessel to create a risk of collision. It was decided to add a small high value circle around the other vessel to counteract this tendency.

6.2 Test

When designing the tests the goal was to test the basic collision avoidance functions of the algorithms. The most important function of the algorithms was to successfully interact with other moving vessels. To this end it was decided on running one test where the second vessel was crossing horizontally in front of the main vessel, and another where the other vessel was moving head on towards the main vessel. Finally, to assess the algorithms performance when interacting with a stationary object, a third scenario was created where the other vessel was at a standstill in the path of the main vessel.

For each test a selection of metrics was extracted. To measure the efficiency of the algorithms, the CPU time spent on calculating evasive maneuvers would need to be reviewed. Additionally it would be interesting to see how much longer the travel path and travel time became. The final metric was to see by how large a margin the two vessels passed each other. A simulation with only the main vessel was run to get a baseline value for each of these metrics.

7 Results

In this section, the results from the algorithm tests will be presented. The scenarios have been developed to show the difference between performance and execution from the various algorithms.

There are three scenarios considered in this thesis:

1. Ship approaching from starboard side
2. Ship approaching head-on
3. Ship standstill

7.1 Simulation Test Results

The simulated scenarios is simulated in assumed ideal conditions, i.e. no external disturbances such as wind, current or waves. The A* algorithm generates a route from start to finish without any static obstacles, and displays the ideal distance and travel time.

7.1.1 Scenario 1: Ship from Starboard Side

In this scenario, the ASV was moving in a straight line from the southern starting point to the northern finishing point. On the way a vessel was approaching from the starboard side.

The ASV performed an evasive maneuver to avoid collision with the approaching vessel as illustrated in Figure 23. The Velocity Obstacle performed a predefined evasive maneuver based on the velocity selection method, which resulted in a starboard turn once imminent collision is detected. With the lack of COLREGs implemented in both MOPSO and VOPSO the evasion maneuver is randomly performed, it can either take a starboard turn or a port turn based on the total cost of the maneuver.

Table 2 shows the given results from each of the algorithms. The three other algorithms were local, and used the A* algorithm generated route for path finding. If the algorithm detected an obstacle and the criteria of the local algorithms were met, collision avoidance would be executed. This would generate new waypoints, and thus travel distance and time would be different from the A* algorithm's ideal distance and time.

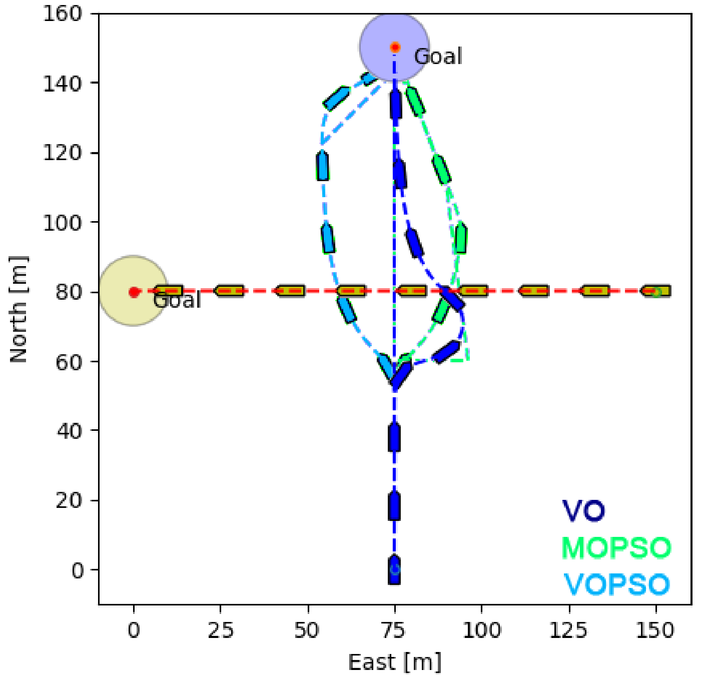


Figure 23: Pass Starboard Test Result

Algorithm	Distance	Min ship distance	Avoidance CPU time	Travel time
A*	142.77	N/A	N/A	51.05
Velocity Obstacle	157.99	19.23	0.015625	60.70
MOPSO	152.28	9.21	3.90625	52.50
VOPSO	159.21	14.62	2.015625	59.3

Table 2: Pass Starboard Test Result

7.1.2 Scenario 2: Ship Head-on

In this scenario as in the first one, the ASV is moving in a straight line from the southern starting point to the northern finishing point. On the way a vessel is approaching from straight ahead in the direction of travel of the ASV.

The ASV then performed a similar evasion maneuver as in Scenario 1 [7.1.1] illustrated in Figure 24. The only difference was that in Scenario 2 [7.1.2] the MOPSO made a turn to the port side, unlike in scenario 1 where it made a starboard turn. The same goes for the VOPSO, where the VOPSO made a port side turn in Scenario 1 [7.1.1].

Table 3 shows the given results from each of the algorithms from the Head-on test.

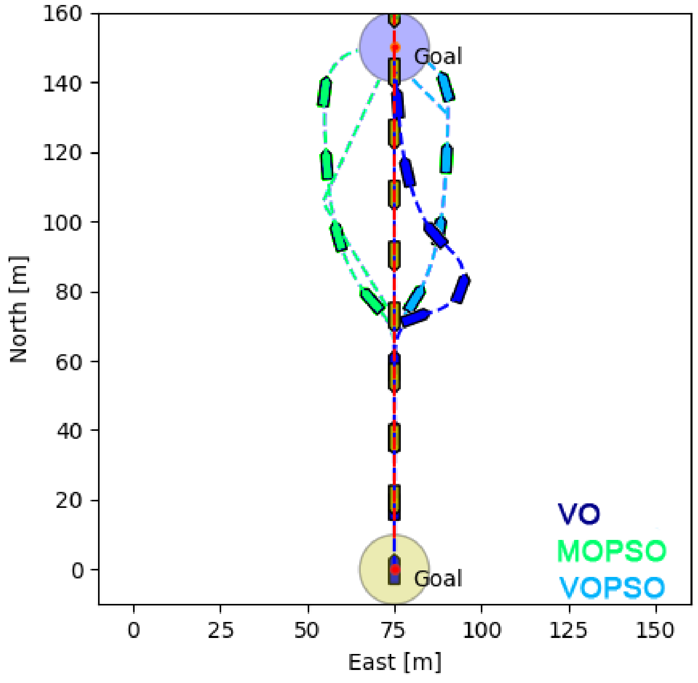


Figure 24: Head-on Test Results

Algorithm	Distance	Min ship distance	Avoidance CPU time	Travel time
A*	142.77	N/A	N/A	51.05
Velocity Obstacle	158.19	14.70	0.015625	60.75
MOPSO	159.77	10.50	3.859375	55.00
VOPSO	153.66	8.83	2.00	55.90

Table 3: Head-on Test Result

7.1.3 Scenario 3: Ship Standstill

In this scenario as in the first and second one, the ASV is moving in a straight line from the southern starting point to the northern finishing point. On the way the ASV is approaching a standstill vessel, on a head-on course.

The ASV then performed a similar evasion maneuver as in Scenario 1 [7.1.1] illustrated in Figure 25. The difference was that both VOPSO and MOPSO now performed a port side turn, while the Velocity Obstacle still performed a starboard turn. They also performed the evasive maneuver earlier than in both Scenario 1 [7.1.1] and Scenario 2 [7.1.2].

Table 3 shows the given results from each of the algorithms from the Head-on standstill test.

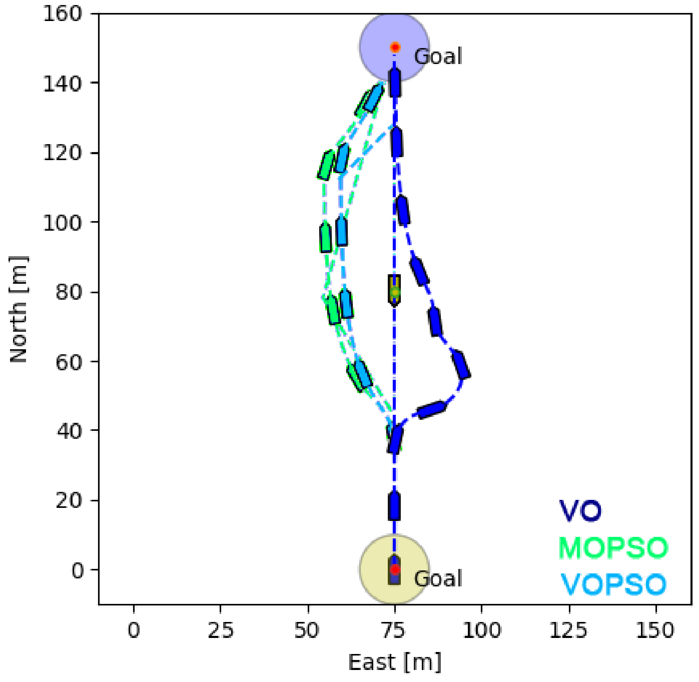


Figure 25: Standstill Test Results

Algorithm	**Distance**	Min ship distance	**Avoidance CPU time**	**Travel time**
A*	142.77	N/A	N/A	51.05
Velocity Obstacle	156.68	8.75	0.03125	64.80
MOPSO	153.48	18.09	4.00	53.35
VOPSO	149.19	14.13	2.296875	53.85

Table 4: Standstill Test Result

8 Discussion

This section covers the evaluation of the results and how to further improve upon the algorithms.

8.1 Project Changes

The project has seen a couple of changes throughout the development process. This is a result of the group getting more educated on the subject which in turn has led to a change in perspective on the project as a whole. Better understanding of the subject led to changes in how the project was approached compared to how it was planned in the Pre-project [C].

8.2 Simulation Results

The most notable result is how much shorter the CPU time of the Velocity Obstacle algorithm was compared to the others. This is explained by the fact that the MOPSO, and by extension the VOPSO, is in its current implementation not running with a convergence value. This means that the algorithm runs through a total of 100 iterations before returning its best value. This is an obvious shortcoming in the implementation of the algorithm. However, we believe that this is not detrimental to the overarching goal of this project, as it still allows us to simulate the behaviour of the algorithm in terms of collision avoidance. We will have to keep in mind that it is not optimized in terms of computation speed.

This does not mean that the time measurements of the algorithms are completely worthless. We are observing that the VOPSO algorithm, which is operating on essentially the same logic as the MOPSO algorithm. This means that VOPSO is also running max iterations for each collision avoidance calculating. With this in mind, we find it significant that in each scenario VOPSO is faster than MOPSO by close to 2 seconds. We attribute this to the fact that the VOPSO algorithm is using fewer geometric checks compared to MOPSO. The most significant of these is the check for the rectangle which represents the forward movement of the vessel, which involves computing 4 triangles in MOPSO compared to the one triangle computed in VOPSO.

The minimum had some varying results which can for the most part be explained by how the algorithms operate. We see VO has the greatest distance in both scenarios where the other vessel is moving, and the smallest distance where the other vessel is standing still. This is directly tied to the method used to select the avoidance velocity. In all scenarios the VO is selecting the largest possible starboard turn, and consequently this gives VO the greatest deviation from the original course of all tested algorithms.

Considering that this greatest deviation occurs simultaneously as the other vessel continues its journey, it makes sense that the distance between the two vessels is the greatest for VO. This maximum turn also explains why the distance is smallest for the stationary obstacle, and it is perfectly illustrated in Figure 25. As the main vessel begins to return to its original path, its velocity vector once again triggers a VO response. We see after this second evasive maneuver, the main vessel returns to its path by passing very close to the obstacle. Since the vessel is not on collision course, this close proximity is not enough to trigger a third VO response. This can be considered a drawback to the geometric nature of the VO algorithm. If we were to add disturbances such as current or wind, the stationary vessel might have enough unpredictable movement to turn this close pass into a potentially dangerous situation. This issue could have been resolved by adding a minimum radius to the collision cone, which would in this scenario have forced a third VO response, and we did in fact account for this in the VOPSO algorithm by adding a high value circle around the second vessel.

Comparing MOPSO and VOPSO, we see that MOPSO has the closest pass proximity in the starboard pass scenario, and VOPSO the closest in the rest. In order to determine the ”danger zones” created by the passing vessel, we had to create some greatly approximated zones, but all of them in the direction of travel for the passing vessel. As we see in the simulation visualization, the main vessel is passing behind the crossing vessel, where there are the least amount of danger zones. Because the rear of the crossing vessel has a minimal amount of high-cost areas, it makes sense the algorithm will choose the closest safe area to the aft of the other vessel. On the other hand, the VOPSO algorithm does not have as many danger zones, but a greater perception of the path of the other vessel. This means the VOPSO will choose a path that focuses on avoidance rather than path fidelity, which explains why VOPSO gives the starboard passing vessel a wider berth.

For the scenarios where VOPSO has a shorter pass distance than MOPSO, we believe it is related to the calculation of relative velocities inherited from the VO algorithm. A common denominator for these two scenarios is that the relative velocity between the vessels ends up in the travel path of the main vessel. This places the edges of the collision cone close to the original path. This means the particle swarm will converge on a safe point close to the original path, with the smallest possible deviation from the goal. It follows that this minimal deviation will result in a close point of passing, and this is corroborated by the fact that in both these scenarios VOPSO also produces the shortest travel path.

Finally we notice some tendencies related to the total travel distance and time. VO produces the longest travel time in all scenarios, but this makes sense considering that it also creates the largest deviation. Further, for scenarios 2 and 3 we see VOPSO produces a shorter travel path, but takes longer to reach the goal. While this difference is generally small, we believe some explanation can be found in the waypoint placement of the two algorithms. Because MOPSO has a larger high-value area in the particle search area, it is forced to make a larger deviation compared to VOPSO. In Figure 24 and Figure 25 we see that the the VOPSO vessel has a almost constant gradual turn while MOPSO has a short period of traveling in a straight line before turning towards the goal. It is possible the MOPSO vessel can attain a higher velocity during this straight travel, which can account for the slight shorter travel time. However, we believe this small difference is not significant for our comparisons.

In summary, we found that VOPSO performs faster than MOPSO by reducing the number of calculations, and plans more efficient routes than VO due to the multi objective evaluation of the particle swarm that check for both maximum collision avoidance and minimum deviation from the main path.

8.3 Further Improvements

Starting off the group had little overall experience with path-finding algorithms, and thus some sub-optimal design decisions were made along the way, this section will cover what aspects we would have done different or further improve given more time to work on the project.

8.3.1 Algorithm

The Velocity Obstacle algorithm was implemented using purely geometry. This means that the avoidance path was decided based on the largest possible turning angle. This could have been done more elegantly by evaluating a selection of possible velocities in a greater range, but the fact remains that there is no inherent method in the algorithm itself to select an optimal course. We found the largest deviation possible would also be the safest, and any further narrowing it down from here would represent a modification of the original algorithm.

The MOPSO algorithm has a couple of obvious flaws, first off COLREG is not properly implemented as the algorithm has equal chance of finding either sides of a oncoming vessel to be the best point. A quick way for better implementation of COLREG could be made by expanding the zones of the cost function to either side of the boat depending on relative angle of the main vessel, this would however have negative impact on the computing time, Which is the second major flaw. The current MOPSO algorithm has no way of checking for convergence except for at the goal, when the cost is at 0. If a convergence check was implemented the algorithm would not need to complete all iterations every time it did a search, but rather stop once the algorithm has reached a convergence point. This check would in term significantly reduce search time by eliminating unnecessary iterations while at the same time retaining the problem solving ability that comes with a higher number of iterations.

The hybrid MOPSO/VO algorithm has also inherited some of the same issues as the MOPSO algorithm in that it does not have any check for convergence. This is not as big a drawback compared to the MOPSO algorithm in that each iteration is faster. Nevertheless the hybrid would also benefit greatly from a convergence check to reduce number of unnecessary iterations which in turn would significantly reduce calculation time.

Another design flaw in the PSO part of the algorithm is that it only searches once when a vessel is encountered. This may lead to issues if the other vessel changes course during the avoidance maneuver, which is also why the random moving vehicle scenario was Sidelined in favor for simpler tests.

8.3.2 Simulator

The kinematic models implemented in the simulator means a correction in course will take some time for the vessel to actuate. This allowed us to gain a more realistic representation of the reaction times for our various algorithms. However, there is still a lot that could be added to make the testing more relevant to real world scenarios. The obvious candidates here are disturbances in motion caused by wind, waves, and current. Despite lacking these elements, we find that the capabilities of the simulator were adequate for the purpose we were employing it.

In the early stages of the project we considered making a digital representation of a specific part of the Norwegian coast, and using AIS data to recreate the traffic through this area for a given time frame. We could then disrupt the ”real” traffic flow, and observe how various algorithms would have handled this disruption. While this was certainly an interesting proposition, it was quickly apparent that this fell outside the scope of our project.

9 Conclusion

We set out to test two collision avoidance algorithms and propose a hybrid algorithm representing a combination of the two. The two chosen algorithms were Velocity Obstacles and Multiple Objective Particle Swarm Optimization, and the hybrid we created from them was named Velocity Obstacle Particle Swarm Optimization. These three algorithms were passed through a series of simple tests to determine the efficiency of this hybrid.

While we found our MOPSO implementation had some design flaws, we decided that they were not detrimental to our analysis of its performance in comparison to the hybrid algorithm. We also wanted to create a seeded random moving ship controller for testing the algorithms in a more unpredictable environment, which worked as planned but due to limitations in the algorithms was scrapped from the final results in favor of simpler scenarios.

Our results showed that by introducing components of Velocity Obstacles to the particle search method of MOPSO we were able to improve performance in areas that were weaknesses in the original algorithms. Velocity Obstacles has no inherent method for selecting a deviation course, and MOPSO requires a large amount of calculation of coordinate values to determine the optimal travel route. The collision cone from Velocity Obstacles allowed us to perform fewer calculations to determine the safest route, which was reflected in the VOPSO requiring less CPU time to calculate the avoidance maneuver. Further, the multiple objective search of MOPSO allowed us to select a collision avoidance path that was both safe and efficient in terms of making the smallest deviation necessary from the original path.

The tests showed that there were some issues in this combination that would warrant further improvements. We see that Velocity Obstacles can create some very short pass distances due to the geometric nature of the algorithm, and this tendency also shows up in the VOPSO tests. The tests also showed positive effects of our algorithm combination in the data extracted, which demonstrated that VOPSO operated faster than MOPSO and with a more efficient avoidance path than Velocity Obstacles.

From this we conclude that while there are some challenges and shortcomings in the results of our project, we still believe that our hybrid algorithm produced some promising results.

References

[1] Peter Hart, Nils Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136. URL: http://ieeexplore.ieee.org/document/4082128/.

[2] IMO. “COLREGS - International Regulations for Preventing Collisions at Sea”. In: *Convention on the International Regulations for Preventing Collisions at Sea, 1972* (1972), pp. 1–74.

[3] Introduction The et al. *Convention on the International Regulations for Preventing Collisions at Sea , 1972 (COLREGs)*. 1977. URL: https://www.imo.org/en/About/Conventions/Pages/COLREG.aspx.

[4] Oussama Khatib. “Real-Time Obstacle Avoidance for Manipulators and Mobile Robots”. In: *Autonomous Robot Vehicles*. New York, NY: Springer New York, 1986, pp. 396–404. DOI: 10.1007/978-1-4613-8997-2{_}29. URL: http://link.springer.com/10.1007/978-1-4613-8997-2_29.

[5] B. Krogh and C. Thorpe. “Integrated path planning and dynamic steering control for autonomous vehicles”. In: *Proceedings. 1986 IEEE International Conference on Robotics and Automation*. Vol. 3. Institute of Electrical and Electronics Engineers, 1986, pp. 1664–1669. ISBN: 0818606959. DOI: 10.1109/ROBOT.1986.1087444. URL: http://ieeexplore.ieee.org/document/1087444/.

[6] Johann Borenstein and Yorem Koren. “Real-Time Obstacle Avoidance for Fast Mobile Robots”. In: *IEEE Transactions on Systems, Man and Cybernetics* 19.5 (Sept. 1989), pp. 1179–1187. ISSN: 21682909. DOI: 10.1109/21.44033. URL: http://ieeexplore.ieee.org/document/44033/.

[7] Franz; Aurenhammer. “Voronoi diagrams—a survey of a fundamental geometric data structure”. In: *ACM Computing Surveys* 23.3 (Sept. 1991), pp. 345–405. ISSN: 0360-0300. DOI: 10.1145/116873.116880. URL: https://dl.acm.org/doi/10.1145/116873.116880.

[8] D. Fox, W. Burgard, and S. Thrun. “The dynamic window approach to collision avoidance”. In: *IEEE Robotics & Automation Magazine* 4.1 (Mar. 1997), pp. 23–33. ISSN: 10709932. DOI: 10.1109/100.580977. URL: http://ieeexplore.ieee.org/document/580977/.

[9] Paolo Fiorini and Zvi Shiller. “Motion planning in dynamic environments using velocity obstacles”. In: *International Journal of Robotics Research* 17.7 (1998), pp. 760–772. ISSN: 02783649. DOI: 10.1177/027836499801700706.

[10] C.A. Coello Coello and M.S. Lechuga. “MOPSO: a proposal for multiple objective particle swarm optimization”. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*. Vol. 2. IEEE, 2002, pp. 1051–1056. ISBN: 0-7803-7282-4. DOI: 10.1109/CEC.2002.1004388. URL: http://ieeexplore.ieee.org/document/1004388/.

[11] Tanja Magoč, Ari Kassin, and Rodrigo Romero. "A line of sight algorithm using fuzzy measures". In: *Annual Conference of the North American Fuzzy Information Processing Society - NAFIPS* (2010). DOI: 10.1109/NAFIPS.2010.554829.

[12] T. Perez and T. I. Fossen. "MSS. Marine Systems Simulator". In: 23/08/2013 (2010). URL: https://github.com/cybergalactic/MSS.

[13] Christensson P. *Algorithm Definition*. 2013. URL: https://techterms.com/definition/algorithm.

[14] Thomas Stenersen. "Guidance Systems for Autonomous Surface Vehicles". PhD thesis. NTNU, 2014, p. 66. DOI: 10.1186/1478-4505-7-15.

[15] Liu Hongdan and Liu Sheng and Yang Zhuo. "Application of Adaptive Wolf Colony Search Algorithm in Ship Collision Avoidance". In: (2015), p. 7.

[16] Thomas Stenersen. "Guidance System for Autonomous Surface Vehicles". In: *123* June (2015), p. 125. ISSN: 0025-5610. URL: https://brage.bibsys.no/xmlui/bitstream/handle/11250/2352498/12747_FULLTEXT.pdf?sequence=1.

[17] Jinfen Zhang et al. "A distributed anti-collision decision support formulation in multi-ship encounter situations under COLREGs". In: *Ocean Engineering* 105 (July 2015), pp. 336–348. ISSN: 00298018. DOI: 10.1016/j.oceaneng.2015.06.054.

[18] Mauro Candeloro, Anastasios M. Lekkas, and Asgeir J. Sørensen. "A Voronoi-diagram-based dynamic path-planning system for underactuated marine vessels". In: *Control Engineering Practice* 61.November 2016 (2017), pp. 41–54. ISSN: 09670661. DOI: 10.1016/j.conengprac.2017.01.007. URL: http://dx.doi.org/10.1016/j.conengprac.2017.01.007.

[19] Yixiong He et al. "Quantitative analysis of COLREG rules and seamanship for autonomous collision avoidance at open sea". In: *Ocean Engineering* 140 (Aug. 2017), pp. 281–291. ISSN: 00298018. DOI: 10.1016/j.oceaneng.2017.05.029.

[20] Xin Wang, Zhengjiang Liu, and Yao Cai. "The ship maneuverability based collision avoidance dynamic support system in close-quarters situation". In: *Ocean Engineering* 146 (Dec. 2017), pp. 486–497. ISSN: 00298018. DOI: 10.1016/j.oceaneng.2017.08.034.

[21] Ørnulf Jan Rødseth. "Definition of autonomy levels for merchant ships". In: August (2018). DOI: 10.13140/RG.2.2.21069.08163.

[22] Luman Zhao and Myung Il Roh. "COLREGs-compliant multiship collision avoidance based on deep reinforcement learning". In: *Ocean Engineering* 191.May (2019), p. 106436. ISSN: 00298018. DOI: 10.1016/j.oceaneng.2019.106436. URL: https://doi.org/10.1016/j.oceaneng.2019.106436.

[23] Jakob Pettermarka. "Kart av Lepsøyrevet". In: (2020). URL: https://www.kartverket.no/globalassets/til-sjos/skolekart/skole_031.pdf.

[24] Red Blog Games. *Grids and Graphs*. 2020. URL: https://www.redblobgames.com/pathfinding/grids/graphs.html.

[25] Sotiris Tzamaras. “Particle Swarm Optimization”. In: (2020). URL: https://github.com/sotostzam/particle-swarm-optimization.

[26] GeeksforGeeks. *A* Search Algoritm*. 2021. URL: https://www.geeksforgeeks.org/a-search-algorithm/.

[27] Renu; Khandelwal. “Particle Swarm Optimization”. In: (2021). URL: https://medium.com/swlh/particle-swarm-optimization-731d9fbb6923.

[28] Amit Patel. *Introduction til Astar*. 2021. URL: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[29] ThinkAutomation. *What is an algorithm? An ’in a nutshell’ explanation*. 2021. URL: https://www.thinkautomation.com/eli5/what-is-an-algorithm-an-in-a-nutshell-explanation/.

[30] Anete Vagale et al. “Path planning and collision avoidance for autonomous surface vehicles I: a review”. In: *Journal of Marine Science and Technology (Japan)* May (2021). ISSN: 09484280. DOI: 10.1007/s00773-020-00787-6.

[31] Wikipedia. *A* Search Algorithm*. 2021. URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

[32] Google. *GoogleMaps*. URL: https://www.google.no/maps/.

[33] James; Kennedy and Russell Eberhart. “Particle swarm optimization”. In: *Proceedings of ICNN’95 - International Conference on Neural Networks*. Vol. 4. IEEE, pp. 1942–1948. ISBN: 0-7803-2768-3. DOI: 10.1109/ICNN.1995.488968. URL: http://ieeexplore.ieee.org/document/488968/.

[34] marinefraffic. *Lepsoyrevet*. URL: https://www.marinetraffic.com/en/ais/home/centerx:6.283/centery:62.593/zoom:14.

[35] MarineTraffic. *MarineTraffic*. URL: https://www.marinetraffic.com/.

[36] Open Robotics. *ROS*. URL: https://www.ros.org/about-ros/.

[37] Open Source Robotics Foundation. *Gazego*. URL: http://gazebosim.org/.

[38] Thomas Skarshaug and Kjetil Vasstein. “Gemini”. In: (). URL: https://github.com/Gemini-team/Gemini.

[39] Thomas Stenersen. “The ASV System Package”. In: (). URL: https://github.com/thomsten/ros_asv_system.

[40] Thomas Stenersen. *The Thesis Simulator*. URL: https://github.com/thomsten/project-thesis-simulator.

[41] Inc. The MathWorks. “matlab toolboxes”. In: (). URL: https://www.mathworks.com/.

[42] USGS. *How Much Water is There on Earth?* URL: https://www.usgs.gov/special-topic/water-science-school/science/how-much-water-there-earth?qt-science_center_objects=0#qt-science_center_objects.

Appendices

A COLREGs

This is an excerpt of the COLREG rules that are relevant to the bachelor thesis. The simulator and algorithms is using these rules when deciding how to avoid collision throughout a simulation.

Rule 8: Action to Avoid Collision

(a) Any action to avoid collision shall be taken in accordance with the Rules of this Part and shall, if the circumstances of the case admit, be positive, made in ample time and with due regard to the observance of good seamanship.

(b) Any alteration of course and/or speed to avoid collision shall, if the circumstances of the case admit, be large enough to be readily apparent to another vessel observing visually or by radar; a succession of small alterations of course and/or speed should be avoided.

(c) If there is sufficient sea-room, alteration of course alone may be the most effective action to avoid a close-quarters situation provided that it is made in good time, is substantial and does not result in another close-quarters situation.

(d) Action taken to avoid collision with another vessel shall be such as to result in passing at a safe distance. The effectiveness of the action shall be carefully checked until the other vessel is finally past and clear.

(e) If necessary to avoid collision or allow more time to assess the situation, a vessel shall slacken her speed or take all way off by stopping or reversing her means of propulsion.

(f) (i) A vessel which, by any of these Rules, is required not to impede the passage or safe passage of another vessel shall, when required by the circumstances of the case, take early action to allow sufficient sea-room for the safe passage of the other vessel.

(ii) A vessel required not to impede the passage or safe passage of another vessel is not relieved of this obligation if approaching the other vessel so as to involve risk of collision and shall, when taking action, have full regard to the action which may be required by the Rules of this part.

(iii) A vessel required not to impede the passage or safe passage of another vessel is not relieved of this obligation if approaching the other vessel so as to involve risk of collision and shall, when taking action, have full regard to the action which may be required by the Rules of this part.

Rule 13: Overtaking

(a) When two power-driven vessels are meeting on reciprocal or nearly reciprocal courses so as to involve risk of collision each shall alter her course to starboard so that each shall pass on the port side of the other.

(b) Such a situation shall be deemed to exist when a vessel sees the other ahead or nearly ahead and by night she could see the masthead lights of the other in a line or nearly in a line and/or both sidelights and by day she observes the corresponding aspect of the other vessel.

(c) When a vessel is in any doubt as to whether such a situation exists she shall assume that it does exist and act accordingly.

Rule 14: Head-on situation

(a) Notwithstanding anything contained in the Rules of part B, sections I and II, any vessel overtaking any other shall keep out of the way of the vessel being overtaken.

(b) A vessel shall be deemed to be overtaking when coming up with another vessel from a direction more than 22.5 degrees abaft her beam, that is, in such a position with reference to the vessel she is overtaking, that at night she would be able to see only the sternlight of that vessel but neither of her sidelights.

(c) When a vessel is in any doubt as to whether she is overtaking another, she shall assume that this is the case and act accordingly.

(d) Any subsequent alteration of the bearing between the two vessels shall not make the overtaking vessel a crossing vessel within the meaning of these Rules or relieve her of the duty of keeping clear of the overtaken vessel until she is finally past and clear.

Rule 15: Crossing situation
When two power-driven vessels are crossing so as to involve risk of collision, the vessel which has the other on her own starboard side shall keep out of the way and shall, if the circumstances of the case admit, avoid crossing ahead of the other vessel.

Rule 16: Action by give-way vessel
Every vessel which is directed to keep out of the way of another vessel shall, so far as possible, take early and substantial action to keep well clear.

Rule 17: Action by stand-on vessel

(a) i Where one of two vessels is to keep out of the way the other shall keep her course and speed.

ii The latter vessel may however take action to avoid collision by her manoeuvre alone, as soon as it becomes apparent to her that the vessel required to keep out of the way is not taking appropriate action in compliance with these Rules.

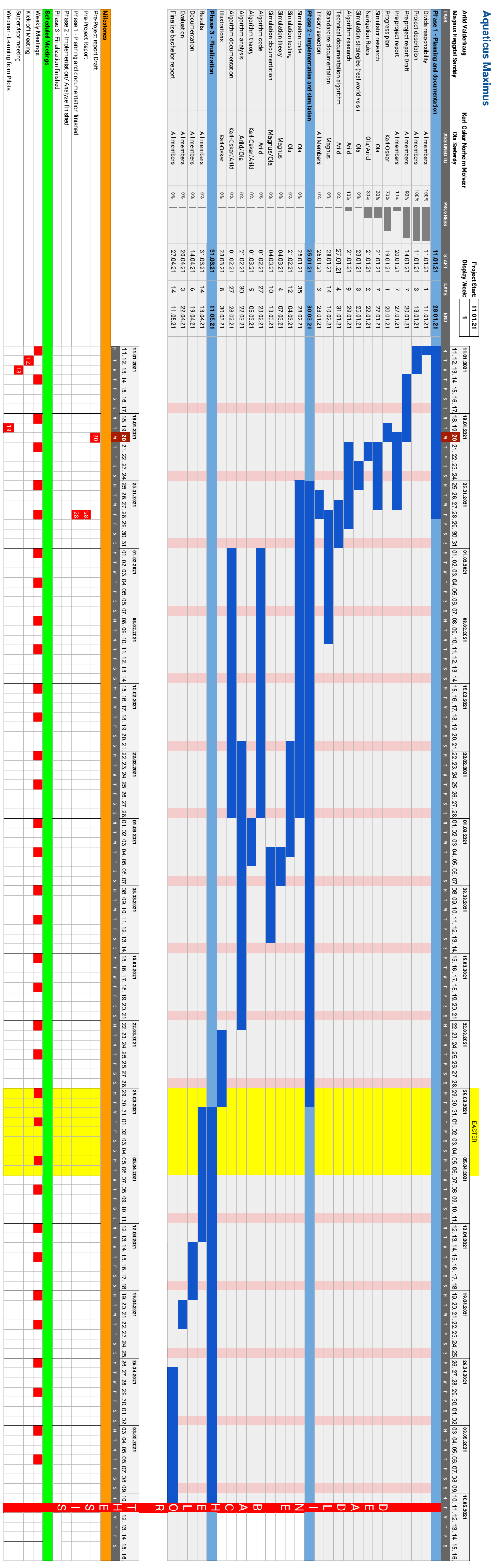
(b) When, from any cause, the vessel required to keep her course and speed finds herself so close that collision cannot be avoided by the action of the give-way vessel alone, she shall take such action as will best aid to avoid collision.

(c) A power-driven vessel which takes action in a crossing situation in accordance with subparagraph (a)(ii) of this Rule to avoid collision with another power-driven vessel shall, if the circumstances of the case admit, not alter course to port for a vessel on her own port side.

(d) This Rule does not relieve the give-way vessel of her obligation to keep out of the way [2].

B Progress Plan

B.1 Progress Plan 20.01.2021



Aquaticus Maximus

Arild Valderhaug — Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy — Ola Sæterøy

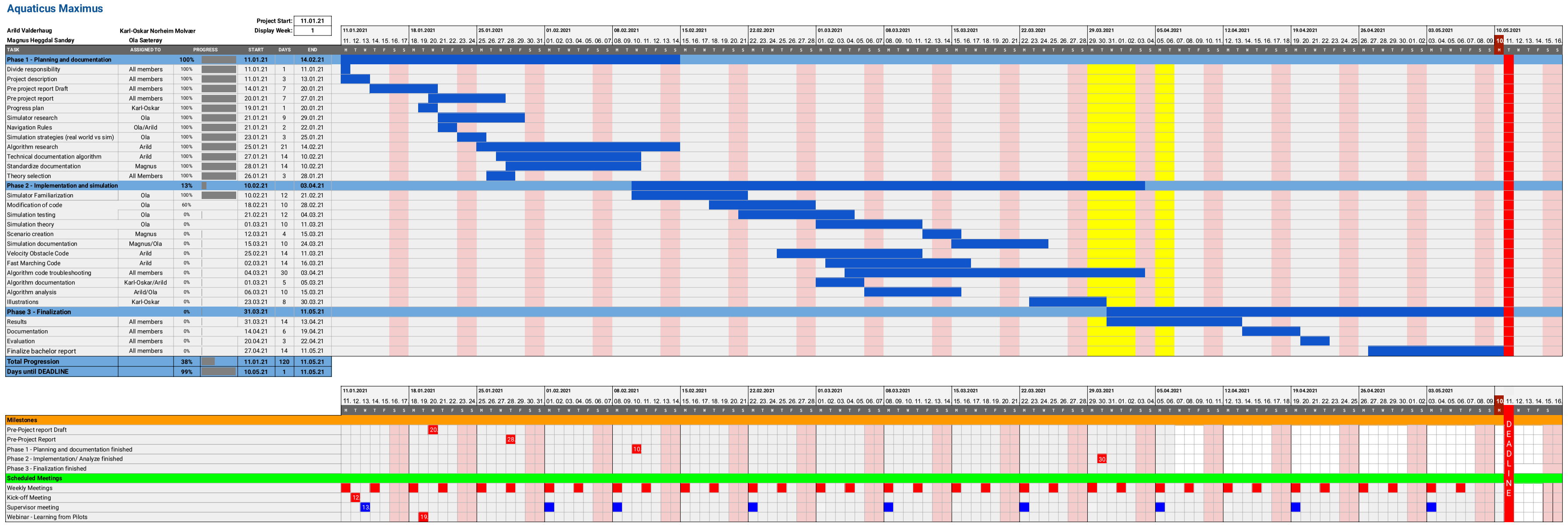
Project Start: 11.01.21
Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	DAYS	END
Phase 1 - Planning and documentation			**11.01.21**		**28.01.21**
Divide responsibility	All members	100%	11.01.21	1	11.01.21
Project description	All members	100%	11.01.21	3	13.01.21
Pre project report Draft	All members	90%	14.01.21	7	20.01.21
Pre project report	All members	10%	20.01.21	7	27.01.21
Progress plan	Karl-Oskar	70%	19.01.21	1	20.01.21
Simulator research	Ola	30%	21.01.21	7	27.01.21
Navigation Rules	Ola/Arild	30%	21.01.21	2	22.01.21
Simulation strategies (real world vs si	Ola	0%	23.01.21	3	25.01.21
Algorithm research	Arild	10%	21.01.21	9	29.01.21
Technical documentation algorithm	Arild	0%	27.01.21	4	31.01.21
Standardize documentation	Magnus	0%	28.01.21	14	10.02.21
Theory selection	All Members	0%	26.01.21	3	28.01.21
Phase 2 - Implementation and simulation			**25.01.21**		**30.03.21**
Simulation code	Ola	0%	25.01.21	35	28.02.21
Simulation testing	Ola	0%	21.02.21	12	04.03.21
Simulation theory	Magnus	0%	04.03.21	4	07.03.21
Simulation documentation	Magnus/Ola	0%	04.03.21	10	13.03.21
Algorithm code	Arild	0%	01.02.21	27	28.02.21
Algorithm theory	Karl-Oskar/Arild	0%	01.03.21	5	05.03.21
Algorithm analysis	Arild/Ola	0%	21.02.21	30	22.03.21
Algorithm documentation	Karl-Oskar/Arild	0%	01.02.21	27	28.02.21
Illustrations	Karl-Oskar	0%	23.03.21	8	30.03.21
Phase 3 - Finalization			**31.03.21**		**11.05.21**
Results	All members	0%	31.03.21	14	13.04.21
Documentation	All members	0%	14.04.21	6	19.04.21
Evaluation	All members	0%	20.04.21	3	22.04.21
Finalize bachelor report	All members	0%	27.04.21	14	11.05.21

Milestones
- Pre-Poject report Draft
- Pre-Project Report
- Phase 1 - Planning and documentation finished
- Phase 2 - Implementation/ Analyze finished
- Phase 3 - Finalization finished

Scheduled Meetings
- Weekly Meetings
- Kick-off Meeting
- Supervisor meeting
- Webinar - Learning from Pilots

B.2 Progress Plan 21.03.2021



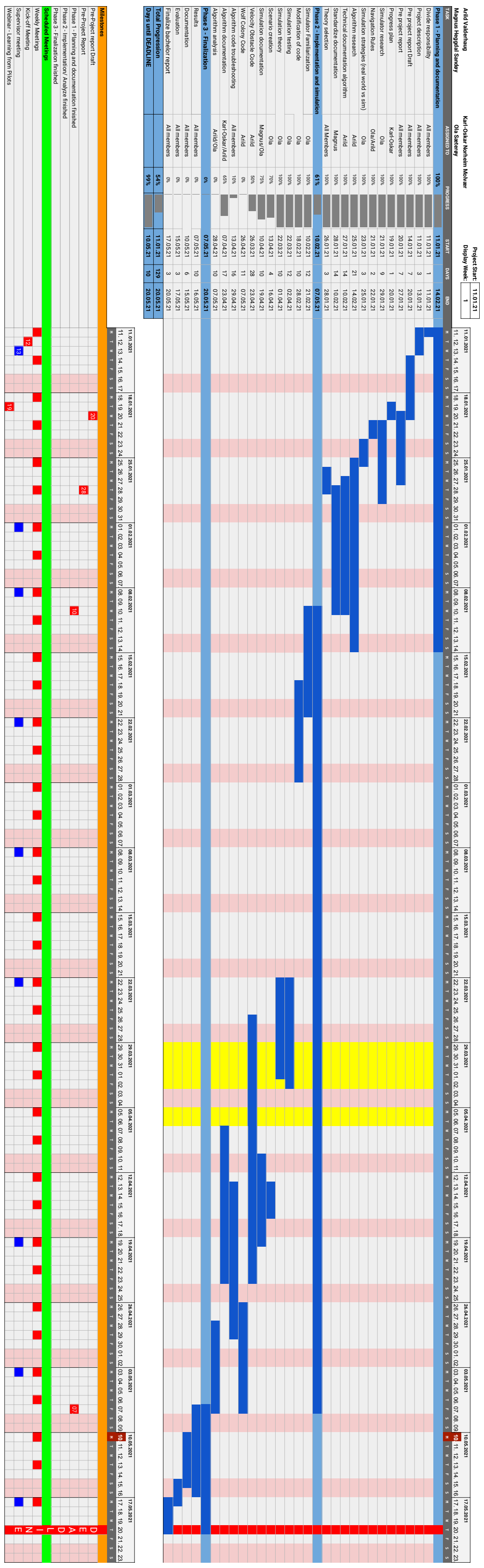
Aquaticus Maximus

Arild Valderhaug	**Karl-Oskar Norheim Molvær**	**Project Start:**	**11.01.21**
Magnus Heggdal Sandøy	**Ola Sæterøy**	**Display Week:**	**1**

TASK	ASSIGNED TO	PROGRESS	START	DAYS	END
Phase 1 - Planning and documentation		**100%**	**11.01.21**		**14.02.21**
Divide responsibility	All members	100%	11.01.21	1	11.01.21
Project description	All members	100%	11.01.21	3	13.01.21
Pre project report Draft	All members	100%	14.01.21	7	20.01.21
Pre project report	All members	100%	20.01.21	7	27.01.21
Progress plan	Karl-Oskar	100%	19.01.21	1	20.01.21
Simulator research	Ola	100%	21.01.21	9	29.01.21
Navigation Rules	Ola/Arild	100%	21.01.21	2	22.01.21
Simulation strategies (real world vs sim)	Ola	100%	23.01.21	3	25.01.21
Algorithm research	Arild	100%	25.01.21	21	14.02.21
Technical documentation algorithm	Arild	100%	27.01.21	14	10.02.21
Standardize documentation	Magnus	100%	28.01.21	14	10.02.21
Theory selection	All Members	100%	26.01.21	3	28.01.21
Phase 2 - Implementation and simulation		**13%**	**10.02.21**		**03.04.21**
Simulator Familiarization	Ola	100%	10.02.21	12	21.02.21
Modification of code	Ola	60%	18.02.21	10	28.02.21
Simulation testing	Ola	0%	21.02.21	12	04.03.21
Simulation theory	Ola	0%	01.03.21	10	11.03.21
Scenario creation	Magnus	0%	12.03.21	4	15.03.21
Simulation documentation	Magnus/Ola	0%	15.03.21	10	24.03.21
Velocity Obstacle Code	Arild	0%	25.02.21	14	11.03.21
Fast Marching Code	Arild	0%	02.03.21	14	16.03.21
Algorithm code troubleshooting	All members	0%	04.03.21	30	03.04.21
Algorithm documentation	Karl-Oskar/Arild	0%	01.03.21	5	05.03.21
Algorithm analysis	Arild/Ola	0%	06.03.21	10	15.03.21
Illustrations	Karl-Oskar	0%	23.03.21	8	30.03.21
Phase 3 - Finalization		**0%**	**31.03.21**		**11.05.21**
Results	All members	0%	31.03.21	14	13.04.21
Documentation	All members	0%	14.04.21	6	19.04.21
Evaluation	All members	0%	20.04.21	3	22.04.21
Finalize bachelor report	All members	0%	27.04.21	14	11.05.21
Total Progression		**38%**	**11.01.21**	**120**	**11.05.21**
Days until DEADLINE		**99%**	**10.05.21**	**1**	**11.05.21**

Milestones	
Pre-Poject report Draft	20
Pre-Project Report	28
Phase 1 - Planning and documentation finished	10
Phase 2 - Implementation/ Analyze finished	30
Phase 3 - Finalization finished	
Scheduled Meetings	
Weekly Meetings	
Kick-off Meeting	12
Supervisor meeting	13
Webinar - Learning from Pilots	19

B.3 Progress Plan 19.04.2021



Arild Valderhaug, Magnus Heggdal Sandøy, Karl-Oskar Norheim Molvær, Ola Sæterøy

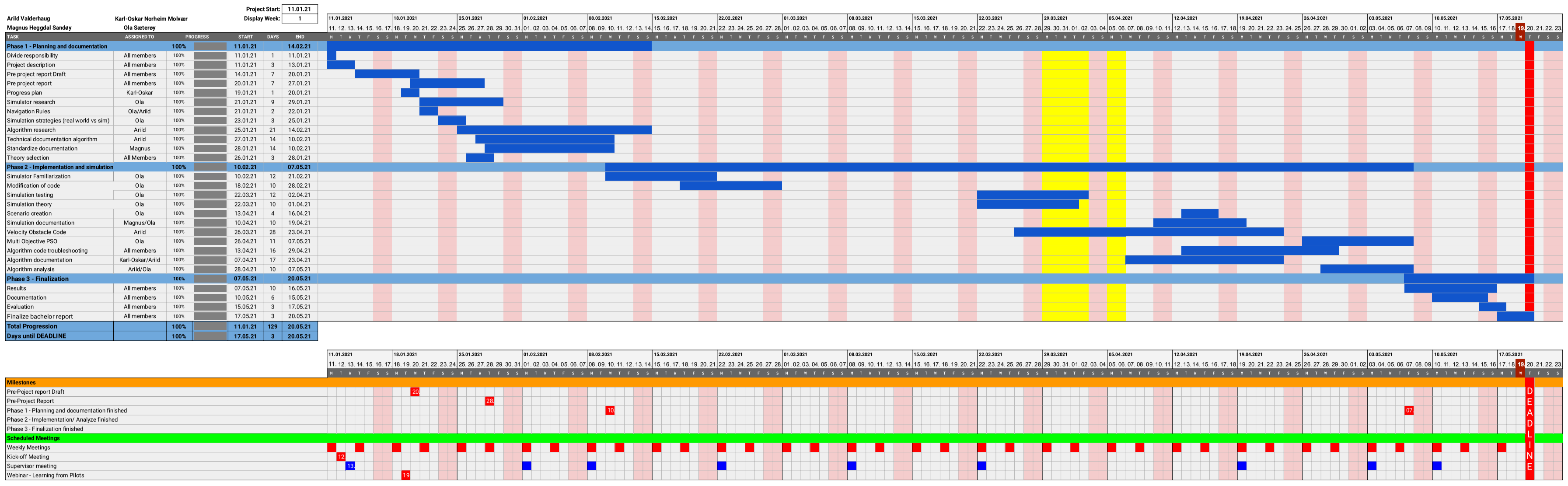
Project Start: 11.01.21

Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	DAYS	END
Phase 1 - Planning and documentation		**100%**	**11.01.21**		**14.02.21**
Divide responsibility	All members	100%	11.01.21	1	11.01.21
Project description	All members	100%	11.01.21	3	13.01.21
Pre project report Draft	All members	100%	14.01.21	7	20.01.21
Pre project report	All members	100%	20.01.21	7	27.01.21
Progress plan	Karl-Oskar	100%	19.01.21	1	20.01.21
Simulator research	Ola	100%	21.01.21	9	29.01.21
Navigation Rules	Ola/Arild	100%	21.01.21	2	22.01.21
Simulation strategies (real world vs sim)	Ola	100%	23.01.21	3	25.01.21
Algorithm research	Arild	100%	25.01.21	21	14.02.21
Technical documentation algorithm	Arild	100%	27.01.21	14	10.02.21
Standardize documentation	Magnus	100%	28.01.21	14	10.02.21
Theory selection	All Members	100%	26.01.21	3	28.01.21
Phase 2 - Implementation and simulation		**61%**	**10.02.21**		**07.05.21**
Simulator Familiarization	Ola	100%	10.02.21	12	21.02.21
Modification of code	Ola	100%	18.02.21	10	28.02.21
Simulation testing	Ola	100%	22.03.21	12	02.04.21
Simulation theory	Ola	100%	22.03.21	10	01.04.21
Scenario creation	Ola	70%	13.04.21	4	16.04.21
Simulation documentation	Magnus/Ola	75%	10.04.21	10	19.04.21
Velocity Obstacle Code	Arild	50%	26.03.21	28	23.04.21
Wolf Colony Code	Arild	0%	26.04.21	11	07.05.21
Algorithm code troubleshooting	All members	10%	13.04.21	16	29.04.21
Algorithm documentation	Karl-Oskar/Arild	65%	07.04.21	17	23.04.21
Algorithm analysis	Arild/Ola	0%	28.04.21	10	07.05.21
Phase 3 - Finalization		**0%**	**07.05.21**		**20.05.21**
Results	All members	0%	07.05.21	10	16.05.21
Documentation	All members	0%	10.05.21	6	15.05.21
Evaluation	All members	0%	15.05.21	3	17.05.21
Finalize bachelor report	All members	0%	17.05.21	3	20.05.21
Total Progression		**54%**	**11.01.21**	**129**	**20.05.21**
Days until DEADLINE		**99%**	**10.05.21**	**10**	**20.05.21**

Milestones
Pre-Poject report Draft
Pre-Poject Report
Phase 1 - Planning and documentation finished
Phase 2 - Implementation/ Analyze finished
Phase 3 - Finalization finished
Scheduled Meetings
Weekly Meetings
Kick-off Meeting
Supervisor meeting
Webinar - Learning from Pilots

B.4 Progress Plan 19.05.2021



Arild Valderhaug | Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy | Ola Sæterøy

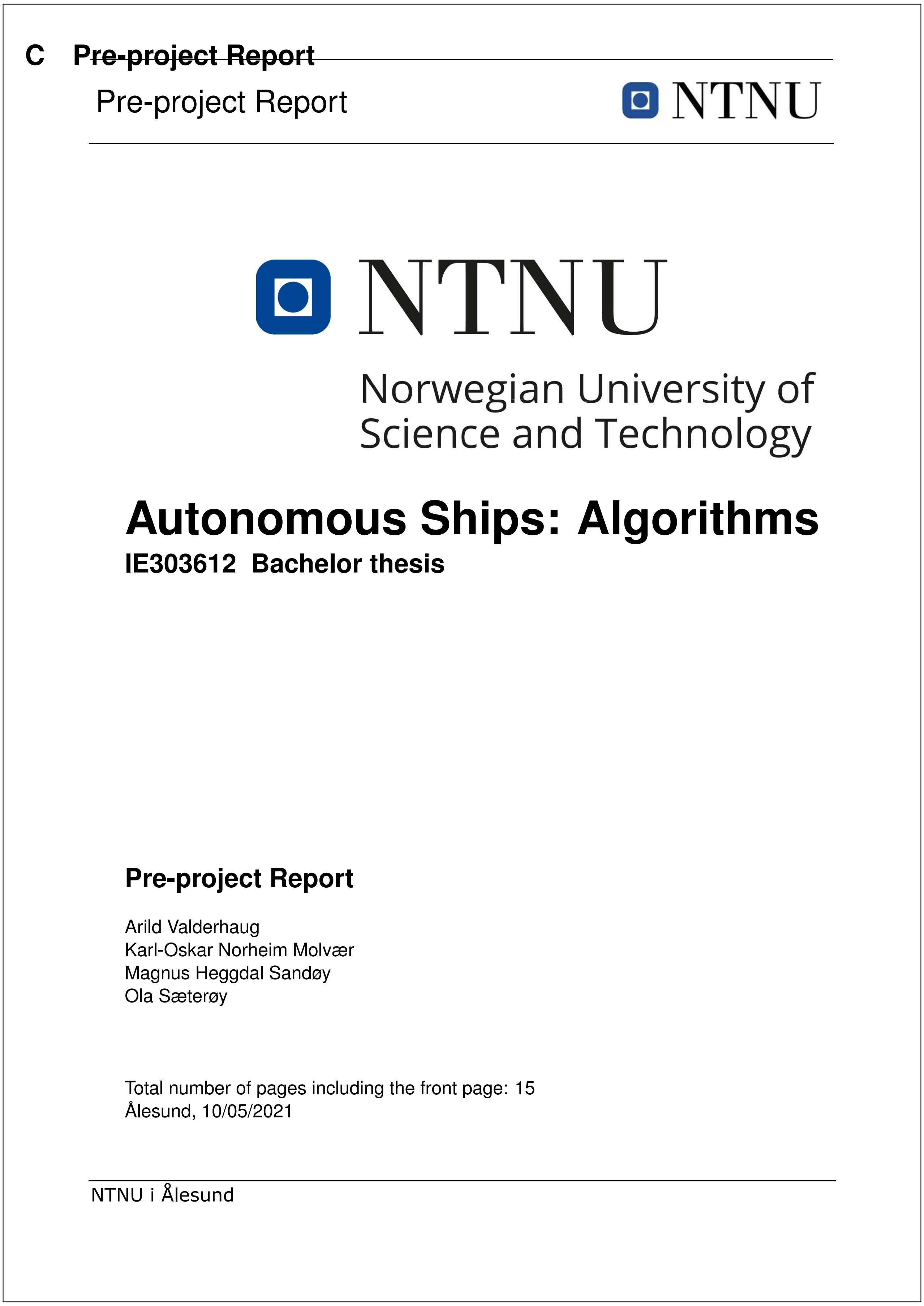
Project Start: 11.01.21
Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	DAYS	END
Phase 1 - Planning and documentation		**100%**	**11.01.21**		**14.02.21**
Divide responsibility	All members	100%	11.01.21	1	11.01.21
Project description	All members	100%	11.01.21	3	13.01.21
Pre project report Draft	All members	100%	14.01.21	7	20.01.21
Pre project report	All members	100%	20.01.21	7	27.01.21
Progress plan	Karl-Oskar	100%	19.01.21	1	20.01.21
Simulator research	Ola	100%	21.01.21	9	29.01.21
Navigation Rules	Ola/Arild	100%	21.01.21	2	22.01.21
Simulation strategies (real world vs sim)	Ola	100%	23.01.21	3	25.01.21
Algorithm research	Arild	100%	25.01.21	21	14.02.21
Technical documentation algorithm	Arild	100%	27.01.21	14	10.02.21
Standardize documentation	Magnus	100%	28.01.21	14	10.02.21
Theory selection	All Members	100%	26.01.21	3	28.01.21
Phase 2 - Implementation and simulation		**100%**	**10.02.21**		**07.05.21**
Simulator Familiarization	Ola	100%	10.02.21	12	21.02.21
Modification of code	Ola	100%	18.02.21	10	28.02.21
Simulation testing	Ola	100%	22.03.21	12	02.04.21
Simulation theory	Ola	100%	22.03.21	10	01.04.21
Scenario creation	Ola	100%	13.04.21	4	16.04.21
Simulation documentation	Magnus/Ola	100%	10.04.21	10	19.04.21
Velocity Obstacle Code	Arild	100%	26.03.21	28	23.04.21
Multi Objective PSO	Ola	100%	26.04.21	11	07.05.21
Algorithm code troubleshooting	All members	100%	13.04.21	16	29.04.21
Algorithm documentation	Karl-Oskar/Arild	100%	07.04.21	17	23.04.21
Algorithm analysis	Arild/Ola	100%	28.04.21	10	07.05.21
Phase 3 - Finalization		**100%**	**07.05.21**		**20.05.21**
Results	All members	100%	07.05.21	10	16.05.21
Documentation	All members	100%	10.05.21	6	15.05.21
Evaluation	All members	100%	15.05.21	3	17.05.21
Finalize bachelor report	All members	100%	17.05.21	3	20.05.21
Total Progression		**100%**	**11.01.21**	**129**	**20.05.21**
Days until DEADLINE		**100%**	**17.05.21**	**3**	**20.05.21**

Milestones
- Pre-Poject report Draft
- Pre-Project Report
- Phase 1 - Planning and documentation finished
- Phase 2 - Implementation/ Analyze finished
- Phase 3 - Finalization finished

Scheduled Meetings
- Weekly Meetings
- Kick-off Meeting
- Supervisor meeting
- Webinar - Learning from Pilots

C Pre-project Report



Norwegian University of Science and Technology

Autonomous Ships: Algorithms

IE303612 Bachelor thesis

Pre-project Report

Arild Valderhaug
Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy
Ola Sæterøy

Total number of pages including the front page: 15
Ålesund, 10/05/2021

NTNU i Ålesund

NTNU ÅLESUND
PRE-PROJECT REPORT

Tittel:
Autonomous Ships: Algorithms

Candidate number (name):
Arild Valderhaug
Karl-Oskar Norheim Molvær
Magnus Heggdal Sandøy
Ola Sæterøy

Date: 10/05/2021	Sub. Code: IE303612	Subject Name: Bachelor thesis	Document Access:

Study Program: Automatiseringsteknikk	Nr. pages/Attachments: 15 /	Bibl. nr:

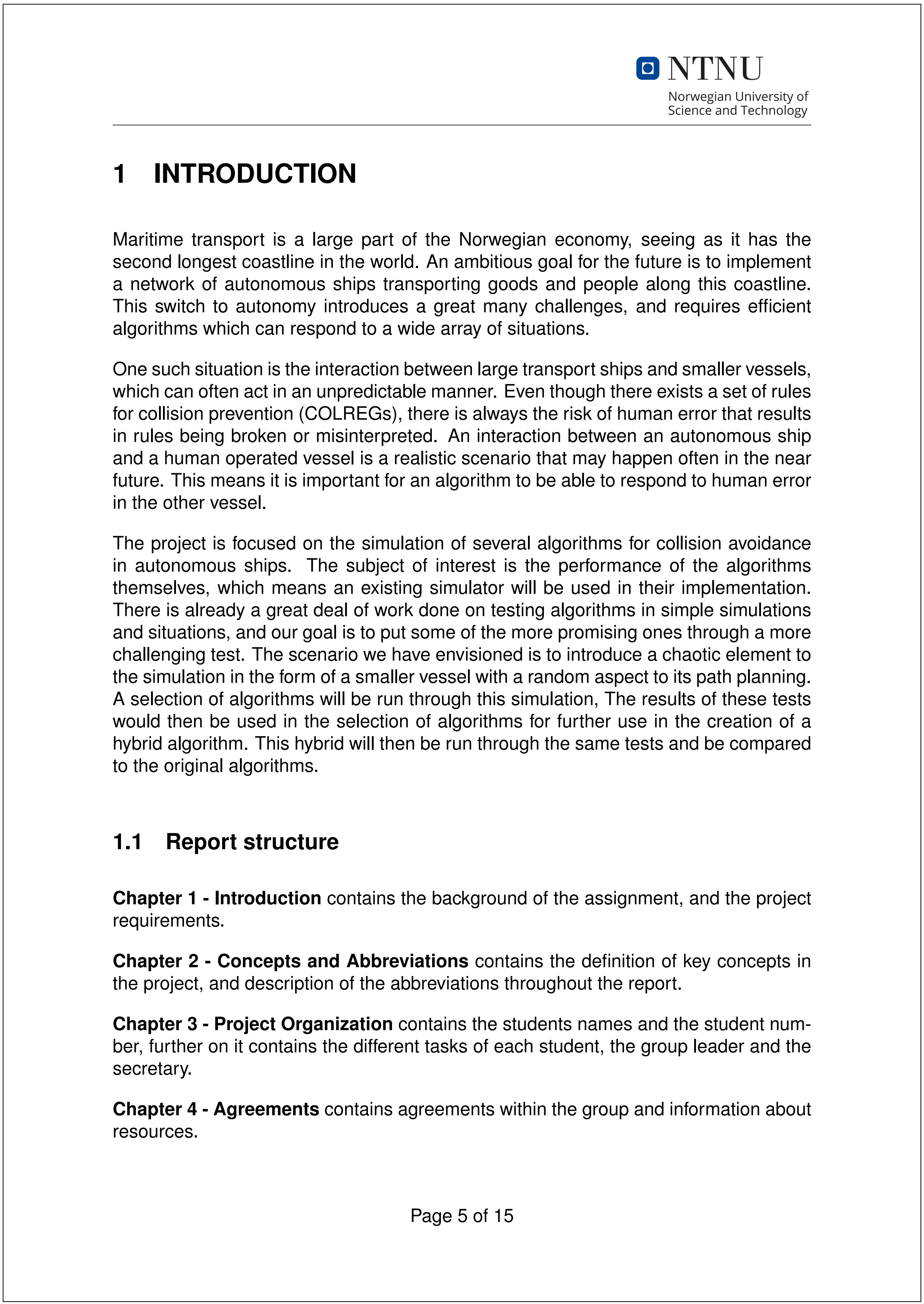
Advisers:
Anete Vagale
Aleksander Skrede
Robin T. Bye
Ottar L. Osen

Summary:

This pre-project report contains information regarding our bachelor thesis on algorithms for autonomous ships. It defines the project structure and description, as well as the group organization with the area of responsibility for each member. The goal of the project is to test several algorithms in simulated situations with unpredictable vessels, and to create a hybrid algorithm which aims to perform better in unpredictable situations than its predecessors. The report will also define scheduled meetings and agreements within the group.

Contents

List of Tables



1 INTRODUCTION

Maritime transport is a large part of the Norwegian economy, seeing as it has the second longest coastline in the world. An ambitious goal for the future is to implement a network of autonomous ships transporting goods and people along this coastline. This switch to autonomy introduces a great many challenges, and requires efficient algorithms which can respond to a wide array of situations.

One such situation is the interaction between large transport ships and smaller vessels, which can often act in an unpredictable manner. Even though there exists a set of rules for collision prevention (COLREGs), there is always the risk of human error that results in rules being broken or misinterpreted. An interaction between an autonomous ship and a human operated vessel is a realistic scenario that may happen often in the near future. This means it is important for an algorithm to be able to respond to human error in the other vessel.

The project is focused on the simulation of several algorithms for collision avoidance in autonomous ships. The subject of interest is the performance of the algorithms themselves, which means an existing simulator will be used in their implementation. There is already a great deal of work done on testing algorithms in simple simulations and situations, and our goal is to put some of the more promising ones through a more challenging test. The scenario we have envisioned is to introduce a chaotic element to the simulation in the form of a smaller vessel with a random aspect to its path planning. A selection of algorithms will be run through this simulation, The results of these tests would then be used in the selection of algorithms for further use in the creation of a hybrid algorithm. This hybrid will then be run through the same tests and be compared to the original algorithms.

1.1 Report structure

Chapter 1 - Introduction contains the background of the assignment, and the project requirements.

Chapter 2 - Concepts and Abbreviations contains the definition of key concepts in the project, and description of the abbreviations throughout the report.

Chapter 3 - Project Organization contains the students names and the student number, further on it contains the different tasks of each student, the group leader and the secretary.

Chapter 4 - Agreements contains agreements within the group and information about resources.

Page 5 of 15

Chapter 5 - Project Description contains the definition of the assignment, requirements for the solution, planned approach and the progress plan.

Chapter 6 - Documentation contains the reports and technical documents.

Chapter 7 - Scheduled Meetings and Reports contains scheduled meeting dates, meeting reports and periodical reports.

Chapter 8 - Planned Deviation Treatment contains the procedures if the assignment do not go as planned.

Chapter 9 - Equipment Requirements/ Conditions for Implementation contains the equipment and software we need to complete the bachelor thesis.

2 CONCEPTS AND ABBREVIATIONS

2.1 Concepts

Algorithms is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation

Simulation is an approximate imitation of a process or system representing its operation over time

COLREG is the equivalent of traffic rules for boats

AIS Radio detection system used for collision avoidance

ASV Boats or ships that operate on the surface of water without a crew

MATLAB A numeric computing environment which allows execution of complex mathematical operations, including algorithms.

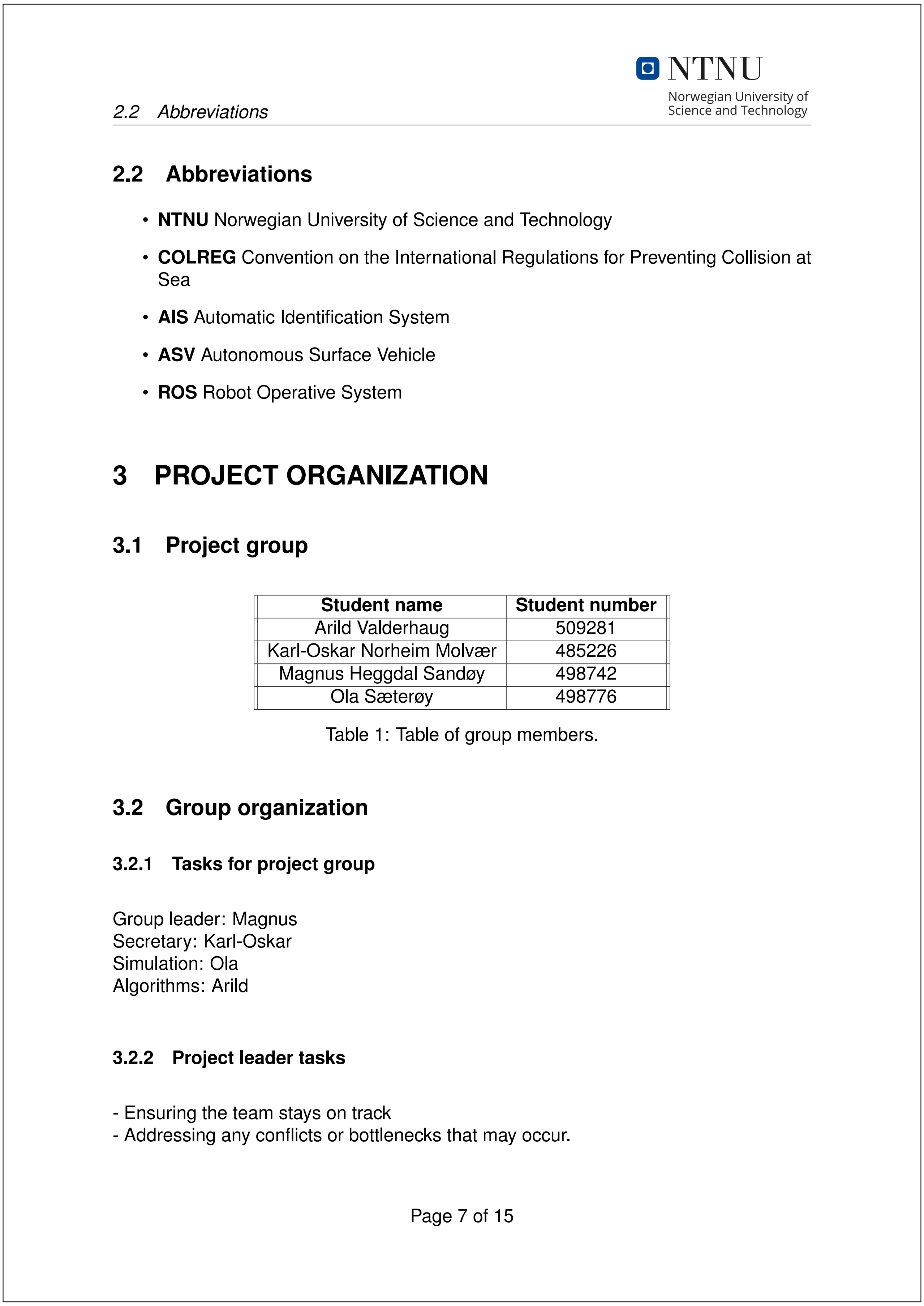
PyCharm An IDE developed by Jetbrains for writing programs in Python.

ReSharper An IDE developed by Jetbrains for writing programs in C++.

Hyper-V Software for Virtual machine servers

Ubuntu Linux based PC Operative system

ROS An open source collection of frameworks for robot software development.



2.2 Abbreviations

- **NTNU** Norwegian University of Science and Technology
- **COLREG** Convention on the International Regulations for Preventing Collision at Sea
- **AIS** Automatic Identification System
- **ASV** Autonomous Surface Vehicle
- **ROS** Robot Operative System

3 PROJECT ORGANIZATION

3.1 Project group

Student name	Student number
Arild Valderhaug	509281
Karl-Oskar Norheim Molvær	485226
Magnus Heggdal Sandøy	498742
Ola Sæterøy	498776

Table 1: Table of group members.

3.2 Group organization

3.2.1 Tasks for project group

Group leader: Magnus
Secretary: Karl-Oskar
Simulation: Ola
Algorithms: Arild

3.2.2 Project leader tasks

- Ensuring the team stays on track
- Addressing any conflicts or bottlenecks that may occur.

3.2.3 Secretary tasks

- Log and follow up meetings
- Maintain bibliography
- Responsible of scheduling meetings
- Responsible for providing the supervisors with the weekly reports

3.2.4 Other members

-Algorithms: Gain knowledge of existing algorithms and relevant research, responsible for correct implementation
-Simulation: Ensure simulation program runs with correct parameters, including correct kinematics and code flow.

3.3 Supervisors

Main Supervisor	Anete Vagale
Co-supervisor	Aleksander Skrede
Resource persons	Robin T. Bye, Ottar L. Osen

Table 2: Table of supervisors

4 AGREEMENTS

4.1 Workspace and resources

Working from home will be the primary workspace. Labs at NTNU are also available, provided Covid-19 guidelines are followed, but for now the primary workspace remains at home.

The group will have access to all NTNU software licenses.

The group will communicate with supervisor(s) primarily through e-mail and Zoom meetings. Regularly scheduled meetings bi-weekly.

4.2 Group standards - Cooperation rules - Attitudes

Due to the Covid-19 situation, the group intends to primarily work from home. This brings some challenges, including maintaining standards for communication and cooperation.

We will not require common work hours, but allow a more individual approach. To ensure everyone stays on track, we will have daily meetings verbally to summarize the progress of the day, and any challenges going forward. The purpose of this loose structure is to help alleviate some of the pressure associated with working from home, and allow each member to manage their own workday. The experience from previous projects together has shown that the group members have a peak productivity period at different times of the day. If this method should appear to work poorly, we will consider a more structured approach.

The final product will be a reflection of our common work-ethic and integrity as soon-to-be engineers. It is therefore important to strive to produce a high-quality product, and not take shortcuts. Furthermore it is important to acknowledge that we can't know everything, and to reach out for assistance within the group before the problems become insurmountable.

5 PROJECT DESCRIPTION

5.1 Problem - Objective - Purpose

The project is focused on simulating at least two algorithms for path planning and collision avoidance in autonomous ships. The goal is to test each algorithms reaction to a randomly moving element in the travel path, and score their performance by a scoring system to be developed. Based on this data a hybrid algorithm will be created which will attempt to handle chaotic elements better. Ideally the findings of this project will be able to contribute to the current research being done on algorithms for autonomous ships.

5.2 Requirements for solution or project results - Specification

The final product will have a collection of algorithms with their acquired score, which will be defined during the main project. We will have saved results from each simulation, as well as visualizations to help illustrate the performance of each algorithm. There will also be a discussion as to why each algorithm was selected for the project.

The final product will also have a hybrid algorithm with its associated test score. The design goal for the hybrid algorithm is to respond to the random element in the simulator better than any individual algorithm, and thus achieve a better score.

5.3 Planned procedure for development work - Method

Identification of problem and information
Systematical gathering of information in relationship to the given task. A basic foundation must be built with information associated with algorithm, legal resources, simulators and simulator criteria.

Goals and objectives
After a basic information foundation is established we will produce a chart with specific objectives and measurable accomplishments to be achieved within a given time period. This provides the group a clear goal to work towards.

Monitoring and Evaluation
Continual communication and a good work ethic from every member of the bachelor group, is a must to reaching the final goal. Regularly progress evaluation to ensure effective use of time and resources such as personnel, and reduce wasted time.

Project management method
The group intend to utilise ”Agile” to be able to fast response on changes throughout the bachelor project. By organising every goals into small tasks with a given time frame, so the group can respond to changes made by limitations or unpredictable events.

5.4 Information collection - Performed and planned

The project consists of two distinct parts which each require gathering of information before starting in full; which simulators are available, and which algorithms are candidates to use in the project.

We already have a handful of simulators that we have evaluated. Marine Systems Simulator is developed by Thor I. Fossen at NTNU Trondheim. Another is Robot Operating System, which allows for a plethora of plugins. One such that we have found comes from a master’s thesis from 2015 written for the Cybernetics and Robotics program in Trondheim [1]. While it implements some algorithms which are still potential candidates for our own project, or primarily interest in this paper is reusing the simulator with some minor modifications.

When it comes to algorithms we will be using the article by some members of our own supervisor team as a starting point [2], and will spend some time gathering supplementary theory on those we find interesting for our own project. To this end we will include a period of information collection in our project plan.

5.5 Assessment - Risk analysis

The project depends on mainly two hinges: first that we successfully implement the simulator of our choice, and second that we successfully integrate the chosen algorithms into the simulator. Both of these elements are depending on our ability to both manipulate existing code, and to recreate algorithms in our chosen programming language. This raises a certain risk that if we become too bogged down in the programming aspect, we will have little to show for many hours of work. We can certainly attempt to alleviate this risk by reaching out for assistance sooner than later, but in the beginning we have to at least acknowledge that the risk is there.

There is also the risk of elevated restrictions associated with Covid-19. However, our primary workspace is from home, and as such any increase in such restrictions will have minimal impact on our project.

5.6 Main activities in further work

The project will be divided into 3 steps.

1. Information gathering
2. Implementation and simulation
3. Finalization

5.7 Progress plan - Management of the project

Phase1: Planning and documentation

Phase 1 - Information gathering
Collect data to build a foundation for the practical work. Working with simulators and algorithm need a basic understanding to construct a good product. As the group does not have any prior experience working with simulators and algorithm.

Goal of Phase 1:
- Decide on a simulator for implementing of algorithm.
- Install necessary software to get simulator up and running.
- Produced standardise documentation (minute of meeting, work log etc.)

Phase2: Implementation and simulation
The main focus in phase 2 is implementation of algorithms into a simulator.Write necessary code, testing, run simulations and collect data. Reasons for decisions and result must be documented trough this phase by all group members.

Goal of Phase 2:
- Have a working simulator
- Implementation of algorithms into the simulator
- Be able to utilise simulator to produce scenarios to push algorithms to its limits
- Documentation of decisions, reasons and result through the process of phase 2.

Phase3: Finalization
The last phase focus is merging all the data and theory we have collected, to finalize our bachelor report, and reflection of the final product.

Management tools We will be using a Gantt-chart as our primary time management tool. It allows us to visualize all individual parts of the project along with their deadlines and the group member responsible for its completion. The Gantt-chart is attached at the end of this document.

Development tools We will be using software licenses available through our NTNU e-mail addresses to complete the project. A complete list can be found in part 9 of this report.

As this project is entirely digital, no physical tools will be required for its completion.

Internal control and evaluation
A combination between work log and results will be utilised to evaluate each groups members work effort. If one of the members are not able to produce any results and have a lot of wasted time, a reevaluation of the work must be done, to ensure the project continues in the right direction.

5.8 Decisions - Decision process

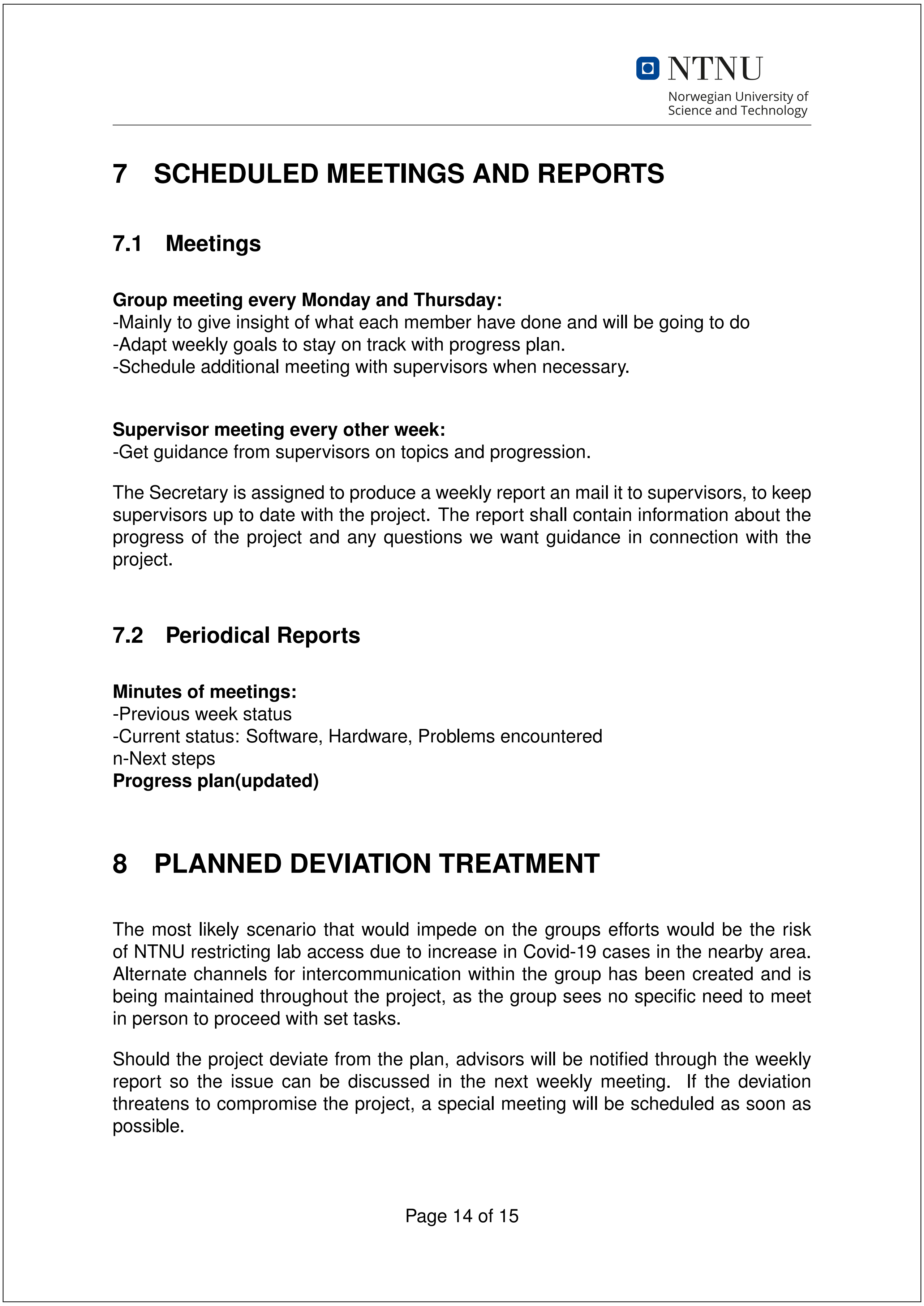
All decision that have a big impact towards progression of the bachelor are made trough a diplomatic possess between every group member. The main reason for this is to get input and response from every group member to build the best possible final result.

If a group member is assigned to a main task of the project, than that person have authority to do minor changes without getting input from the other group members, as long as the decision do not affect other parts of the project. However, it is important to inform the rest of the group of the change in the following meeting.

6 DOCUMENTATION

6.1 Reports and technical documents

- Gantt: prepared to keep track of basic work tasks and goals
- Minutes of meetings: Documentation of every meeting to retain information and appointments in case someone is missing from a meeting or become ill.



7 SCHEDULED MEETINGS AND REPORTS

7.1 Meetings

Group meeting every Monday and Thursday:
-Mainly to give insight of what each member have done and will be going to do
-Adapt weekly goals to stay on track with progress plan.
-Schedule additional meeting with supervisors when necessary.

Supervisor meeting every other week:
-Get guidance from supervisors on topics and progression.

The Secretary is assigned to produce a weekly report an mail it to supervisors, to keep supervisors up to date with the project. The report shall contain information about the progress of the project and any questions we want guidance in connection with the project.

7.2 Periodical Reports

Minutes of meetings:
-Previous week status
-Current status: Software, Hardware, Problems encountered
n-Next steps
Progress plan(updated)

8 PLANNED DEVIATION TREATMENT

The most likely scenario that would impede on the groups efforts would be the risk of NTNU restricting lab access due to increase in Covid-19 cases in the nearby area. Alternate channels for intercommunication within the group has been created and is being maintained throughout the project, as the group sees no specific need to meet in person to proceed with set tasks.

Should the project deviate from the plan, advisors will be notified through the weekly report so the issue can be discussed in the next weekly meeting. If the deviation threatens to compromise the project, a special meeting will be scheduled as soon as possible.

9 EQUIPMENT REQUIREMENTS/ CONDITIONS FOR IMPLEMENTATION

We will be using the following software licenses afforded through registering with our student e-mail:

- MATLAB
- PyCharm
- ReSharper
- Hyper-V
- Unity

We will also be using Robot Operating System, which is an open-source software.

Per our current understanding we believe that the project will not be requiring any special resources from NTNU.

10 REFERENCES

1. T. Stenersen, ”Guidance Systems for Autonomous Surface Vehicles,” M.S. thesis, Dept. Engineering Cybernetics, NTNU Trondheim, 2015.
2. A. Vagale, R. Oucheikh, R.T. Bye, O.L. Osen, T.I. Fossen, ”Path Planning and Collision Avoidance for Autonomous Surface Vehicles II: A Comparative Study of Algorithms”, J. Marine Science and Technology, 2020.

D External Links

This is the complete Source Code of the Bachelor Thesis ASV simulator `https://github.com/Frostmort/Bachelor-ASV-Simulator`, which is based on the ASV simulator `https://github.com/thomsten/project-thesis-simulator` created by Thomas Stenersen in 2015 in connection with his master's thesis.

Following is the video presentation of the project `https://drive.google.com/file/d/1W7fB7CXR0Wonmzm1uFiIZk-xtX-DZxyH/view?usp=sharing`.

E Source Code

E.1 Random Moving Vessel Class

```python
#!/usr/bin/env python
import numpy as np
import random

from utils import Controller


class Wafi(Controller):
    def __init__(self, R2=60, mode='wafi', seed='rand', turnfreq=40):


        self.cGoal = None  # Current Goal
        self.cWP = 0  # Used if waypoint-navigation
        self.nWP = 0
        self.is_initialized = False

        self.goaly = 0
        self.goalx = 0

        self.seed = seed
        self.iterator = 1
        self.rng = random
        self.R2 = R2
        self.mode = mode
        self.stepcounter = 0
        self.turnfreq = turnfreq

        self.wps = None

    def update(self, vobj, world, vesselArray):

        if not self.is_initialized:
            # Reference to the vessel object's waypoints
            self.wps = vobj.waypoints

            #initial setup for wafi, set seed and get starting goal
            if self.mode == 'wafi':
                if self.seed == 'rand':
                    self.seed = random.randrange(0, 10000, 1)
                self.rng.seed(self.seed)
                print('Seed:', self.seed)

                self.new_goal(vobj)
            self.is_initialized = True

        x = vobj.x[0]
        y = vobj.x[1]

        if self.mode == 'waypoint' or self.mode == 'pursuit':
            vobj.psi_d = np.arctan2(self.cGoal[1] + x,
                                    self.cGoal[0] + y)






        if self.mode == 'wafi':
            vobj.psi_d = np.arctan2(self.goalx -y,
                                    self.goaly - x)

            self.stepcounter = self.stepcounter + 1
#            print('step:',self.stepcounter)
            if self.stepcounter >= self.turnfreq:
                self.new_goal(vobj)
                self.stepcounter = 0



    def draw(self, axes, N, fcolor, ecolor):
        axes.plot(self.wps[:, 0], self.wps[:, 1], 'k--')

    def visualize(self, fig, axarr, t, n):
        if self.mode == 'goal-switcher' or self.mode == 'waypoint':
            axarr[0].plot(self.wps[:, 0], self.wps[:, 1], 'k--')
            axarr[0].plot(self.cGoal[0], self.cGoal[1], 'rx', ms=10)

    def random_waypoint(self):
        x = self.rng.randrange(-50, 50, 1)
        y = self.rng.randrange(-50, 50, 1)

        print('new random numbers x:',x,'y:',y)

        return (x, y)

    def new_goal(self,vobj):
        x1, y1 = self.random_waypoint()
        list = np.copy(vobj.x)
        x2, y2 = list[0:2]
        self.goalx= x1 + x2
        self.goaly= y1 + y2


        print('set new goal', self.goaly,',',self.goalx)



if __name__ == "__main__":
    vessel = Wafi()
    vessel.rng.seed(1337)
    for x in range(1, 11):
        x1 = vessel.random_waypoint(x1=(0, 0))
        print("new waypoints:", x1)

```

E.2 Velocity Obstacle Class

```python
import copy
import time


import numpy as np

from vessel import Vessel
from utils import Controller, PriorityQueue

from matplotlib2tikz import save as tikz_save


class VO(Controller):
    def __init__(self, scanDistance=50):
        self.scanDistance = scanDistance
        self.tc = 0
        self.world = 0
        self.newVesselParams = [0, 0]
        self.totalTime = 0

    def update(self, vobj, world, vesselArray):
        self.world = world
        tic = time.process_time_ns()
        for v in vesselArray:
            if not v.is_main_vessel:
                scanData = self.scan(vobj, v)
                if scanData[0] <= self.scanDistance:
                    VOarray = self.createVO(vobj, v, scanData)
                    if VOarray[3] > VOarray[8] > VOarray[4]:
                        print("Collision imminent!")
                        self.newVesselParams = self.collisionAvoidance(vobj, v, scanData)
                        vobj.u_d = self.newVesselParams[0]
                        vobj.psi_d = self.newVesselParams[1]
                        self.totalTime = self.totalTime + (time.process_time_ns() - tic)
                        print(self.totalTime)


    def scan(self, vessel1, vessel2):
        xd = (vessel2.x[0] - vessel1.x[0])
        yd = (vessel2.x[1] - vessel1.x[1])
        distance = abs(np.sqrt(xd ** 2 + yd ** 2))
        angle = np.arctan2(yd, xd)

        return [distance, angle]

    # Creates the VO array for use in collision detection and
    # Array has following contents:
    # [0 crossing direction, 1 distance between ships, 2 angle between ships, 3 left
collision cone edge,
    # 4 right collision cone edge, 5 velocity of A, 6 velocity of B, 7 relative velocity
magnitude,
    # 8 relative velocity angle]
    def createVO(self, vessel1, vessel2, scanData):
        VO = [0, 0, 0, 0, 0, 0, 0, 0, 0]
        # find which side crossing vessel is coming from
        if vessel2.x[0] > vessel1.x[0] and (np.pi / 2 < vessel2.x[2] < 3 * np.pi / 2):
            VO[0] = 'r'
        elif vessel2.x[0] < vessel1.x[0] and (vessel2.x[2] < np.pi / 2 or vessel2.x[2] >
3 * np.pi / 2):
            VO[0] = 'l'
        else:
            VO[0] = 'n'

        # find left and right boundaries of collision cone
        VO[1] = scanData[0]
        VO[2] = scanData[1]
        angle = np.arctan2(scanData[0] / 2, scanData[0])

        VO[3] = VO[2] + np.arctan2((scanData[0] / 2) + 5, scanData[0])
        VO[4] = VO[2] - np.arctan2((scanData[0] / 2) + 5, scanData[0])

        # find vector (xab) and angle (lab) of relative velocity
        VO[5] = [np.cos(vessel1.x[2]), np.sin(vessel1.x[2])]
        VO[6] = [(np.cos(vessel2.x[2])), (np.sin(vessel2.x[2]))]
        VO[7] = [VO[5][0] - VO[6][0], VO[5][1] - VO[6][1]]
        VO[8] = np.arctan2(VO[7][1], VO[7][0])

        return VO



    def collisionAvoidance(self, v1, v2, scanData):

        xyc = [0, 0]
        self.tc = self.getCollisionTime(v1, v2, xyc)
        RV = self.getRV(v1)
        newParams = self.getRAV(v1, v2, RV, scanData)
        return newParams

    def getCollisionTime(self, v1, v2, xyc):
        r1 = ([v1.x[0], v1.x[3] * np.cos(v1.x[2]), v1.x[1], v1.x[3] * np.sin(v1.x[2
    ])])  # vessel 1 velocity vector
        r2 = ([v2.x[0], v2.x[3] * np.cos(v2.x[2]), v2.x[1], v2.x[3] * np.sin(v2.x[2
    ])])  # vessel 2 velocity vector

        tx = (r2[0] - r1[0]) / (r1[1] - r2[1]) # time to intersect in x
        ty = (r2[2] - r1[2]) / (r1[3] - r2[3]) # time to intersect in y

        return (tx + ty) / 2 # returns average of x and y times

    def getRV(self, v1):
        u_max = v1.model.est_u_max # max surge velocity
        u_min = v1.model.est_u_min # min surge velocity (reverse)
        r_max = v1.model.est_r_max # max yaw velocity

        du_max = v1.model.est_du_max # max surge acceleration
        du_min = v1.model.est_du_min # min surge acceleration (reverse)
        dr_max = v1.model.est_dr_max # max yaw acceleration

        t = 1

        rt = dr_max * t
        ut = du_max * t

        if rt > r_max:
            rt = r_max

        if ut > u_max:
            ut = u_max

        maxstraight = ut
        maxreverse = -ut
        maxstarboard = [ut * np.cos(rt), ut * np.sin(rt)]
        maxport = [-1 * maxstarboard[0], maxstarboard[1]]

        return [maxstraight, maxreverse, maxstarboard, maxport]

    def getRAV(self, v1, v2, RV, scanData):
        testVessel = Vessel(copy.deepcopy(v1.x), np.zeros((1, 6)), v1.h, v1.dT, v1.N
    , [], False, 'viknes')
        testVessel.world = copy.deepcopy(self.world)

        testVessel.x = copy.deepcopy(v1.x)

        print('test starboard')
        testVessel.x[3] = np.sqrt(RV[2][0]**2 + RV[2][1]**2)
        testVessel.x[2] = testVessel.x[2] - (np.pi/2 - np.arctan2(RV[2][1], RV[2][0]))
        testVO = self.createVO(testVessel, v2, scanData)
        if self.checkNewVO(testVO):
            newParams = [testVessel.x[3], testVessel.x[2]]
            return newParams

        print('test port')
        testVessel.x[3] = np.sqrt(RV[3][0]**2 + RV[3][1]**2)
        testVessel.x[2] = testVessel.x[2] - (np.pi/2 - np.arctan2(RV[3][1],RV[3][0]))
        testVO = self.createVO(testVessel, v2, scanData)
        if self.checkNewVO(testVO):
            newParams = [testVessel.x[3], testVessel.x[2]]
            return newParams

        print('test ahead')
        testVessel.x[3] = RV[0]
        testVessel.x[2] = v1.x[2]
        testVO = self.createVO(testVessel, v2, scanData)
        if self.checkNewVO(testVO):
            newParams = [testVessel.x[3], testVessel.x[2]]
            return newParams

        print('test reverse starboard')
        testVessel.x[3] = RV[1]
        testVessel.x[2] = testVessel.x[2] - (np.pi/2 - np.arctan2(RV[2][1], RV[2][0]))
        testVO = self.createVO(testVessel, v2, scanData)
        if self.checkNewVO(testVO):
            newParams = [testVessel.x[3], testVessel.x[2]]
            return newParams

        print('test reverse port')
        testVessel.x[3] = RV[1]
        testVessel.x[2] = testVessel.x[2] - (np.pi/2 - np.arctan2(RV[3][1],RV[3][0]))
        testVO = self.createVO(testVessel, v2, scanData)
        if self.checkNewVO(testVO):
            newParams = [testVessel.x[3], testVessel.x[2]]
            return newParams

        print('test reverse')
        testVessel.x[3] = RV[1]
        testVessel.x[2] = v1.x[2]
        testVO = self.createVO(testVessel, v2, scanData)
        if self.checkNewVO(testVO):
            newParams = [testVessel.x[3], testVessel.x[2]]
            return newParams

        print('No RAV found')
        return [0, v1.x[2]]

    def checkNewVO(self, VO):
        if not VO[3] > VO[8] > VO[4]:
            return True
        else:
            return False

    def checkLand(self, vessel1, params):
        vessel1.u_d = params[0]
        vessel1.psi_d = params[1]
        for x in range(self.world.n, round(self.world.n + (self.tc * 100))):
            vessel1.update_model(x)
            p0 = vessel1.model.x[0:2]
            if self.world._map.is_occupied(p0, safety_region=False):
                return True

        return False

```

E.3 Multi Object Particle Swarm Optimization Class

```python
import sys, time
import heapq

import random
import numpy as np
import matplotlib.pyplot as plt

from map import Map
from utils import Controller
from vessel import Vessel

DIMENSIONS = 2  # Number of dimensions
GLOBAL_BEST = 0  # Global Best of Cost function
MIN_RANGE = 0  # Lower boundary of search space
MAX_RANGE = 50 # Upper boundary of search space
POPULATION = 50  # Number of particles in the swarm
V_MAX = 1 # Maximum velocity value
PERSONAL_C = 2.0  # Personal coefficient factor
SOCIAL_C = 2.0  # Social coefficient factor
CONVERGENCE = 0  # Convergence value
MAX_ITER = 100  # Maximum number of iterations
BIGVAL = 10000.
MINDIST = 20


class Mopso(Controller):
    def __init__(self, x0, xg, the_map, search_radius=50, replan=False):
        self.start = x0[0:3]
        self.goal = xg[0:3]
        self.scanRadius = search_radius
        self.world = None
        self.grid_size = the_map.get_dimension
        self.graph = SearchGrid(the_map, [1.0, 1.0, 25.0/360.0])
        self.map = the_map
        self.to_be_updated = True
        self.replan = replan
        self.path_found = False
        self.wpUpdated = False
        self.currentcWP = 0

        self.totalTime = 0

        self.particles = []  # List of particles in the swarm
        self.best_pos = None  # Best particle of the swarm
        self.best_pos_z = np.inf  # Best particle of the swarm

    def update(self, vobj, world, vesselArray):
        tic = time.process_time_ns()
        if len(vesselArray) > 1:
            v2 = vesselArray[1]
            scanData = self.scan(vobj.x[0:2], v2.x[0:2])
            if scanData[0] <= self.scanRadius and not self.wpUpdated:
                self.currentcWP = vobj.controllers[1].cWP
                nextWP = self.search(vobj.x[0:2], vesselArray, scanData)
                print("Vessel 1: ", vobj.x[0:2])
                print("Vessel 2: ", v2.x[0:2])
                for x in range(0, 3):
                    print("Waypoint ", self.currentcWP + x, ": ", nextWP)
                    vobj.controllers[1].wp = np.insert(vobj.waypoints, self.currentcWP +
x, nextWP, axis = 0)
                    vobj.waypoints = np.insert(vobj.waypoints, self.currentcWP + x,
nextWP, axis = 0)
                    scanData = self.scan(nextWP, v2.x[0:2])
                    nextWP = self.search(nextWP, vesselArray, scanData)
                self.wpUpdated = True
                self.totalTime = self.totalTime + (time.process_time_ns() - tic)
                print(self.totalTime)

    def search(self, vobjx, vesselArray, scanData):

        # Initialize swarm
        x0 = vobjx[0:2]
        print("Sverm0 ", x0)

        localMin = [vobjx[0] - MAX_RANGE, vobjx[1] - MAX_RANGE]
        localMax = [vobjx[0] + MAX_RANGE, vobjx[1] + MAX_RANGE]

        swarm = Swarm(POPULATION, V_MAX, self.goal, x0, vesselArray, scanData)
        # Initialize inertia weight
        inertia_weight = 0.5 + (np.random.rand() / 2)
        curr_iter=0
        for i in range(MAX_ITER):

            for particle in swarm.particles:

                for i in range(0, DIMENSIONS):
                    r1 = np.random.uniform(0, 1)
                    r2 = np.random.uniform(0, 1)

                    # Update particle's velocity
                    personal_coefficient = PERSONAL_C * r1 * (particle.best_pos[i] - particle.pos[i])
                    social_coefficient = SOCIAL_C * r2 * (swarm.best_pos[i] - particle.pos[i])
                    new_velocity = inertia_weight * particle.velocity[i] + personal_coefficient + social_coefficient

                    # Check if velocity is exceeded
                    if new_velocity > V_MAX:
                        particle.velocity[i] = V_MAX
                    elif new_velocity < -V_MAX:
                        particle.velocity[i] = -V_MAX
                    else:
                        particle.velocity[i] = new_velocity

                # Update particle's current position
                particle.pos += particle.velocity

                particle.pos_z = swarm.cost_function(particle.pos[0], particle.pos[1], vesselArray)

                # Update particle's best known position
                if particle.pos_z < swarm.cost_function(particle.best_pos[0], particle.best_pos[1], vesselArray):
                    particle.best_pos = particle.pos.copy()

                    # Update swarm's best known position
                    if particle.pos_z < swarm.best_pos_z:
                        swarm.best_pos = particle.pos.copy()
                        swarm.best_pos_z = particle.pos_z

                # # Check if particle is within boundaries
                biggest = 0
                hypeCheck = np.hypot(particle.pos[0] - x0[0], particle.pos[1] - x0[1])
                if hypeCheck > biggest:
                    biggest = hypeCheck
                if np.hypot(particle.pos[0] - x0[0], particle.pos[1] - x0[1]) > MAX_RANGE:
                    r = np.random.uniform(MIN_RANGE, MAX_RANGE)
                    theta = np.random.uniform(0, 2 * np.pi)
                    particle.pos[0] = (r * np.cos(theta)) + x0[0]
                    particle.pos[1] = (r * np.sin(theta)) + x0[1]


            # Check for convergence
            if abs(swarm.best_pos_z - GLOBAL_BEST) < CONVERGENCE:
                print("The swarm has met convergence criteria after " + str(curr_iter) + " iterations.", 'at:',swarm.best_pos)
                break
            curr_iter += 1

        if abs(swarm.best_pos_z - GLOBAL_BEST) > CONVERGENCE:
            print("The swarm has reached max iterations after " + str(curr_iter) + "
iterations.", 'at:',
                  swarm.best_pos)
        print("Swarm: ", swarm.best_pos)
        print("Biggest: ", biggest)
        return [swarm.best_pos[0], swarm.best_pos[1]]




    def scan(self, vessel1, vessel2):
        xd = (vessel2[0] - vessel1[0])
        yd = (vessel2[1] - vessel1[1])
        distance = abs(np.sqrt(xd**2 + yd**2))
        angle = np.arctan2(yd, xd)

        return [distance, angle]


#########################################################################################
##################
class Swarm():
    def __init__(self, pop, v_max, goal, x0, vesselArray, scanData):
        self.particles = []         # List of particles in the swarm
        self.best_pos = None        # Best particle of the swarm
        self.best_pos_z = np.inf    # Best particle of the swarm
        self.x0 = x0                #ship pos
        self.goal = goal
        self.scanData = scanData
        self.vesselArray = vesselArray
        for _ in range(pop):
            r = np.random.uniform(MIN_RANGE, MAX_RANGE)
            theta = np.random.uniform(MIN_RANGE, MAX_RANGE*np.pi)
            x = (r * np.cos(theta))+x0[0]
            y = (r * np.sin(theta))+x0[1]


            z = self.cost_function(x, y, goal)
            velocity = np.random.rand(2) * v_max
            particle = Particle(x, y, z, velocity)
            self.particles.append(particle)
            if self.best_pos != None and particle.pos_z < self.best_pos_z:
                self.best_pos = particle.pos.copy()
                self.best_pos_z = particle.pos_z
            else:
                self.best_pos = particle.pos.copy()
                self.best_pos_z = particle.pos_z


    def cost_function(self, x1, y1, goal):
        devW = 1
        statW = 1
        dynW = 1
        weighingMatrix = np.array([[devW], [statW], [dynW]])
        pos = x1,y1
        x2,y2=self.goal[0],self.goal[1]


        deviation_cost = (np.sqrt((x2-x1)**2 + (y2-y1)**2))   #distance from goal

        statitc_obs_cost = 0
        # if not self.graph.passable(pos):                    #Check if static obstacle
        #     statitc_obs_cost= BIGVAL

        distance = np.hypot(x1 - self.vesselArray[1].x[0], y1 - self.vesselArray[1].x[1
])    #check for dynamic obstacle
        if distance <= 5:
            dyn_obs_cost = BIGVAL
        elif 5 < distance <= 10:
            dyn_obs_cost = 100
        # elif 10 < distance <= 20:
        #     dyn_obs_cost = 50
        else:
            dyn_obs_cost = 0

        if self.is_inside(self.get_dangercone(self.vesselArray[1]), pos):
            dyn_obs_cost = BIGVAL
        if self.is_inside2(self.get_dangercube(self.vesselArray[1]), pos):
            dyn_obs_cost = BIGVAL

        if dyn_obs_cost < 0:
            dyn_obs_cost = 0

        cost = np.sum(np.array([[deviation_cost], [statitc_obs_cost], [dyn_obs_cost
]]) * weighingMatrix)
        return cost

    def is_inside(self, triangle, pos):           #check if point is inside cone
        x1 = triangle[0]
        x2 = triangle[1]
        x3 = triangle[2]
        xp = pos

        c1 = (x2[0]-x1[0]) * (xp[1]-x1[1]) - (x2[1]-x1[1]) * (xp[0]-x1[0])
        c2 = (x3[0]-x2[0]) * (xp[1]-x2[1]) - (x3[1]-x2[1]) * (xp[0]-x2[0])
        c3 = (x1[0]-x3[0]) * (xp[1]-x3[1]) - (x1[1]-x3[1]) * (xp[0]-x3[0])
        if (c1 < 0 and c2 < 0 and c3 < 0) or (c1 > 0 and c2 > 0 and c3 > 0):
            return True
        else:
            return False

    def is_inside2(self,square,pos):
        xp = pos
        x1 = square[0]
        x2 = square[1]
        x3 = square[2]
        x4 = square[3]

            #x1,x2 and xp
        area1 = np.abs((x1[0]*x2[1] + x2[0]*xp[1] + xp[0]*x1[1]) - (x1[1]*x2[0] + x2[1]*
xp[0] + xp[1]*x1[0]))*0.5

            #x2,x3 and xp
        area2 = np.abs((x3[0]*x2[1] + x2[0]*xp[1] + xp[0]*x3[1]) - (x3[1]*x2[0] + x2[1]*
xp[0] + xp[1]*x3[0]))*0.5

            #x3,x4 and xp
        area3 = np.abs((x3[0]*x4[1] + x4[0]*xp[1] + xp[0]*x3[1]) - (x3[1]*x4[0] + x4[1]*
xp[0] + xp[1]*x3[0]))*0.5

            #x1,x4 and xp
        area4 = np.abs((x1[0]*x4[1] + x4[0]*xp[1] + xp[0]*x1[1]) - (x1[1]*x4[0] + x4[1]*
xp[0] + xp[1]*x1[0]))*0.5

        areasquare = np.hypot(x1[0] - x2[0], x1[1] - x2[1]) * np.hypot(x1[0] - x3[0], x1
[1] - x3[1])

        if np.sum([area1,area2,area3,area4]) <= areasquare:
            return True
        else:
            return False
    def get_dangercone(self, vobj):
        phi = 2.02
        l = BIGVAL
        p0 = [vobj.x[0],vobj.x[1]]
        p1 = [((l*np.cos(vobj.x[2] - phi/2)) + vobj.x[0]), ((l*np.sin(vobj.x[2] - phi/2
) + vobj.x[1]))]
        p2 = [((l*np.cos(vobj.x[2] + phi/2)) + vobj.x[0]), ((l*np.sin(vobj.x[2] + phi/2
) + vobj.x[1]))]

        # print("Trækant: ", p0, p1, p2)

        return [p0, p1, p2]

    def get_dangercube(self, vobj):
        l = 2 * MAX_RANGE
        p0 = [MINDIST * np.cos(vobj.x[2] - np.pi/2) + vobj.x[0], MINDIST * np.sin(vobj.x
[2] - np.pi/2) + vobj.x[1]]
        p1 = [MINDIST * np.cos(vobj.x[2] + np.pi/2) + vobj.x[0], MINDIST * np.sin(vobj.x
[2] + np.pi/2) + vobj.x[1]]
        p2 = [l * np.cos(vobj.x[2]) + p0[0], l * np.sin(vobj.x[2]) + p0[1]]
        p3 = [l * np.cos(vobj.x[2]) + p1[0], l * np.sin(vobj.x[2]) + p1[1]]

        #print("Square: ", p0, p1, p2, p3)

        return [p0, p1, p2, p3]

# Particle class
class Particle():
    def __init__(self, x, y, z, velocity):
        self.pos = [x, y]
        self.pos_z = z
        self.velocity = velocity
        self.best_pos = self.pos.copy()

class SearchGrid(object):
    """General purpose N-dimentional search grid."""
    def __init__(self, the_map, gridsize, N=2, parent=None):
        self.N        = N
        self.grid     = the_map.get_discrete_grid()
        self.map      = the_map
        self.gridsize = gridsize
        self.gridsize[0] = the_map.get_gridsize()
        self.gridsize[1] = the_map.get_gridsize()
        dim = the_map.get_dimension()

        self.width  = dim[0]
        self.height = dim[1]

        """
        In the discrete map, an obstacle has the value '1'.
        We multiply the array by a big number such that the
        grid may be used as a costmap.
        """
        self.grid *= BIGVAL


    def get_grid_id(self, state):
        """Returns a tuple (x,y,psi) with grid positions."""
        return (int(state[0]/self.gridsize[0]),
                int(state[1]/self.gridsize[1]),
                int(state[2]/self.gridsize[2]))

    def in_bounds(self, state):
        return 0 <= state[0] < self.width and 0 <= state[1] < self.height

    def cost(self, a, b):
        #if b[0] > self.width or b[1] > self.height:
        #    return 0
        return np.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)#self.grid[int(b[0]/self.gridsize
[0]), int(b[1]/self.gridsize[1])]

    def passable(self, state):
        # TODO Rename or change? Only returns true if object is _inside_ obstacle
        # Polygons add safety zone by default now.

        if state[0] > self.width or state[1] > self.height:
            return True

        return self.grid[int(state[0]/self.gridsize[0]),
                         int(state[1]/self.gridsize[1])] < BIGVAL

    def neighbors(self, state, est_r_max):
        """
        Applies rudder commands to find the neighbors of the given state.

        For the Viknes 830, the maximum rudder deflection is 15 deg.
        """
        step_length = 2.5*self.gridsize[0]
        avg_u       = 3.5
        Radius      = 2.5*avg_u / est_r_max
        dTheta      = step_length / Radius
        #print(Radius, dTheta*180/np.pi)

        trajectories = np.array([[step_length*np.cos(dTheta), step_length*np.sin(dTheta
    ), dTheta],
                                 [step_length, 0.,  0.],
                                 [step_length*np.cos(dTheta), -step_length*np.sin(dTheta
    ), -dTheta]])

        #print(trajectories)
        results = []
        for traj in trajectories:
            newpoint = state + np.dot(Rz(state[2]), traj)
            if self.passable(newpoint):
                results.append(newpoint)

        #results = filter(self.in_bounds, results)

        return results



if __name__ == "__main__":
    mymap = Map("s1", gridsize=1.0, safety_region_length=4.5)

    x0 = np.array([0, 0, np.pi / 2, 3.0, 0.0, 0])
    xg = np.array([30, 86, np.pi / 4])
    mopso = Mopso(x0, xg, mymap)
    myvessel = Vessel(x0, xg, 0.05, 0.5, 1, [mopso], True, 'viknes')
    vesselArray = [myvessel]


    mopso.update(myvessel, vesselArray)

    fig = plt.figure()
    ax = fig.add_subplot(111, autoscale_on=False)
    #   ax.plot(myvessel.waypoints[:, 0],
    #           myvessel.waypoints[:, 1],
    #           '-')

    # ax.plot(x0[0], x0[1], 'bo')
    ax.plot(xg[0], xg[1], 'ro')
    myvessel.draw_patch(ax, myvessel.x, fcolor='b')

    # nonpass = np.array(nonpassable)
    # ax.plot(nonpass[:,0], nonpass[:,1], 'rx')

    ax.axis('equal')
    ax.axis('scaled')
    ax.axis([-10, 160, -10, 160])
    mymap.draw(ax, 'g', 'k')
    ax.grid()

    plt.show()

```

E.4 Velocity Obstacle Particle Swarm Optimization Class

```python
import time

import numpy as np
import matplotlib.pyplot as plt

from map import Map
from utils import Controller
from vessel import Vessel

DIMENSIONS = 2  # Number of dimensions
GLOBAL_BEST = 0  # Global Best of Cost function
MIN_RANGE = 0  # Lower boundary of search space
MAX_RANGE = 50 # Upper boundary of search space
POPULATION = 50  # Number of particles in the swarm
V_MAX = 1 # Maximum velocity value
PERSONAL_C = 2.0  # Personal coefficient factor
SOCIAL_C = 2.0  # Social coefficient factor
CONVERGENCE = 0  # Convergence value
MAX_ITER = 100  # Maximum number of iterations
BIGVAL = 10000.
MINDIST = 20


class Vopso(Controller):
    def __init__(self, x0, xg, the_map, search_radius=50, replan=False):
        self.start = x0[0:3]
        self.goal = xg[0:3]
        self.scanRadius = search_radius
        self.world = None
        self.grid_size = the_map.get_dimension
        self.graph = SearchGrid(the_map, [1.0, 1.0, 25.0/360.0])
        self.map = the_map
        self.to_be_updated = True
        self.replan = replan
        self.path_found = False
        self.wpUpdated = False
        self.currentcWP = 0

        self.alter = 0
        self.totalTime = 0

        self.particles = []  # List of particles in the swarm
        self.best_pos = None  # Best particle of the swarm
        self.best_pos_z = np.inf  # Best particle of the swarm

    def update(self, vobj, world, vesselArray):
        tic = time.process_time_ns()
        if len(vesselArray) > 1:
            v2 = vesselArray[1]
            resetPoint = -1
            scanData = self.scan(vobj.x[0:2], v2.x[0:2])
            if scanData[0] <= self.scanRadius and not self.wpUpdated:

                # Create VO
                VOarray = self.createVO(vobj, v2, scanData)
                if VOarray[3] > VOarray[8] > VOarray[4]:
                    # Implement MOPSO
                    self.currentcWP = vobj.controllers[1].cWP
                    nextWP = self.search(vobj.x[0:2], vesselArray, scanData, VOarray)
                    print("Vessel 1: ", vobj.x[0:2])
                    print("Vessel 2: ", v2.x[0:2])
                    for x in range(0, 2):
                        print("Waypoint ", self.currentcWP + x, ": ", nextWP)
                        vobj.controllers[1].wp = np.insert(vobj.waypoints, self.
currentcWP + x, nextWP, axis = 0)
                        vobj.waypoints = np.insert(vobj.waypoints, self.currentcWP + x,
nextWP, axis = 0)
                        scanData = self.scan(nextWP, v2.x[0:2])
                        nextWP = self.search(nextWP, vesselArray, scanData, VOarray)
                    self.wpUpdated = True
                    self.totalTime = self.totalTime + (time.process_time_ns()-tic)
                    print(self.totalTime)

            if vobj.controllers[1].cWP == self.currentcWP + 1:
                vobj.wp = None
                vobj.controllers[1].cWP = 0
                vobj.controllers[0].to_be_updated = True
                vobj.controllers[1].wp_initialized = False
                self.wpUpdated = False
                self.totalTime = self.totalTime + (time.process_time_ns() - tic)
                print(self.totalTime)


    def search(self, vobjx, vesselArray, scanData, VOarray):

        # Initialize swarm
        x0 = vobjx[0:2]
        print("Sverm0 ", x0)

        swarm = Swarm(POPULATION, V_MAX, self.goal, x0, vesselArray, scanData, VOarray)
        # Initialize inertia weight
        inertia_weight = 0.5 + (np.random.rand() / 2)
        curr_iter=0
        for i in range(MAX_ITER):

            for particle in swarm.particles:

                for i in range(0, DIMENSIONS):
                    r1 = np.random.uniform(0, 1)
                    r2 = np.random.uniform(0, 1)

                    # Update particle's velocity
                    personal_coefficient = PERSONAL_C * r1 * (particle.best_pos[i] - particle.pos[i])
                    social_coefficient = SOCIAL_C * r2 * (swarm.best_pos[i] - particle.pos[i])
                    new_velocity = inertia_weight * particle.velocity[i] + personal_coefficient + social_coefficient

                    # Check if velocity is exceeded
                    if new_velocity > V_MAX:
                        particle.velocity[i] = V_MAX
                    elif new_velocity < -V_MAX:
                        particle.velocity[i] = -V_MAX
                    else:
                        particle.velocity[i] = new_velocity

                # Update particle's current position
                particle.pos += particle.velocity

                particle.pos_z = swarm.cost_function(particle.pos[0], particle.pos[1], vesselArray)

                # Update particle's best known position
                if particle.pos_z < swarm.cost_function(particle.best_pos[0], particle.best_pos[1], vesselArray):
                    particle.best_pos = particle.pos.copy()

                    # Update swarm's best known position
                    if particle.pos_z < swarm.best_pos_z:
                        swarm.best_pos = particle.pos.copy()
                        swarm.best_pos_z = particle.pos_z

                # # Check if particle is within boundaries
                biggest = 0
                hypeCheck = np.hypot(particle.pos[0] - x0[0], particle.pos[1] - x0[1])
                if hypeCheck > biggest:
                    biggest = hypeCheck
                if np.hypot(particle.pos[0] - x0[0], particle.pos[1] - x0[1]) >
    MAX_RANGE:
                    r = np.random.uniform(MIN_RANGE, MAX_RANGE)
                    theta = np.random.uniform(0, 2 * np.pi)
                    particle.pos[0] = (r * np.cos(theta)) + x0[0]
                    particle.pos[1] = (r * np.sin(theta)) + x0[1]


            # Check for convergence
            if abs(swarm.best_pos_z - GLOBAL_BEST) < CONVERGENCE:
                print("The swarm has met convergence criteria after " + str(curr_iter
) + " iterations.", 'at:',swarm.best_pos)
                break
            curr_iter += 1

        if abs(swarm.best_pos_z - GLOBAL_BEST) > CONVERGENCE:
            print("The swarm has reached max iterations after " + str(curr_iter) + "
iterations.", 'at:',
                  swarm.best_pos)
        print("Swarm: ", swarm.best_pos)
        print("Biggest: ", biggest)
        return [swarm.best_pos[0], swarm.best_pos[1]]

    def scan(self, vessel1, vessel2):
        xd = (vessel2[0] - vessel1[0])
        yd = (vessel2[1] - vessel1[1])
        distance = abs(np.sqrt(xd**2 + yd**2))
        angle = np.arctan2(yd, xd)

        return [distance, angle]

    def createVO(self, vessel1, vessel2, scanData):
        VO = [0, 0, 0, 0, 0, 0, 0, 0, 0]
        # find which side crossing vessel is coming from
        if vessel2.x[0] > vessel1.x[0] and (np.pi / 2 < vessel2.x[2] < 3 * np.pi / 2):
            VO[0] = 'r'
        elif vessel2.x[0] < vessel1.x[0] and (vessel2.x[2] < np.pi / 2 or vessel2.x[2
] > 3 * np.pi / 2):
            VO[0] = 'l'
        else:
            VO[0] = 'n'

        # find left and right boundaries of collision cone
        VO[1] = scanData[0]
        VO[2] = scanData[1]
        angle = np.arctan2(scanData[0] / 2, scanData[0])

        VO[3] = VO[2] + np.arctan2((scanData[0] / 2) + 5, scanData[0])
        VO[4] = VO[2] - np.arctan2((scanData[0] / 2) + 5, scanData[0])

        # find vector (xab) and angle (lab) of relative velocity
        VO[5] = [np.cos(vessel1.x[2]), np.sin(vessel1.x[2])]
        VO[6] = [(np.cos(vessel2.x[2])), (np.sin(vessel2.x[2]))]
        VO[7] = [VO[5][0] - VO[6][0], VO[5][1] - VO[6][1]]
        VO[8] = np.arctan2(VO[7][1], VO[7][0])

        return VO

class Swarm():
    def __init__(self, pop, v_max, goal, x0, vesselArray, scanData, VOarray):
        self.particles = []         # List of particles in the swarm
        self.best_pos = None        # Best particle of the swarm
        self.best_pos_z = np.inf    # Best particle of the swarm
        self.x0 = x0                #ship pos
        self.goal = goal
        self.scanData = scanData
        self.vesselArray = vesselArray
        self.VOarray = VOarray
        for _ in range(pop):
                r = np.random.uniform(MIN_RANGE, MAX_RANGE)
                theta = np.random.uniform(MIN_RANGE, MAX_RANGE*np.pi)
                x = (r * np.cos(theta))+x0[0]
                y = (r * np.sin(theta))+x0[1]

                z = self.cost_function(x, y, goal)
                velocity = np.random.rand(2) * v_max
                particle = Particle(x, y, z, velocity)
                self.particles.append(particle)
                if self.best_pos != None and particle.pos_z < self.best_pos_z:
                    self.best_pos = particle.pos.copy()
                    self.best_pos_z = particle.pos_z
                else:
                    self.best_pos = particle.pos.copy()
                    self.best_pos_z = particle.pos_z


        def cost_function(self, x1, y1, goal):
            devW = 1
            statW = 1
            dynW = 1
            weighingMatrix = np.array([[devW], [statW], [dynW]])
            pos = x1,y1
            x2,y2=self.goal[0],self.goal[1]

            deviation_cost = (np.sqrt((x2-x1)**2 + (y2-y1)**2))   #distance from goal

            statitc_obs_cost = 0

            distance = np.hypot(x1 - self.vesselArray[1].x[0], y1 - self.vesselArray[1].x[1
    ])    #check for dynamic obstacle
            if distance <= 5:
                dyn_obs_cost = BIGVAL
            elif 5 < distance <= 10:
                dyn_obs_cost = 100
            elif 10 < distance <= 15:
                dyn_obs_cost = 50
            else:
                dyn_obs_cost = 0

            if self.is_inside(self.get_dangercone(), pos):
                dyn_obs_cost = BIGVAL

            if dyn_obs_cost < 0:
                dyn_obs_cost = 0

            cost = np.sum(np.array([[deviation_cost], [statitc_obs_cost], [dyn_obs_cost
    ]]) * weighingMatrix)
            return cost

        def is_inside(self, triangle, pos):           #check if point is inside cone
            x1 = triangle[0]
            x2 = triangle[1]
            x3 = triangle[2]
            xp = pos

            c1 = (x2[0]-x1[0]) * (xp[1]-x1[1]) - (x2[1]-x1[1]) * (xp[0]-x1[0])
            c2 = (x3[0]-x2[0]) * (xp[1]-x2[1]) - (x3[1]-x2[1]) * (xp[0]-x2[0])
            c3 = (x1[0]-x3[0]) * (xp[1]-x3[1]) - (x1[1]-x3[1]) * (xp[0]-x3[0])
            if (c1 < 0 and c2 < 0 and c3 < 0) or (c1 > 0 and c2 > 0 and c3 > 0):
                return True
            else:
                return False

        def get_dangercone(self):
            v1 = self.vesselArray[0]
            v2 = self.vesselArray[1]
            VOarray = self.VOarray
            tc = self.getCollisionTime(v1, v2)

        p0 = [v1.x[0] + (v2.x[3] * np.cos(v2.x[2]))*tc, v1.x[1] + (v2.x[3] * np.sin(v2.x[2]))*tc]
        p1 = [((self.scanData[0] + v2.x[3]*tc) * np.cos(VOarray[3]) + p0[0]), (self.scanData[0] + v2.x[3]*tc) * np.sin(VOarray[3]) + p0[1]]
        p2 = [((self.scanData[0] + v2.x[3]*tc) * np.cos(VOarray[4]) + p0[0]), (self.scanData[0] + v2.x[3]*tc) * np.sin(VOarray[4]) + p0[1]]

        #print("Triangle: ", p0, p1, p2)

        return [p0, p1, p2]

    def getCollisionTime(self, v1, v2):
        r1 = ([v1.x[0], v1.x[3] * np.cos(v1.x[2]), v1.x[1], v1.x[3] * np.sin(v1.x[2])])  # vessel 1 velocity vector
        r2 = ([v2.x[0], v2.x[3] * np.cos(v2.x[2]), v2.x[1], v2.x[3] * np.sin(v2.x[2])])  # vessel 2 velocity vector

        tx = (r2[0] - r1[0]) / (r1[1] - r2[1]) # time to intersect in x
        ty = (r2[2] - r1[2]) / (r1[3] - r2[3]) # time to intersect in y

        return (tx + ty) / 2 # returns average of x and y times


# Particle class
class Particle():
    def __init__(self, x, y, z, velocity):
        self.pos = [x, y]
        self.pos_z = z
        self.velocity = velocity
        self.best_pos = self.pos.copy()


class SearchGrid(object):
    """General purpose N-dimentional search grid."""
    def __init__(self, the_map, gridsize, N=2, parent=None):
        self.N        = N
        self.grid     = the_map.get_discrete_grid()
        self.map      = the_map
        self.gridsize = gridsize
        self.gridsize[0] = the_map.get_gridsize()
        self.gridsize[1] = the_map.get_gridsize()
        dim = the_map.get_dimension()

        self.width  = dim[0]
        self.height = dim[1]

        """
        In the discrete map, an obstacle has the value '1'.
        We multiply the array by a big number such that the
        grid may be used as a costmap.
        """
        self.grid *= BIGVAL


    def get_grid_id(self, state):
        """Returns a tuple (x,y,psi) with grid positions."""
        return (int(state[0]/self.gridsize[0]),
                int(state[1]/self.gridsize[1]),
                int(state[2]/self.gridsize[2]))

    def in_bounds(self, state):
        return 0 <= state[0] < self.width and 0 <= state[1] < self.height

    def cost(self, a, b):
        #if b[0] > self.width or b[1] > self.height:
        #    return 0
        return np.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)#self.grid[int(b[0]/self.gridsize[0]), int(b[1]/self.gridsize[1])]
```

```python

    def passable(self, state):
        # TODO Rename or change? Only returns true if object is _inside_ obstacle
        # Polygons add safety zone by default now.

        if state[0] > self.width or state[1] > self.height:
            return True

        return self.grid[int(state[0]/self.gridsize[0]),
                         int(state[1]/self.gridsize[1])] < BIGVAL

    def neighbors(self, state, est_r_max):
        """
        Applies rudder commands to find the neighbors of the given state.

        For the Viknes 830, the maximum rudder deflection is 15 deg.
        """
        step_length = 2.5*self.gridsize[0]
        avg_u       = 3.5
        Radius      = 2.5*avg_u / est_r_max
        dTheta      = step_length / Radius
        #print(Radius, dTheta*180/np.pi)

        trajectories = np.array([[step_length*np.cos(dTheta), step_length*np.sin(dTheta
), dTheta],
                                 [step_length, 0.,  0.],
                                 [step_length*np.cos(dTheta), -step_length*np.sin(dTheta
), -dTheta]])

        #print(trajectories)
        results = []
        for traj in trajectories:
            newpoint = state + np.dot(Rz(state[2]), traj)
            if self.passable(newpoint):
                results.append(newpoint)

        #results = filter(self.in_bounds, results)

        return results


#############################################################################


if __name__ == "__main__":
    mymap = Map("s1", gridsize=1.0, safety_region_length=4.5)

    x0 = np.array([0, 0, np.pi / 2, 3.0, 0.0, 0])
    xg = np.array([30, 86, np.pi / 4])
    mopso = Mopso(x0, xg, mymap)
    myvessel = Vessel(x0, xg, 0.05, 0.5, 1, [mopso], True, 'viknes')
    vesselArray = [myvessel]


    mopso.update(myvessel, vesselArray)

    fig = plt.figure()
    ax = fig.add_subplot(111, autoscale_on=False)
    #   ax.plot(myvessel.waypoints[:, 0],
    #           myvessel.waypoints[:, 1],
    #           '-')

    # ax.plot(x0[0], x0[1], 'bo')
    ax.plot(xg[0], xg[1], 'ro')
    myvessel.draw_patch(ax, myvessel.x, fcolor='b')

    # nonpass = np.array(nonpassable)
    # ax.plot(nonpass[:,0], nonpass[:,1], 'rx')

    ax.axis('equal')
    ax.axis('scaled')
    ax.axis([-10, 160, -10, 160])
    mymap.draw(ax, 'g', 'k')
    ax.grid()

    plt.show()

```