Autonomous Docking

Trajectory planning and dynamic route adaptation for autonomously docking a marine vessel using Model Predictive Control

Master's thesis in Systems, Control and Mechatronics

ERIK LAITALA
EVELINA STRÖMDAHL

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Autonomous Docking

Trajectory planning and dynamic route adaptation for autonomously docking a marine vessel using Model Predictive Control

ERIK LAITALA
EVELINA STRÖMDAHL

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
Gothenburg, Sweden 2024

Autonomous Docking
Trajectory planning and dynamic route adaptation for autonomously docking a marine vessel using Model Predictive Control
ERIK LAITALA
EVELINA STRÖMDAHL

Supervisors: Axel Måneskiöld & Daniel Söderberg, CPAC Systems AB
Examiner: Peter Forsberg, Department of Mechanics and Maritime Sciences

Master's Thesis 2024
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: From CPAC's easy boating video.

Typeset in LaTeX
Printed by Teknologtryck
Gothenburg, Sweden 2024

Autonomous Docking
Trajectory planning and dynamic route adaptation for autonomously docking a marine vessel using Model Predictive Control
Erik Laitala
Evelina Strömdahl
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Abstract

This thesis presents a comprehensive study on the application of Model Predictive Control (MPC) for the autonomous docking of marine vessels. The research focuses on developing and implementing trajectory planning and dynamic route adaptation algorithms that enable a vessel to autonomously dock in various maritime environments. The research on autonomous docking is important for its potential to enhance safety, efficiency, and reliability in maritime operations, particularly in crowded or challenging docking scenarios. This technology aims to minimize human error and simplify the docking procedure in busy marine environments. The key challenges addressed include path planning, collision avoidance, trajectory tracking, and the integration of real-time dynamic adjustments to account for moving obstacles and environmental changes. Our methodology utilizes MPC to continuously predict and optimize the vessel's path, thereby ensuring safe and efficient docking maneuvers. A simulation environment created in Python and real-world simulations created in a Unity-based environment were utilized to validate the effectiveness of the proposed algorithm. Simulation results demonstrated a functional trajectory planner capable of successfully following a reference path, avoiding obstacles, and docking a marine vessel in narrow spaces, indicating the potential for using MPC to autonomously dock a boat. Initial tests in a real-world environment were performed to further confirm the potential of the proposed solution. Comparative performance analysis highlights the strengths and limitations of the system during different conditions, demonstrating its potential for real-world application. Future works aim to enhance the complexity of the maritime scenarios and vessel dynamics handled by the algorithms, as well as additional testing in real-world environments to further validate and refine the system. The presented analysis also demonstrates the potential for additional features to improve the stability and reliability of the MPC-based system. This includes the integration of various sensor data for extensive environmental mapping providing real-time updates about the surroundings to ensure a safer docking procedure.

Keywords: Model Predictive Control, Receding Horizon Control, Path Planning, Trajectory Planning, Collision Avoidance, Autonomous docking, Maritime Navigation, Dynamic Route Adaptation.

Acknowledgements

We would like to thank *CPAC Systems AB* for providing the facilities and equipment to perform this work. In particular, we would like to thank Peter Forsberg our examiner, and our supervisors Daniel Söderberg and Axel Måneskiöld for their continuous support and insightful ideas. Furthermore, we would like to give special thanks to Erik Lund for his invaluable assistance with simulations and Jonatan Bergenwall for his exceptional academic guidance.

Erik Laitala & Evelina Strömdahl, Gothenburg, May 2024

Thesis advisors: Axel Måneskiöld & Daniel Söderberg, CPAC Systems AB
Thesis examiner: Peter Forsberg, Mechanics and Maritime Sciences

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

CTE	Cross-Track Error
DWA	Dynamic Window Approach
EVC	Electronic Vessel Control
GPS	Global Positioning System
IMU	Inertial Measurement Unit
IPOPT	Interior Point OPTimizer
LiDAR	Light detection and ranging
MPC	Model Predictive Control
NMPC	Nonlinear Model Predictive Control
OCP	Optimal Control Problem
OpEn	Optimization Engine
PANOC	Proximal Averaged Newton-Type Method
RHC	Receding Horizon Control
RK4	Runge Kutta 4
RRT	Rapidly Exploring Random Tree
SNOPT	Sparce Nonlinear OPTimizer
UAV	Unmanned Aerial Vehicle

Nomenclature

Below is the nomenclature of parameters and simulation weights that has been used throughout this thesis.

Parameters

$\boldsymbol{\eta}$	Pose vector
$\dot{\boldsymbol{\eta}}$	Kinematic model
$\boldsymbol{\nu}$	Velocity vecotor
$\mathcal{R}(\psi)$	Rotation matrix
$u_{\min}$	Vehicle constraint on the minimal velocity possible (m/s)
$u_{\max}$	Vehicle constraint on the maximal velocity possible (m/s)
$v_{\min}$	Vehicle constraint on the minimal velocity possible (m/s)
$v_{\max}$	Vehicle constraint on the maximal velocity possible (m/s)
$r_{\min}$	Vehicle constraint on the maximal angular velocity (radians/s)
$r_{\max}$	Vehicle constraint on the maximal angular retardation (considered symmetric) (radians/s)
n_u	Number of control inputs
n_x	Number of states
t_s	Time step in Unity and Python (s)
$\mathcal{P}$	Boundary coordinates
$\mathcal{O}$	Obstacle list
N_O	Max number of static obstacles
N	Length of the receding horizon controller in Python/Unity
path_interval	Path interval points sent to the MPC

Simulation Weights

$\mathcal{Q}_{\mathcal{O}}$	Weight on static obstacle avoidance
$\mathcal{Q}_{\mathcal{B}}$	Weight on the boundary constraints
$\mathcal{Q}_{\mathcal{D}}$	Weight on dynamic obstacle avoidance
$\mathcal{Q}_{CTE}$	Weight on the cross-track error
r	Weight on the deviation from the reference control input
$\mathcal{Q}_{v}$	Weight on prioritizing forward drive
q_{av}	Weight on acceleration in surge direction
q_{au}	Weight on acceleration in sway direction
$q_{a\psi}$	Weight on acceleration in the angular rotation
$\mathcal{Q}_{\mathcal{N}}$	Weight on the terminal state
$\mathcal{Q}_{\psi\mathcal{N}}$	Weight on the terminal heading

Contents

List of Figures

List of Tables

1
Introduction

In contrast to the automotive sector, where significant technical improvements have been made in the development of automotive features, the maritime industry is still in the starting phase. Unlike land-based vehicles, marine vessels face unique challenges, including factors such as wind, current, and strong waves significantly impacting navigation and docking procedures. For beginners and even experienced sailors, maneuvering and docking, particularly in adverse conditions, can be difficult and potentially lead to undesirable consequences.

The current approach to docking heavily relies on manual techniques and the skill of the operator. This master's thesis seeks to address this gap by proposing the development of an algorithm capable of safely and autonomously docking marine vessels in a busy environment, specifically targeting leisure boats. However, the focus extends to broader applicability across vessel types. The algorithm's objective includes navigating vessels around both static and dynamic obstacles in a busy harbor with narrow passages while considering environmental factors. Making a significant step towards enhanced efficiency and user-friendliness in docking operations. Utilizing Model Predictive Control (MPC) to predict optimal navigation, this thesis aims to develop a functional algorithm through an in-depth analysis of existing literature, dynamic vessel modeling, and rigorous simulation and testing.

1.1 Related work

In the research field of autonomous docking, a shift is underway to address the difficulties of a marine environment. Complexities introduced by factors such as wind, current, and strong waves present unique challenges for marine vessels during docking and navigation, making it a critical area of advancement in the maritime industry. Automating the docking procedure shifts the responsibility for vessel navigation from the operator's skills to the algorithm, ensuring a safer and more consistent operation. However, to rely on an algorithm to control a vessel, it must be capable of consistently ensuring safe maneuvering without collisions or unnecessary risks. This requires a thorough understanding of both the vehicle's dynamics and the surrounding environment.

In light of these challenges and opportunities, this section reviews existing research efforts in autonomous docking systems, involving various path-planning algorithms and methods for trajectory planning, highlighting advancements, limitations, and

key insights that motivate the presented thesis.

In the paper "Dijkstra's and a-star in finding the shortest path: A tutorial" [1] the utilization of path planning on a preprocessed map is discussed as a well-established method for controlling and navigating robots or vehicles. By transforming the target environment into a simplified 2D space, partitioned into walkable and non-walkable sections and by imposing predefined restrictions, such as avoiding obstacles or minimizing travel distance, path planning aims to find an optimal path for navigation. While various algorithms exist for this purpose, each with its advantages and drawbacks, they typically search for the shortest and safest path within the defined constraints. Moreover, while some may consider the orientation of the vehicle, these algorithms operate under the constraint of solely relying on coordinates or nodes, without utilizing the dynamics of the robot or vehicle.

The trajectory planning process for autonomous marine vessels followed by establishing a predefined path is outlined in a study by Lutz et. al [2]. This method involves the development of a model of the boat capable of capturing the dynamics of the vessel. Addressing various constraints, including the model, the generated path, environmental factors, prioritizing of forward drive, etc. is accomplished through the implementation of MPC which is a widely used method for solving constrained optimization problems. In MPC the dynamic model is used to represent the system while addressing specific constraints. These characteristics make MPC well-suited for trajectory planning in the context of autonomous marine navigation.

The interest in autonomous docking of marine vessels has gained more attention due to growing interest in autonomous marine operations such as autonomous ferries and cargo vessels. Despite this, limited literature and research on the topic exists, where the focus has more often been on specific vessel types. The paper "Autonomous docking using direct optimal control" [3] proposes a method that introduces the docking problem as an optimal control task. This paper's approach to autonomous docking involves numerical optimal control techniques. A method that ensures maneuverability while complying with physical constraints is proposed where the kinematics and thrust configuration of a cargo ship are modeled. Spatial constraints are also considered during the docking procedure to ensure collision avoidance. The findings of the research suggest that the proposed method is effective at successfully demonstrating safe docking maneuvers without violating any safety restrictions. However, while the paper provides a foundation for autonomous docking using optimal control, it leaves room for extending the research to address different challenges encountered in other boat operations concerning smaller marine vessels. This includes more challenging scenarios such as navigating narrow spaces and dynamically changing environments.

To achieve a smooth automatic parking algorithm, a paper presented a lateral vehicle trajectory planner and a control algorithm using MPC for an automatic perpendicular parking system [4]. A clothoid path planner was used together with a simple kinematic model to achieve the smooth steering motion. The MPC optimizes a fi-

nite time horizon to satisfy vehicle constraints and control the motion. Experiments from the paper showcased an effective algorithm that controlled a vehicle to handle a smooth convergence to a set parking spot; even meeting the requirements outlined in ISO 16787:2017 ”Intelligent transport systems and assisted parking parking systems” [4].

An important aspect to consider in the field of autonomous driving is the problem of collision avoidance. As a consequence, there is extensive research that focuses on this topic. In a paper by Helling et al. [5] a dual collision detection method in MPC with various culling techniques is proposed for obstacle avoidance. The authors employ optimization-based approaches, using dual re-formulation of indicator, distance, and signed distance functions to address nonlinear, non-continuously differentiable formulations precisely. The implementation of culling techniques such as frustum, occlusion, and backface culling, is shown to reduce computational complexity successfully. However, despite promising results in confined environments, such as a path-following scenario for an autonomous surface vessel, multiple challenges remain unaddressed. The main drawbacks include the method’s scalability to more dynamic and unpredictable environments and the system’s flexibility to various obstacle geometries necessary for an autonomous docking system. This thesis aims to address these challenges by focusing on dynamic route adaptation and obstacle avoidance in a complex environment such as a busy harbour containing dynamic, and static obstacles and narrow passages. This could advance the practical applicability of MPC in different autonomous docking scenarios.

The paper "Obstacle avoidance in real time with nonlinear model predictive control of autonomous vehicles" [6] presents a study of nonlinear model predictive control (NMPC) for trajectory tracking and obstacle avoidance in autonomous road vehicles at realistic speeds. The focus of the study primarily lies on the performance relative to the prediction horizon of the NMPC. The study compares two different techniques for obstacle avoidance in real-time; the first method uses trigonometric identities to find the nearest point on the reference trajectory, this point is then used as the desired trajectory point for the current controller step. This method is based on the principle of repelling the vehicle from the closest point on the trajectory. In the second method, at each time step, a desired point on the trajectory is calculated based on lines drawn from the center of gravity of the vehicle to the reference trajectory and the point closer to the goal point is selected as the desired trajectory to follow. The study contributes to existing literature in the field of autonomous vehicle control and emphasizes the importance of longer prediction horizons for effective obstacle avoidance. However, although the focus is on realistic driving environments, narrow spaces and passages are not considered. The paper also only deals with static obstacles and does not consider the additional challenges that appear with dynamic obstacles.

1.1.1 Research questions

Based on the background and related work in the field of automation in marine environments, this thesis aims to formulate an algorithm that can autonomously dock a marine vessel in a constrained environment that includes both static and dynamic obstacles and narrow passages. The algorithm needs to safely steer the vessel, avoid obstacles, and dock the vessel at a preset destination with the right orientation.

This will be achieved by answering the following research questions:

- How can an optimized path that prioritizes minimum length, safety, and user comfort in different marine navigation scenarios be found?
- How can an MPC be formulated with the optimized path as a reference and the dynamics of the boat in mind?
- What metrics and criteria are suitable for evaluating the performance of the algorithm in terms of path selection, obstacle avoidance, user comfort, and reaching the destination?
- How does the proposed system perform in challenging maritime scenarios and what insights can be gained about its applicability in real maritime operations?

1.2 Scope and limitations

The proposed algorithm theoretically has applicability across various harbor environments, as long as it has been appropriately pre-processed before. To streamline implementation and reduce pre-processing requirements of the harbor structure and its obstacles' positions, boundaries, and orientation; this project focuses on the harbor layout and obstacle configurations at Volvo Penta's harbor in Krossholmen. The map is assumed to describe the environment of Krossholmen with navigable and non-navigable space. The obstacles are limited to approximations of different-sized polygons with four vertices and varying orientations in the world map; with nodes constructing their shape.

Boat dynamics are modeled with certain assumptions to ensure a functional representation and still capture boats with varying sizes. These assumptions include neglecting roll, pitch, and heave by assuming calm water and modeling the boat's motion using a purely kinematic model without any force interactions. Testing and simulations are done in three steps. The first steps include simulations in an environment created in Python using Matplotlib as a replica of the harbor in Krossholmen. The second step involves validating the algorithm's performance by testing and simulations on a boat simulator at CPAC providing more realistic maritime scenarios. The third step involves real-world testing in an actual maritime environment. Testing will primarily be done on a Cranchi Endurance 41 with Volvo Penta D6 engines, with the model tailored to reasonably accommodate variations in boat sizes while maintaining practicality. The third step is performed with limitations due to the time constraints of the thesis work.

1.3 Methodology

To reach the goals of this master thesis and to answer the stated research questions, the following steps were taken.

1. Review of existing literature:
   A comprehensive literature review was conducted, focusing on topics related to autonomous boat docking, dynamic models, and model predictive control.
2. Vessel modeling:
   A model was identified and constructed to accurately capture the motion of a marine vessel.
3. Mathematical formulation of the optimization problem:
   The optimization problem was mapped out by integrating a kinematic boat model, environmental factors, and related issues associated with autonomous docking. The mathematical structure within which the algorithm operates was established.
4. Simulations of different scenarios:
   Multiple simulations were performed to explore the algorithm and achieve satisfactory results under varying conditions. Factors such as obstacle avoidance, precise navigation, and adaptability to environmental changes were considered.
5. Rig tests and simulation on a boat simulator:
   Rig tests were conducted to validate the algorithm's performance in a simulated boat environment under controlled conditions. A boat simulator was used to test the algorithm's response time and accuracy, emulating real-world scenarios.
6. Real-world tests based on viable simulation results:
   When the simulations yielded promising results, real-world boat tests were conducted.

2
Theory

The following chapter introduces the theoretical framework essential for the research presented in this thesis. The chapter comprises three parts. Firstly relevant path-planning algorithms critical to the field of automation are presented/described. The second part introduces the basics and necessary building stones of a model predictive controller (MPC). In the third part, relevant software architecture used for solving optimization problems is also introduced.

2.1 Path planning algorithms

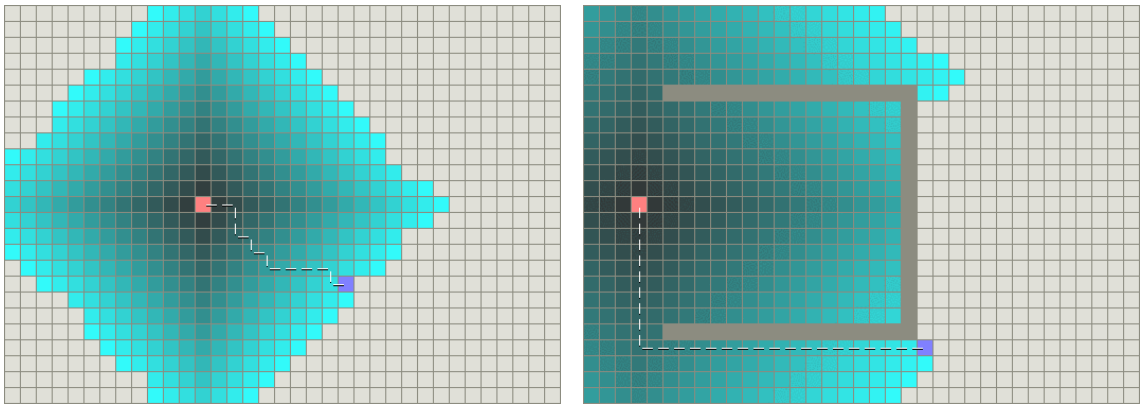
Path planning algorithms and trajectory planning are essential parts of the field of automation. Path planning consists of generating a geometric path between an initial point and a final point, focusing only on the spatial aspects of the journey [7]. The trajectory planning algorithm consists of assigning a time law to the geometric path [8], taking aspects such as velocity and acceleration into account. These sections focus on a deeper understanding of different algorithms aiming to uncover their limitations and applications within the field of automation.

2.1.1 Dijkstra algorithm

Dijkstra's algorithm is a widely used algorithm for path planning formulated by computer scientist Edsger W. Dijkstra in 1956 and published in 1959 [1]. The algorithm finds the shortest path from a single source vertex to all other vertices in a weighted graph. The traditional Dijkstra algorithm searches for the shortest path in the order of an increasing path length. The basic idea behind it is to find the shortest path from a node to a destination node in a map with a single source [9]. How the Dijkstra algorithm searches for the optimum path is shown in Figure 2.2. The pseudocode for Dijkstra's algorithm follows the structure in Appendix A.2.
A drawback of the Dijkstra algorithm is that it solves the single shortest path problem for a graph with only non-negative edge path costs. It is impossible to pass a path if a graph contains negative edges which may result in inaccurate results [1].

2.1.2 A* algorithm

The A* algorithm is one of the most widely used path-finding algorithms in computer science, originally formulated by Peter Hart et al. in 1968 [11]. A* is an informed search algorithm, meaning it uses heuristic information to guide its search



(a) Dijkstra algorithm without obstacles. From: Introduction to A*[10]

(b) Dijkstra algorithm with obstacles. From: Introduction to A*[10]

Figure 2.1: The Dijkstra search algorithm mapping out a path from the start node (red) to the end node (purple), visited nodes are in colors unvisited in gray. From: Introduction to A*[10]

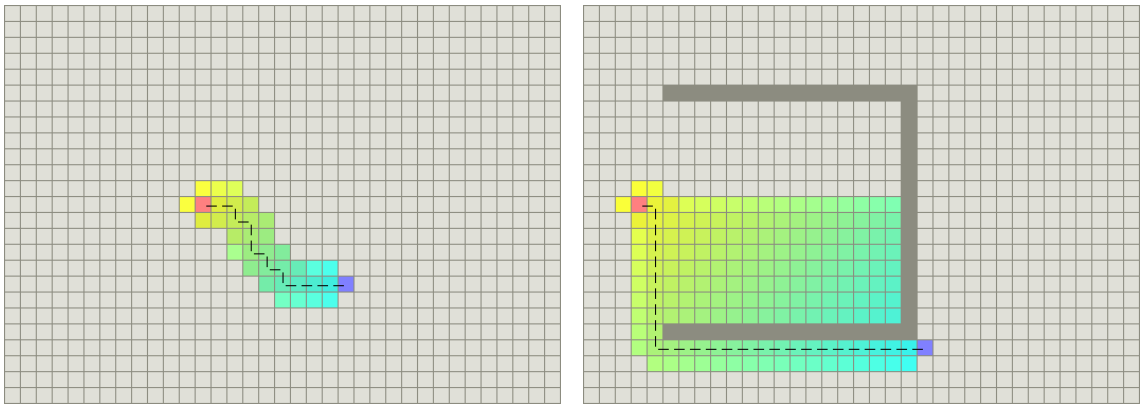
process towards the goal. Unlike uninformed search algorithms like Dijkstra's algorithm, A* selects which path to explore first, leading to a significant improvement in efficiency [1]. The algorithm works by having a priority queue of nodes yet to be explored. At each interaction step, the algorithm selects the node with the lowest combined cost of the path from the start node and the heuristic estimate of the distance to the goal; this cost is calculated using the formula 2.1 [11].

$$f(n) = g(n) + h(n) \tag{2.1}$$

g(n) is the cost of the path from the start node to node n and h(n) is the heuristic estimate of the distance from node n to the goal. How the A* algorithm searches for the optimum path is shown in Figure 2.2. The pseudocode for the A* algorithm is formulated in Appendix A.1.

2.1.3 Dynamic window approach

The dynamic window approach (DWA) is a local reactive avoidance technique that works within the velocity space [12]. The method describes an object's, often a robot's, trajectory through a sequence of circular and straight-line arcs showcased in Figure 2.3.



(a) A* algorithm without obstacles.
From: Introduction to A*[10]

(b) A* algorithm with obstacles.
From: Introduction to A*[10]

Figure 2.2: The A* search algorithm mapping out a path from the start node (red) to the end node (purple), visited nodes are in colors and unvisited are in gray.
From: Introduction to A*[10]

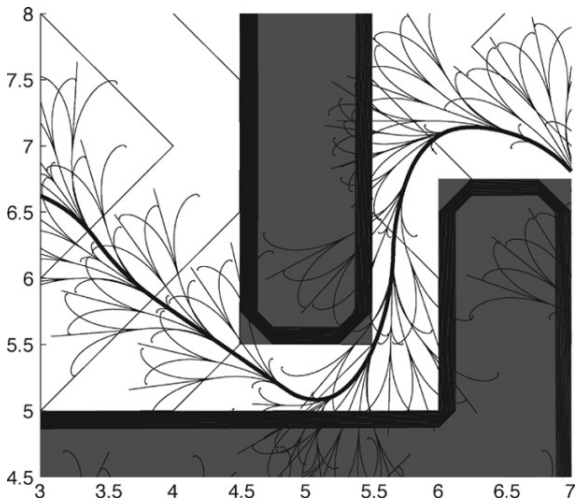


Figure 2.3: Example of the local reactive avoidance technique employed by DWA. An MPC-controlled robot with a trajectory including all the evaluated options.
From: A convergent dynamic window approach to obstacle avoidance[13] .

The search space is constrained by the kinematic and dynamic constraints of the robot, focusing on a specific range of velocities centered around the current velocity vector (v_c, w_c). Where (v_c, w_c) are the robot's current linear velocity and angular velocity. The range is reachable within the next sampling interval Δt, which defines the dynamic window V_d of possible reachable velocities, as described in equation 2.2.

$$V_d = \{(v, \omega) \mid v \in [v_c - \dot{v}_b \Delta t, v_c + \dot{v}_a \Delta t] \wedge \omega \in [\omega_c + \dot{\omega}_b \Delta t, \omega_c + \dot{\omega}_a \Delta t]\} \tag{2.2}$$

Here $\dot{v}_a$ and $\dot{v}_b$ represent the maximal translation acceleration and deceleration, while $\dot{w}_a$ and $\dot{w}_b$ represents the maximal rotational acceleration and deceleration. V_d is described as safe if the robot can stop along the trajectory defined by it before

encountering any obstacles. To ensure a possible and fast reactive response, the dynamic window approach focuses exclusively on the first sampling interval, assuming constant velocities for the remaining intervals [12]. This reduction simplifies the search space to a two-dimensional set of velocity tuples (v, w). The set V_a of acceptable velocities can be calculated according to equation 2.3 [12].

$$V_a = \{(v, \omega) \mid \leq \sqrt{2\rho_{min}(v, \omega)v_b} \wedge \sqrt{2\rho_{min}(v, \omega)\omega_b}\} \tag{2.3}$$

The term $\rho_{min}(v, w)$ represents the distance to the closest obstacle on the corresponding curvature.

2.2 Model Predictive Control

Model Predictive Control (MPC) is a control technique in which the current control action is obtained by solving, at each sampling instance, a finite, open-loop, optimal control problem [14]. The fundamental idea behind MPC is built upon the concept of optimal control. The basic idea is to use a dynamic model to predict system behavior and optimize the prediction to provide the optimal decision - the control action at the current time [15]. The model of the system is hence of high importance when using MPC since the optimal control depends on the initial state of the dynamic system. A second concept of MPC is therefore to use past measurements to estimate the most likely initial state of the system. This involves examining records of past data and integrating them with the system model to solve the state estimation problem and identify the most likely value of the state at any given time. In summary, both the control problem where predictions of a model are used to determine the optimal control action and the state estimation problem where past data is used to estimate an optimal state value involves dynamic models and optimization.

The core of MPC can be summarized as the *receding horizon idea* which follows the steps:

1. At time instant k, *predict* the process response over a finite prediction horizon N. This response will depend on the sequence of future control inputs over the control horizon, M.
2. Choose the control sequence that minimizes a set of cost functions, i.e., which gives the best performance in terms of a specified objective.
3. Apply the first element of the control sequence, the current control action, to the process, discard the rest of the sequence, and return to step 1.

The key ingredients of formulating MPC involve an internal model describing the processes and disturbances of a system, an estimator/predictor to determine the evolution of the state, and an objective/criterion to express the desired system behavior. It also contains an optimization algorithm to determine future control actions and the receding horizon principle explained above [16]. In the following subsections, each of these parts will be studied further.

2.2.1 Dynamic models

Employing MPC for trajectory generation for autonomous vehicles starts by developing a dynamic model of vehicle motion. Dynamic models build on differential equation models,

$$\begin{aligned} \frac{dx}{dt} &= f(x, u, t) \\ y &= h(x, u, t) \\ x(t_0) &= x_0 \end{aligned} \tag{2.4}$$

where $x \in \mathbb{R}^n$ is the state. $u \in \mathbb{R}^n$ is the input, $y \in \mathbb{R}^p$ is the output, and $t \in \mathbb{R}^n$ is the time [15]. The initial value for the system, x_0, is specified by the values of the state x at the time $t = t_0$. The emphasis for dynamic modeling of the motion of a vehicle will be on discrete-time state space models, but these often originate from continuous-time models of a process, which may be obtained by linearizing an originally non-linear model.

2.2.1.1 Discretization methods

In most MPC formulations the system dynamics are given in continuous time in the form of the differential equations presented in 2.4. However, when searching for solutions to engineering problems the controller will, in practice, almost always be implemented through a digital computer by sampling the variables of the system and transmitting the control action to the system at discrete time steps. Hence, discretizing the continuous time model to a discrete-time model is a necessary step to take [17]. There are multiple methods for discretizing a continuous time model, a more simple method is the first-order Euler forward integration method

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \cdot f(\mathbf{x}_k, \mathbf{u}_k), \quad k = 0, \ldots, N \tag{2.5}$$

where $\mathbf{u}_k = \mathbf{u}(t_k)$, starting from the initial conditions $\mathbf{x}_0$. The larger the time step Δt is, the higher the deviation from the true model trajectory $\mathbf{x}(t)$. Hence, to capture the true model dynamics of the system and get a reasonable accuracy a shorter time step is necessary, which would increase the optimization problem's complexity. The numerical stability is also an aspect to consider when choosing a reliable integration method. For the Euler scheme, by having a time step, Δt that is too large, there is a possibility that any error will be amplified during each iteration, meaning that after some time the value of $\mathbf{x}_k$ might become unstable and hence also meaningless. In other words, the explicit Euler scheme will deliver an unstable simulation when the time step Δt is too large [18].

A more widely used method of integration is Runge Kutta 4 (RK4). The RK4 method can be viewed as a predictor-corrector method as it first predicts a value in the mid-point and then corrects it to a better estimate at the next step. RK4 is a popular fourth-order method and involves four steps to advance from n to $n+1$. For the same step size Δt, RK4 is more efficient and accurate than the Euler scheme [18]. RK4 is a commonly used method as the trade-off between computational complexity (CPU time) and accuracy is preferable. The RK4 method has four stages and the integration order of the scheme is four [19]. The method is formulated as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \sum_{i=1}^{4} \mu_i \lambda_i \tag{2.6}$$

where

$$\begin{aligned} \lambda_1 &= f(\mathbf{x}_k, \mathbf{u}(t_k) \\ \lambda_2 &= f(\mathbf{x}_k + \frac{\Delta t}{2}\lambda_1, \mathbf{u}(t_k + \frac{\Delta t}{2})) \\ \lambda_3 &= f(\mathbf{x}_k + \frac{\Delta t}{2}\lambda_2, \mathbf{u}(t_k + \frac{\Delta t}{2})) \\ \lambda_4 &= f(\mathbf{x}_k + \Delta t\lambda_3, \mathbf{u}(t_k + \Delta t)) \end{aligned} \tag{2.7}$$

and

$$\mu_1 = \frac{1}{6},\ \mu_2 = \frac{1}{3},\ \mu_3 = \frac{1}{3},\ \mu_4 = \frac{1}{6} \tag{2.8}$$

The fully assembled RK step is then formulated as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t\Big(\frac{1}{6}\lambda_1 + \frac{1}{3}\lambda_2 + \frac{1}{3}\lambda_3 + \frac{1}{6}\lambda_4\Big) \tag{2.9}$$

2.2.2 Problem formulation

MPC is a control method in which the next control action is obtained by solving online, at each sampling instant, a finite horizon optimal control problem (OCP) where the current state of the plant is the initial state of the system. The OCP, $\mathbb{P}_\infty(x)$, is defined by

$$\begin{aligned} &\min_{u(.)} \mathcal{J}_\infty(x, u(.)) \\ &\text{st. } \dot{x} = f(x, u),\ x(0) = x_0 \\ &\quad (x(t), u(t)) \in \mathbb{Z} \text{ for all } t \in \mathbb{R}_{\geq 0} \end{aligned} \tag{2.10}$$

where $\mathcal{J}_\infty$ is the objective function or the cost function defined by

$$\mathcal{J}_\infty(x, u(.)) = \int_0^\infty \ell(x(t), u(t))dt \tag{2.11}$$

in which $x(t)$ and $u(t)$ satisfies $\dot{x} = f(x, u)$. The cost function quantifies the system's performance and typically includes terms related to the control objective such as tracking set points, deviations from desired states, minimizing control efforts, and constraints. Solving this optimization problem yields a finite control sequence and the first control action in this sequence is applied to the plant [15], i.e., only the control action computed for the first time step is implemented and the remainder of the solution is discarded. After this, the time horizon is shifted and a new open-loop optimal control is found for the next time horizon. The process is repeated every time step, which is why MPC also sometimes is referred to as receding horizon control (RHC) [6]. When there exists a solution to this problem it is denoted by $u^0_\infty(-; x)$ and the resulting optimal value function by $V^0_\infty(x)$.
As mentioned in 2.2.1.1 it is favorable to replace the continuous time differential equation with a discrete-time difference equation when searching for solutions to engineering problems. The discretization for both the state space equations and the objective defined in 2.10 can be made using both methods described in 2.2.1.1.

However, a general expression for the control of a constrained time-invariant system is described by the nonlinear difference equation

$$x^+ = f(x, u) \quad f : \mathbb{X} \times \mathbb{U} \to \mathbb{X} \tag{2.12}$$

As mentioned previously, another key ingredient in formulating MPC is the construction of an estimator to determine the evolution of the state [16],[15]. However, for the purpose of this thesis, a state estimator is not of the essence as the focus of the MPC lies on trajectory tracking. Hence the concept of defining a state estimator in an MPC is not further investigated.

2.2.3 Nonlinear Model Predictive Control

MPC is an optimal control strategy incorporating cost functions to find the optimal open-loop controller over a finite horizon. In nonlinear model predictive control (NMPC), a nonlinear mathematical plant is added to the cost functions to find the optimal control sequence that minimizes the resulting cost while satisfying any constraints [6]. NMPC extends the concept of MPC to handle nonlinear systems and uses nonlinear models, which can include time-varying parameters, for prediction and optimization. Unlike MPC, the constraints on the states and control inputs can be nonlinear, which allows for more accurate handling of system limitations. NMPC involves solving an OCP over a finite prediction horizon, however, unlike MPC, these problems are necessarily not convex, and as a result, NMPC can be computationally more expensive [20].

2.2.4 Obstacle avoidance techniques

There are various methods available for solving obstacle avoidance in MPC. These methods differ in their robustness, computational complexity, and suitability for specific scenarios. Not all methods will be covered but a selected few that seems suitable for the marine environment.

2.2.4.1 The repulsive potential function

The repulsive potential function P_k acts as a penalty term that discourages an MPC-controlled point from approaching a singularity point too closely. The potential function is typically defined in terms of the controlled point's position (x, y) and the position (x_0, y_0) of the obstacle and is given by the equation 2.13 [6].

$$P_k = \frac{1}{(x - x_0)^2 + (y - y_0)^2 + \epsilon} \tag{2.13}$$

Here, ϵ is a small positive number introduced to avoid singularity issues when (x, y) and (x_0, y_0) are getting too close. The value of ϵ determines the strength of the penalty; a smaller value of ϵ results in a stronger penalty, while a larger value of ϵ results in a weaker penalty. For linear obstacles, an approach presented by Abbas et al. [6] is to find the nearest point on the obstacle's line to the current position (x, y) and treat this point as the reference point (x_0, y_0) for the repulsive potential function P_k. Similar methods can be applied to other shapes.

2.2.4.2 Dynamic hyperplane

The Dynamic Hyperplane Method aims to separate two convex sets by dividing each set into a separate halfplane defined by a hyperplane, according to equation 2.14[21].

$$H(\lambda, \mu) = \{s \in \mathbb{R}^N \mid \lambda^\top s = \mu\} \tag{2.14}$$

Where s is a point in space, λ is the normal vector and μ is an offset. By taking two convex sets V_1 and V_2 described by a polygon with the points $V = [p_1, p_2 ... p_n]^T$ and $V_1 \cap V_1 = \varnothing$ then the dynamic hyperplane method defines two halfplanes that separate the two sets according to 2.15 - 2.17[21].

$$H^-(\lambda, \mu) = \{V_1 \in \mathbb{R}^{N*N_p} \mid \lambda^\top V_1 \leq \mu\} \tag{2.15}$$

$$H^+(\lambda, \mu) = \{V_2 \in \mathbb{R}^{N*N_p} \mid \lambda^\top V_2 \geq \mu\} \tag{2.16}$$

$$\lambda^\top V_1 \geq \mu^\top, \quad \lambda^\top V_2 \leq \mu^\top, \quad \|\lambda\| > 0. \tag{2.17}$$

By adding the offset μ as a decision variable to the existing decision variables in the OPC; the condition to separate the two sets from colliding holds if the OPC can find a hyperplane for each iteration. As seen by the showcase in Figure 2.4, by setting this condition as a constraint or a cost with a high penalty weight the OPC will be forced to find a solution that satisfies the dynamic hyperplane; navigating the controlled object in a collision-free path [21].

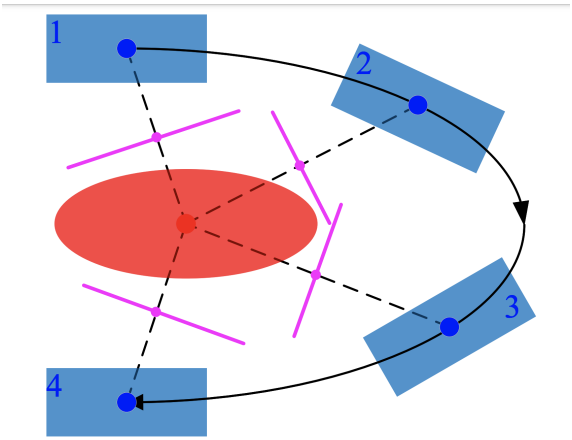


Figure 2.4: A showcase where a controlled object depicted in blue for an increasing iteration, avoids an object depicted in red and its corresponding dynamic hyperplane, depicted in magenta.
From: Efficient optimization-based trajectory planning [21].

2.3 Software architecture

In this section, the concept of an MPC solver is introduced. An MPC solver is a type of optimization tool designed to solve complex control problems. It formulates and

solves optimization problems in real time, making predictions of a dynamic system based on a mathematical model. Notably, there are several open-source MPC solvers available, each with unique strengths and weaknesses. For this study, two optimizers have been investigated: OpEn and IPOPT.

2.3.1 OpEn

OpEn is an open-source optimization engine designed specifically for dynamic, real-time, optimization in various applications, including robotic systems, autonomous vehicles, and unmanned aerial vehicles (UAVs) [22]. OpEn sets up the following optimization problem to minimize the given objective function while satisfying the specified constraints,

$$\begin{aligned} \mathcal{P}(p) : \text{Minimize} \quad & u \in \mathbb{R}^n \\ \text{subject to} \quad & f(u,p) \\ & u \in U \\ & F_1(u,p) \in C \end{aligned} \tag{2.18}$$

where $u \in \mathbb{R}^{n_u}$ represents the vector of decision variables of the problem, and $p \in \mathbb{R}^{n_p}$ is a vector of parameters, $F_1 : \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_1}$ is a smooth mapping and C is a closed convex set.
OpEn utilizes the Proximal Averaged Newton-Type Method (PANOC) for solving parametric nonconvex optimization problems [22]. PANOC is a line-search method that combines forward-backward iterations with fast Newton-type steps over the forward-backward envelope, which helps in shaping and directing the optimization process, ensuring convergence towards a solution while efficiently handling the constraints and objectives of the optimization problem [23].
When the code has been formulated in Python (or Matlab) OpEn will use a code generation tool to create Rust code to increase efficiency and make it more robust. Many potential runtime errors are caught at compile time, due to Rust's strict type system and borrow checker, leading to more robust and reliable code
A drawback of using OpEn is when the Rust code has been generated and later called, the number of parameters is set. This hinders the amount of obstacle parameters the OpEn solver can handle which could cause problems in an obstacle-filled environment.

2.3.2 IPOPT

IPOPT (Interior Point OPTimizer) is a widely used optimization solver which is included with the CasADI symbolic framework installation. IPOPT is used for optimizing problems shown in 2.19.

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_L \leq g(x) \leq g_U \\ & x_L \leq x \leq x_U \end{aligned} \tag{2.19}$$

x represents the optimization variables, $f(x)$ is the objective function, and $g(x)$ is the general nonlinear constraints. The bounds g_L, g_U, x_L, and x_U represent the lower and upper bounds for the constraints and variables, respectively [24].
IPOPT employs an interior point line search filter method to find a local solution to the nonlinear problem. This method combines primal-dual interior point techniques with a filter line search to handle both equality and inequality constraints effectively [24].

2.3.3 Solver comparison

Comparing the two solvers one main difference is the pre-set number of parameters in PANOC a problem that IPOPT doesn't have. But as the solver will be used in real-time with a dynamic environment the solver time will be the main factor for which optimizer to be used. A runtime comparison between IPOPT, PANOC, and another solver called SNOPT (Sparse Nonlinear OPTimizer), was made in the paper "A Penalty Method Based Approach for Autonomous Navigation using Nonlinear Model Predictive Control" [25].

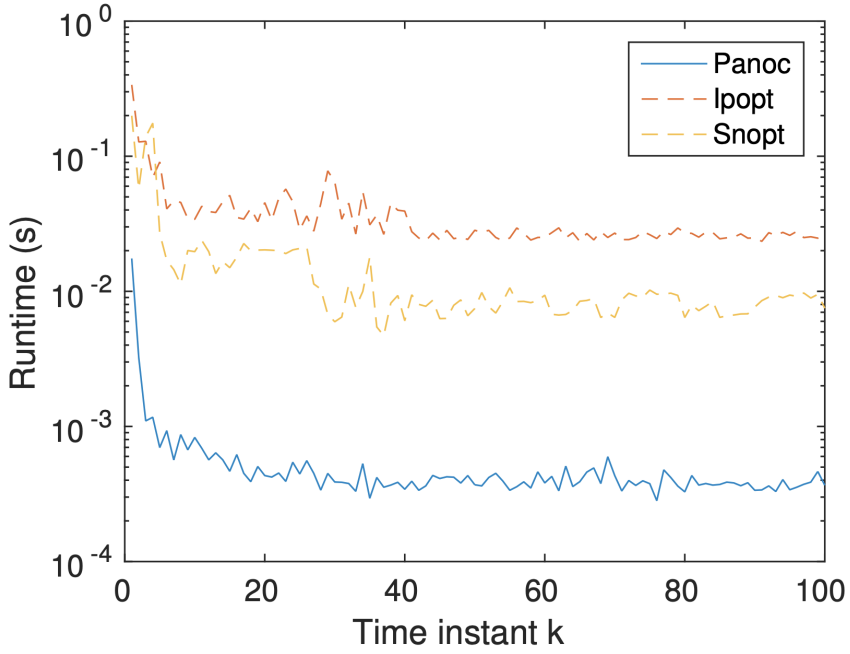


Figure 2.5: Solver-time comparison between IPOPT, PANOC, and SNOPT from: A Penalty Method-Based Approach for Autonomous Navigation using Nonlinear Model Predictive Control. [25]

The runtime comparison shown in Figure 2.5 clearly shows that the OpEn PANOC has a solver runtime that is faster than IPOPT by a factor around roughly 10^2.

3
Methods

This section details the approach taken to develop, implement, and evaluate the autonomous docking system. Which is composed of the implemented path planner, the design of the trajectory planner, and the simulation environments used for testing. It provides an overview of the techniques and processes employed to create a robust and reliable system for autonomous docking.

3.1 Path planning algorithm

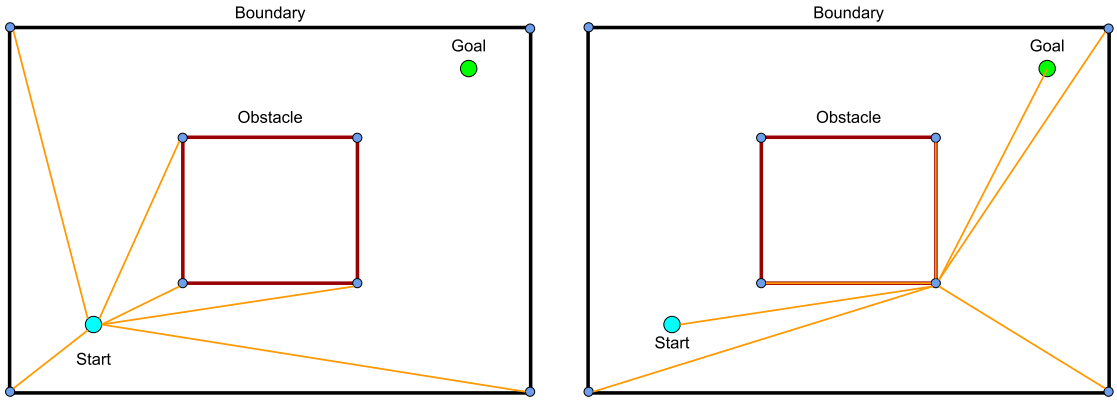
To maintain the algorithm's speed and predictability, the A* search algorithm was chosen over Dijkstra's algorithm and the dynamic window approach. A* offered a more efficient search strategy by incorporating heuristic information to guide the search towards the goal, resulting in faster pathfinding while ensuring optimality. The path planner implemented is an already constructed path planner from the ExtremityPathfinder library in Python [26]. The path planner navigates environments represented by polygons, applying a visibility graph optimized for shortest pathfinding. The environment for the ExtremityPathfinder was set up by using a boundary polygon $\mathcal{P}$ and obstacle polygons $\mathcal{O}_i$. To allow the ExtremityPathfinder to distinguish between boundary and obstacle polygons, the boundary polygon was oriented counter-clockwise, and the obstacles were oriented clockwise. The polygons were also defined to meet ExtremityPathfinder's requirements:

- The polygons can not self-intersect.
- Must not have consecutive identical vertices.
- The polygons must consist of at least three vertices.

The polygons were then stored in the following arrays.

$$\mathcal{P} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \tag{3.1}$$
$$\mathcal{O}_i = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\} \tag{3.2}$$
$$\text{boundary\_coordinates} = \mathcal{P} \tag{3.3}$$
$$\text{obstacles} = \{O_1, O_2, \ldots, O_k\} \tag{3.4}$$

The ExtremityPathfinder then proceeded by constructing a visibility graph $\mathcal{G}$. This graph connected all vertices that could directly see each other.



(a) All visible vertices from the starting node.

(b) All visible vertices from the obstacle's bottom left vertices.

Figure 3.1: Vertices seen from a) starting node and b) obstacle's bottom left vertex.

Figures 3.1a and 3.1b show what the visibility graph sees from the starting node and one of the obstacle's vertex. This was done on all vertices and each connection e_{ij} was weighted by an Euclidean distance $d(i, j)$ between the vertices v_i and v_j

$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

To compute the shortest path, the ExtremityPathfinder temporarily extends its structure to include the start and goal nodes. Visibility checks were performed from these nodes to existing graph nodes. If either the start or goal node had no visible connections, the path was deemed impossible. The A* algorithm was then employed to find the shortest path in the visibility graph $\mathcal{G}$, by using a heuristic based on Euclidean distance to prioritize nodes during the search, equation 2.1. If a path was found, the sequence of nodes (including the start and goal) was translated back into coordinate points and returned, and the total path length $\mathcal{L}$ was computed according to

$$\mathcal{L} = \sum_{i=1}^{k-1} d(v_i, v_{i+1}). \tag{3.5}$$

3.2 Trajectory planning using model predictive control

In MPC, a cost function is essential to quantify the performance of the control actions over a prediction horizon. The cost functions designed for a specific system guide the optimization process to determine the control action that minimizes costs, ensuring that the system behaves as desired. The final optimization problem integrates all cost functions into a single objective function, where the goal is to minimize this function subject to constraints and dynamics that ensure the boat's safe and efficient navigation.

The following section outlines the method for formulating the optimization problem within the MPC framework for an autonomous docking system with obstacle avoidance. The approach involves the development of a kinematic model describing the vessel's motion, the formulation of appropriate cost functions for various objectives, and the integration of these elements with set constraints of the system to a comprehensive optimization problem.

3.2.1 Vessel model

The first step towards using MPC for trajectory generation and obstacle avoidance for an autonomous boat involves developing a model of the boat's motion. A first strategy for this was to model the motion of the boat in the 2D plane by assuming that the boat's motion acts on a plane surface, i.e., neglecting roll and pitch. This means that the motion of the boat is assumed to be constant relative to the water surface. Environmental forces such as wind, waves, and currents are neglected when developing the model in the 2D plane. Although these factors may significantly affect the boat's motion in the real world, this simplification reduces the complexity of the model while still capturing the essential aspects of the boat's motion. For this thesis, the model of the boat is assumed to be purely kinematic, meaning that the motion of the boat is described without considering the forces and torques that cause this motion. The model only focuses on the geometry of the motion, and by doing so, ideal conditions can be assumed by neglecting friction, drag, or other forces.
The motion of the marine vessel is represented by the pose vector

$$\boldsymbol{\eta} = \begin{bmatrix} x & y & \psi \end{bmatrix}^\top \in \mathbb{R}^2 \times \mathbb{S}, \tag{3.6}$$

which also represents the states of the system and the velocity vector

$$\boldsymbol{\nu} = \begin{bmatrix} u & v & r \end{bmatrix}^\top \in \mathbb{R}^3 \tag{3.7}$$

i.e., the control inputs of the systems. Here (x, y) represents the position of the vessel in the earth-fixed reference frame, ψ represents the orientation around the z-axis, the yaw angle, (u, v) represents the body-fixed linear velocities, surge, and sway, and r is the yaw rate, see Figure 3.2.
Using these notations, a 3-DOF kinematic vessel model can be described as

$$\dot{\boldsymbol{\eta}} = \mathcal{R}(\psi)\boldsymbol{\nu} \tag{3.8}$$

where $\mathcal{R}(\psi)$ is the rotational matrix given by

$$\mathcal{R}(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.9}$$

The final kinematic model is given by

$$\dot{\boldsymbol{\eta}} = \mathcal{R}(\psi)\boldsymbol{\nu} = \begin{bmatrix} u\cos(\psi) - v\sin(\psi \\ u\sin(\psi) + v\cos(\psi) \\ r \end{bmatrix}. \tag{3.10}$$

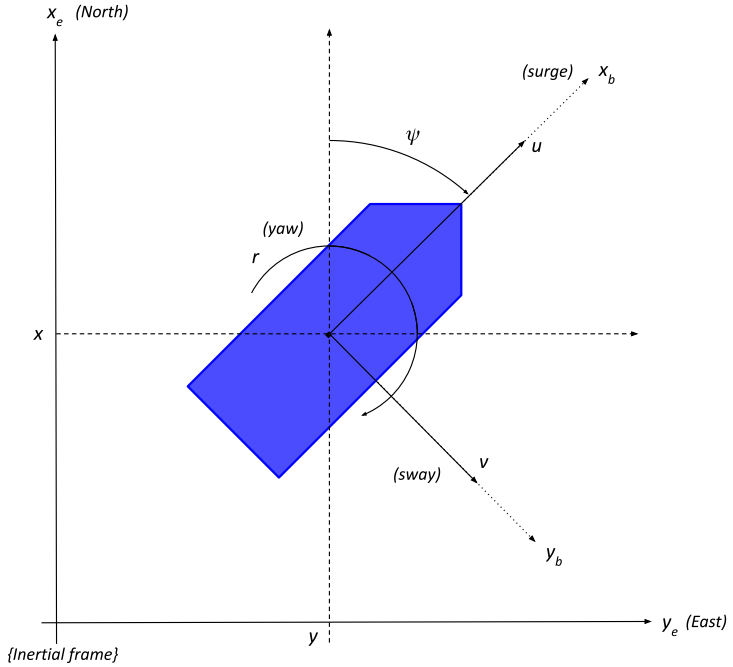


Figure 3.2: Varibles used to describe the motion of a boat in the horizontal plane. (x, y) - boat position coordinates, (u, v) - linear body fixed velocities (surge, sway) ψ - yaw angle, r - yaw rate.

3.2.2 Collision avoidance

To ensure safe maneuvering while driving toward the docking position, a safe collision avoidance technique is of high importance, both for avoiding static and dynamic obstacles but also for staying within the harbor environment.

To ensure a safe distance between the boat and the obstacles or the environment's boundaries, padding was added to each obstacle and boundary, see Figure 3.3.

The padded obstacle is then used as the real obstacle when planning the path and trajectory of the boat. The distance from the boat to each obstacle is calculated from the boat's center to the edges of each padded obstacle. This means that the size of the padding is determined by the longest distance from the center of the boat to the edge of the boat to ensure that the padding is large enough to avoid collision when driving close to an obstacle, see Figure 3.4a. However, to ensure that the least amount of drivable space gets removed by padding, two extra reference points were added to the boat according to Figure 3.4b, to minimize the size of the padding. There is a possibility to add more reference points depending on the size of the boat and to minimize the padding further. However, to minimize the computation time it is more suitable to have as few reference points as possible such that the padding covers the entire boat while also removing the least amount of drivable space. For this thesis, three reference points were deemed suitable.

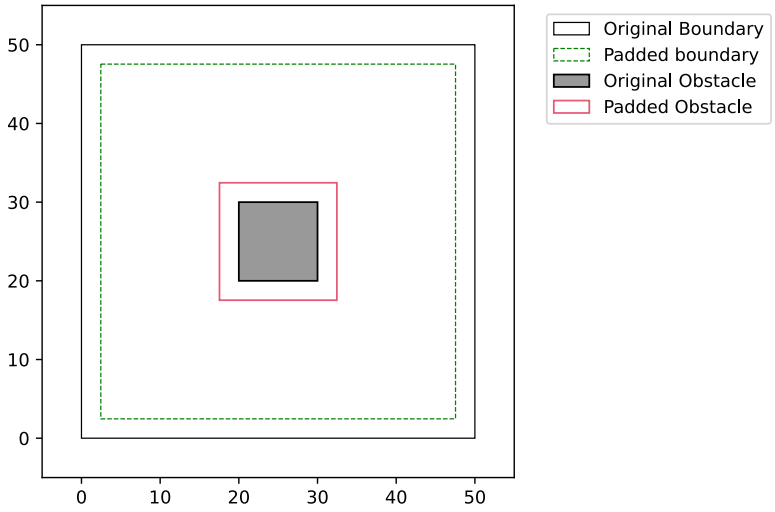


Figure 3.3: Example of environment displaying padding of obstacle and boundary.

3.2.2.1 Static obstacles & boundaries

Obstacles can be interpreted in many different ways but are, in this thesis, modeled as convex polygons with a maximum of four vertices. Each polygon obstacle is represented by a set of half-spaces in $\mathbb{R}^2$ according to

$$H_n = \{\boldsymbol{p} \in \mathbb{R}^2 : \boldsymbol{b}_{n,m} - \boldsymbol{a}_{n,m}^T \boldsymbol{p} > 0,\ m \in \mathbb{N}_{[1,M]}\} \tag{3.11}$$

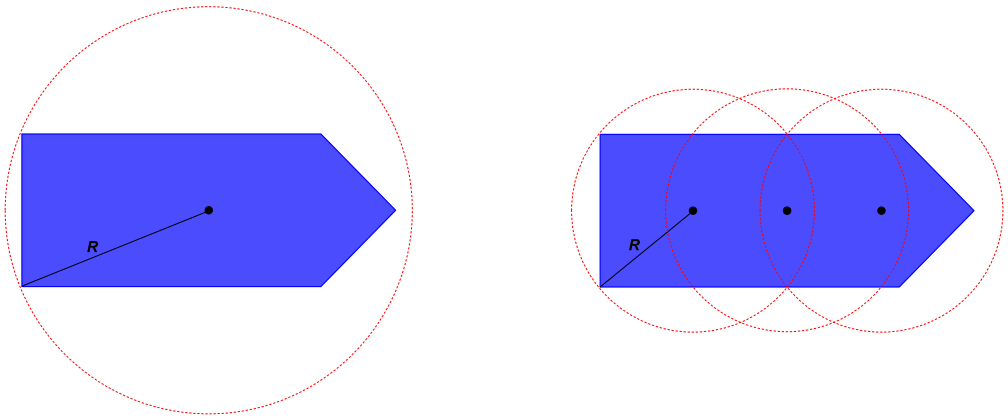
where each vector $\boldsymbol{a}_{n,m}$ and scalar $\boldsymbol{b}_{n,m}$ defines a half-space and the constraint H_n represents the intersection of these half-spaces. In this expression, M defines the number of inequalities forming the polygon, i.e., one inequality for each half-space, n denotes the n-th obstacle, and $\boldsymbol{p}$ represents a point in the two-dimensional space with coordinates (x, y) which is the position of the boat. The constraint H_n is defined as the collection of points $\boldsymbol{p}$ such that the condition $h_{n,m} = \boldsymbol{b}_{n,m} - \boldsymbol{a}_{n,m}^T \boldsymbol{p} > 0$ holds for the finite set of M inequalities. Hence, for a point $\boldsymbol{p}$ to not be inside an obstacle, $\boldsymbol{p} \notin H_n$, constraint 3.12 must hold.

$$\forall n \in \mathbb{N}_{[1,N_o]},\ \prod_{m=1}^{M} [h_{n,m}(\boldsymbol{p})]_+^2 = 0 \tag{3.12}$$

Based on this constraint, a soft obstacle avoidance cost with penalty $Q_{\mathcal{O}}$ is defined according to

$$\mathcal{J}_{\mathcal{O}}(\boldsymbol{p}) = Q_{\mathcal{O}} \sum_{n=1}^{N_o} \prod_{m=1}^{M} [h_{n,m}(\boldsymbol{p})]_+^2 \tag{3.13}$$

meaning that if, a point $\boldsymbol{p}$ is inside an obstacle such as in Figure 3.5, the cost $\mathcal{J}_{\mathcal{O}}$ will increase.
Since additional reference points were added to the boat to minimize the padding of each obstacle, these points were also used when calculating the obstacle avoidance



(a) One reference point. **(b)** Three reference points.

Figure 3.4: Example of how reference points added at the center of the boat are used for collision avoidance to minimize the size of the padding of each obstacle. The radius R of the red circles represents the size of the padding.

cost. This means the final obstacle avoidance cost will be the sum of the cost for all three points.

$$\mathcal{J}_{\mathcal{O}} = \sum_{i=1}^{3} \mathcal{J}_{\mathcal{O}}(\boldsymbol{p}_i) \tag{3.14}$$

For the MPC, a boundary region is also defined as the region in which the docking procedure will take place, i.e., the boundary of the harbor environment. Meaning that the boat should always stay inside the boundaries when in autonomous docking mode. The boundary region is also defined as a convex polygon with four vertices, hence the boundary constraint and cost are defined similarly to the constraint and cost for the static obstacle avoidance but in the opposite way. For a point $\boldsymbol{p}$ in the boat to inside all M_B half-spaces of the boundary region constraint 3.15 must hold.

$$\sum_{m=1}^{M} [h_m(\boldsymbol{p})]_-^2 = 0 \tag{3.15}$$

The cost for the boat being outside the boundaries can then be formulated as

$$\mathcal{J}_{\mathcal{B}}(\boldsymbol{p}) = Q_{\mathcal{B}} \sum_{m=1}^{M} [h_m(\boldsymbol{p})]_-^2 \tag{3.16}$$

meaning that if, a point $\boldsymbol{p}$ in the boat is outside the boundary region, the cost $\mathcal{J}_{\mathcal{O}}$ will increase with a scale of the penalty weight $Q_{\mathcal{B}}$. Similar to the static obstacle avoidance cost, the cost of being outside of the boundaries is calculated for all reference points in the boat, giving the final cost as

$$\mathcal{J}_{\mathcal{B}} = \sum_{i=1}^{3} \mathcal{J}_{\mathcal{B}}(\boldsymbol{p}_i). \tag{3.17}$$

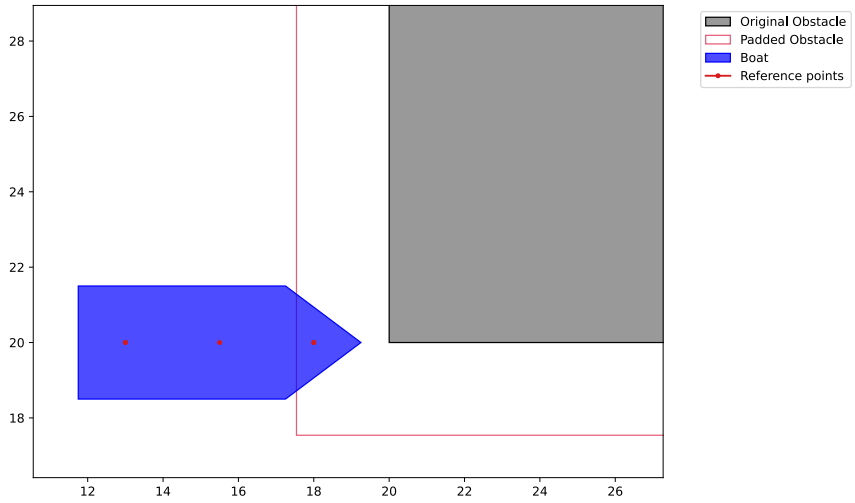


Figure 3.5: Example of when the cost $\mathcal{J}_{\mathcal{O}}$ will increase. The red dots represent the three reference points aligned at the boat's center, if one or more reference points are inside the padded obstacle, the cost will increase.

3.2.2.2 Dynamic obstacles

Incorporating dynamic obstacle avoidance in the MPC is crucial for ensuring safe navigation in dynamic environments, such as busy harbors. Dynamic obstacles, similar to static obstacles, can be modeled using various methods. In this thesis, they are represented as ellipses due to the simplicity of their mathematical representation, making them easy to incorporate into optimization problems and hence favorable in the context of optimization in autonomous systems.
The ellipse is defined as the implicit function

$$e(\boldsymbol{p}) = \left[1 - (\boldsymbol{p} - \mathbf{c})^\top E(\boldsymbol{p} - \mathbf{c})\right]_+^2 \tag{3.18}$$

where $\boldsymbol{p}$ is the current position, (x, y) and $\mathbf{c}$ is the center of the ellipse. E is a positive-definitive matrix that encodes the shape and orientation of the ellipse according to

$$E = \mathcal{R}(\phi)^\top \begin{bmatrix} \frac{1}{a^2} & 0 \\ 0 & \frac{1}{b^2} \end{bmatrix} \mathcal{R}(\phi) \tag{3.19}$$

with semi-major axis a, semi-minor axis b, rotation angle ϕ, and where $\mathcal{R}(\phi)$ is the rotation matrix

$$\mathcal{R}(\phi) = \begin{bmatrix} cos(\phi) & -sin(\phi) \\ sin(\phi) & cos(\phi). \end{bmatrix} \tag{3.20}$$

This definition is made to facilitate the integration of a cost function for obstacle avoidance. To ensure safe navigation in a busy harbor with dynamic obstacles, the cost function is defined to penalize the states of the boat that are too close or inside

a dynamic obstacle. By using the implicit function 3.18, the cost can be formulated as

$$\mathcal{J}_{\mathcal{D}}(\boldsymbol{p}) = \sum_{n=1}^{N_D} \mathcal{Q}_{\mathcal{D}} \left[1 - (\boldsymbol{p} - \mathbf{c})^\top E(\boldsymbol{p} - \mathbf{c})\right]_+^2 \tag{3.21}$$

where N_D is the number of dynamic obstacles and $[.]_+^2$ denotes the non-negative part of the expression, ensuring that only values less than one contribute to the cost.
As for the static obstacles and boundaries, the dynamic obstacle avoidance cost is calculated for all three reference points in the boat, giving a final cost

$$\mathcal{J}_{\mathcal{D}} = \sum_{i=1}^{3} \mathcal{J}_{\mathcal{D}}(\boldsymbol{p}_i). \tag{3.22}$$

3.2.3 Reference path

To make sure that the boat follows the predetermined reference path given by the path planner, the deviation from this path can be penalized using a soft cost function in the MPC. This cost can be formulated in several ways, for example by penalizing the difference between the boat's current position to the nearest waypoint on the reference path or by using a cross-track error (CTE) which is the method used in this thesis. CTE is chosen since it directly measures the lateral deviation from the reference path, simplifying the cost function by enhancing precision and stability. The CTE is defined to minimize the deviation of the boat from the desired reference path. The cost function with penalty Q_{CTE} is given by

$$\mathcal{J}_{CTE}(\boldsymbol{p}) = Q_{CTE} \cdot \left(\min(\| \boldsymbol{p} - s_1 - t^\star(s_2 - s_1) \|^2)\right), \tag{3.23}$$

where $\boldsymbol{p}$ is the current position of the boat in the two-dimensional space with coordinates (x, y). Unlike the collision avoidance costs, only the current position of the boat, i.e., the reference boat at the center of the boat is used for the CTE. s_1 and s_2 represents two consecutive points on the reference path and $t^\star$ is the optimal interpolation parameter along the line segment between s_1 and s_2 that minimizes the perpendicular distance to the boat's position. To find $t^\star$,the projection of the point $\boldsymbol{p}$ onto the line segment between s_1 and s_2 was first computed according to

$$\hat{t} = \frac{(\boldsymbol{p} - s_1) \cdot (s_2 - s_1)}{\| s_2 - s_1 \|^2 + \epsilon} \tag{3.24}$$

where ϵ is a small value to avoid dividing by zero. To find the t that minimizes the distance on the segment, $\hat{t}$ is projected back to $[0, 1]$ using $t^\star = \min(\max(\hat{t}, 0), 1)$. The perpendicular distance from $\boldsymbol{p}$ to the line segment, i.e. the minimum distance, is then computed as $\mathbf{d} = s_1 + t^\star(s_2 - s_1) - \boldsymbol{p}$ and the cross-track error cost is calculated as 3.23 with a penalty weight of Q_{CTE}.
In summary, the CTE is calculated by iterating through the waypoints on the reference path, projecting the boat's position onto each path segment, and finding the minimum squared distance to any segment. An example of this can be viewed in Figure 3.6. This distance is then used to compute cost, which is weighted by the

penalty Q_{CTE}. This will ensure that the boat stays close to the desired reference path, minimizing the CTE.

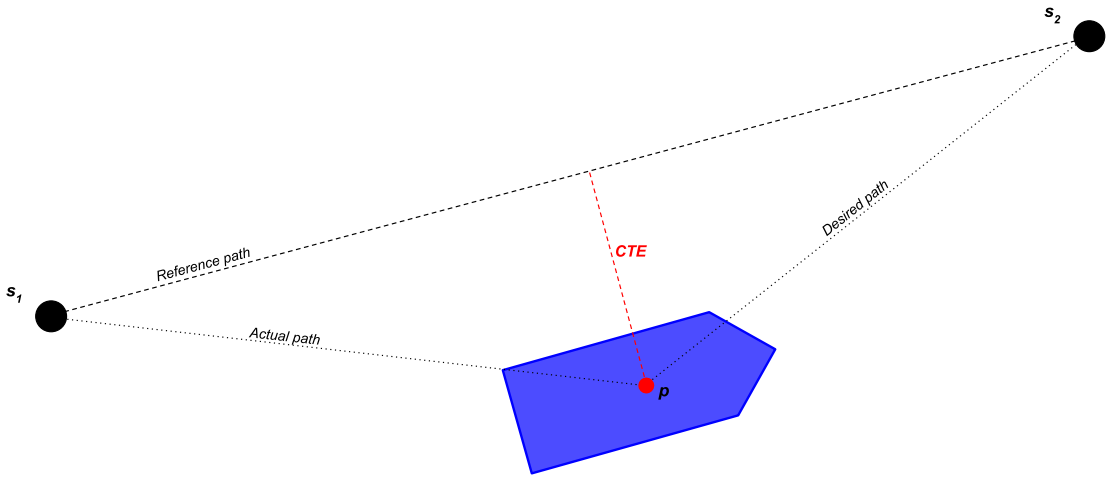


Figure 3.6: Example image of how the cross-track error is defined.

3.2.4 Control action & initial acceleration

To follow the given path with smooth behaviors, the objective function is defined by several cost functions as explained in the sections above. To ensure smooth transitions cost functions penalizing the deviations from the original trajectory, deviating from the original velocity and acceleration is necessary. Deviations from the original trajectory are handled with the cross-track error as explained in 3.2.3. Deviations from the original velocity can be penalized by adding a cost to the control actions. The parameters that the MPC controls are the motions that the marine vessel is capable of exhibiting in multiple directions, surge (u), sway (v), and yaw rate (r). To ensure safe and comfortable maneuvering, a reference control input is sent to the MPC to ensure that the boat will keep the desired velocities at each instance. For this to happen, a cost function for the control action is designed to penalize deviations from the reference control input. This makes the controller follow a desired control trajectory closely. The control action cost with penalty weight r and horizon N is defined by

$$\mathcal{J}_\nu = r \cdot \parallel \tilde{\boldsymbol{\nu}}_k - \boldsymbol{\nu}_k \parallel^2, \quad k \in \mathbb{N}_{[0,N-1]} \tag{3.25}$$

where $\boldsymbol{\nu}$ is the control input, $[u, v, r]^\top$, and $\tilde{\boldsymbol{\nu}}$ is the reference input, $[\tilde{u}, \tilde{v}, \tilde{r}]^\top$, i.e., the desired control action the system aims to follow. The control action cost is crucial for tasks that require precise control, such as trajectory tracking or maintaining stability. It also helps ensure smooth control actions by penalizing large deviations from the reference control input.

To follow the given trajectory with smooth behaviors, the objective function is also defined by a cost function that penalizes the acceleration:

$$\mathcal{J}_{\mathcal{A}} = \mathcal{Q}_a \cdot \parallel \boldsymbol{\nu}_{k+1} - \boldsymbol{\nu}_k \parallel^2, \quad k \in \mathbb{N}_{[0,N-1]} \tag{3.26}$$

where $\boldsymbol{\nu}_k$ represent the control input at time step k, and $\boldsymbol{\nu}_{k+1}$ represents the control input at subsequent time step $k+1$. $\mathcal{Q}_a$ is the penalty weight matrix, $[q_{au}, q_{av}, q_{a\psi}]$, that penalizes the changes in the control inputs, the acceleration, and N is the prediction horizon. This cost function penalizes changes in control inputs across three directions, the longitudinal, lateral, and rotational acceleration, a_u, a_v, and a_r, and is essential for providing smooth transitions and reducing abrupt changes in control inputs.

3.2.5 Final state

Reaching the final docking position with the right orientation is crucial in a docking scenario. Hence, a cost function is formulated to ensure the boat reaches its desired docking position with minimal positional and orientation errors. The terminal positions cost function with penalties $\mathcal{Q}_{\mathcal{N}}$ and $\mathcal{Q}_{\psi\mathcal{N}}$ is given by

$$\mathcal{J}_{\mathcal{F}} = \mathcal{Q}_{\mathcal{N}} \left((x - x_f)^2 + (y - y_f)^2 \right) + \mathcal{Q}_{\psi\mathcal{N}} (\psi - \psi_f)^2 \tag{3.27}$$

where x and y are the current position and ψ is the current orientation (heading angle) of the boat. The final state, i.e. the desired terminal position and the desired terminal orientation is x_f, y_f, and ψ_f. $\mathcal{Q}_{\mathcal{N}}$ is the penalty associated with the positional deviation and $\mathcal{Q}_{\psi\mathcal{N}}$ with the final orientation deviation and by appropriately tuning these weights, one can balance the importance of the positional and orientation accuracy to the requirements of the control task.

3.2.6 Prioritizing forward drive

A marine vessel is capable of exhibiting motion in multiple directions, including longitudinal (surge), lateral (sway), and rotational (yaw) movements. This encompasses forward and backward propulsion, lateral translation, and rotation around its vertical axis. To ensure a comfortable and safe drive toward the docking position and before the docking procedure starts, forward or backward propulsion is preferred, i.e., motion in the longitudinal direction (surge). To make sure that this movement is prioritized before the docking procedure starts, a cost function, $\mathcal{J}_v$, is designed to ensure that the vessel minimizes lateral acceleration, which in turn prioritizes forward drive stability and smoothness. The cost function can be expressed as

$$\mathcal{J}_v = \mathcal{Q}_v \cdot a_v^2 = \mathcal{Q}_v \cdot \left(\frac{v_k - v_{k-1}}{\Delta t}\right)^2, \tag{3.28}$$

where $\mathcal{Q}_v$ is the penalty weight determining the importance of minimizing the lateral acceleration, a_v is the lateral acceleration which is based on the change in control input over a given time step, i.e. the current control input in the lateral direction, the lateral velocity v_k at time step k, and the previous control input in the lateral direction v_{k-1} at time step $k-1$. Δt is the time step between the current and previous control input. By penalizing the lateral acceleration, the control strategy encourages the boat to maintain a straight and stable trajectory, enhancing both control and safety.

3.2.7 Optimization problem

The objective function is the overall cost function to be minimized and will include the terms described in sections 3.2.2-3.2.6. Besides the definition of the objective function for the optimization problem, a set of hard constraints for the system is also necessary to define the optimization problem. These constraints include the vessel dynamics, the maximum and minimum velocities of the vessel, and the initial and final state of the system, i.e., the start and docking position.

The complete optimization problem with the objective function and constraints is formulated as:

$$\min_{\boldsymbol{\nu}_{[0:N-1]}} \sum_{k=0}^{N-1} [\mathcal{J}_\mathcal{O} + \mathcal{J}_\mathcal{B} + \mathcal{J}_\mathcal{D} + \mathcal{J}_{CTE} + \mathcal{J}_{\boldsymbol{\nu}} + \mathcal{J}_\mathcal{A} + \mathcal{J}_\mathcal{F} + \mathcal{J}_v] \tag{3.29a}$$

$$\text{s.t. } \forall k \in \mathbb{N}_{[0,N-1]}, \tag{3.29b}$$

$$\boldsymbol{\eta}_{k+1} = \boldsymbol{\eta}_k + \frac{\Delta t}{6}(\lambda_1 + 2\lambda_2 + 2\lambda_3 + \lambda_4) \tag{3.29c}$$

$$\text{with } \lambda_1 = f(\boldsymbol{\eta}_k, \boldsymbol{\nu}_k) \tag{3.29d}$$

$$\lambda_2 = f(\boldsymbol{\eta}_k + \frac{\Delta t}{2}\lambda_1, \boldsymbol{\nu}_{k+\frac{1}{2}}) \tag{3.29e}$$

$$\lambda_3 = f(\boldsymbol{\eta}_k + \frac{\Delta t}{2}\lambda_2, \boldsymbol{\nu}_{k+\frac{1}{2}}) \tag{3.29f}$$

$$\lambda_4 = f(\boldsymbol{\eta}_k + \Delta t\lambda_1, \boldsymbol{\nu}_{k+1}) \tag{3.29g}$$

$$\text{and } f(\boldsymbol{\eta}, \boldsymbol{\nu}) = \mathcal{R}(\psi)\boldsymbol{\nu} \tag{3.29h}$$

$$\eta_0 = \eta_{initial}, \quad \eta_N = \eta_{final} \tag{3.29i}$$

$$\boldsymbol{\nu}_{min} \leq \boldsymbol{\nu} \leq \boldsymbol{\nu}_{max}. \tag{3.29j}$$

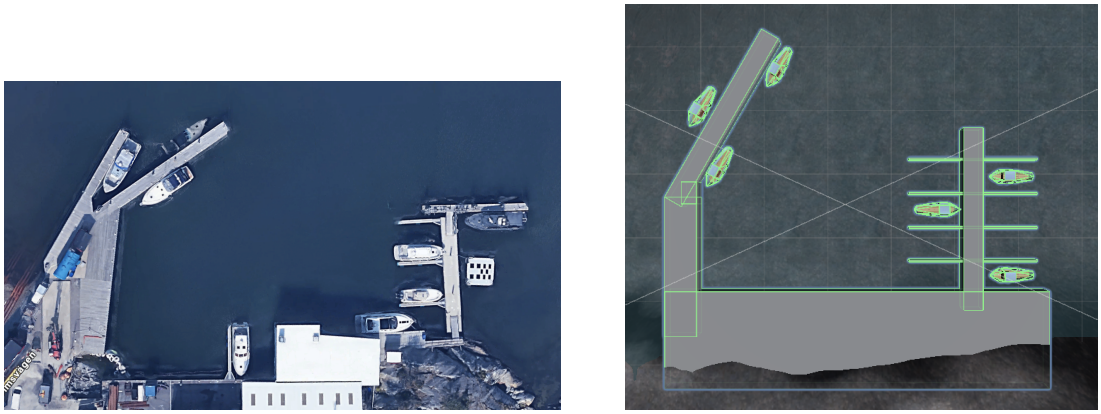
3.3 Environment setup

For simulations, two different environments were used and the OpEn solver was selected for its fast computational time, shown in Figure 2.5. One environment was implemented in Python, where all data was confined within the Python simulations.

The other environment was implemented in Unity, controlled by a proportional controller in a twin Electronic Vessel Control (EVC) installation rig. The twin EVC installation rig is an external rig used to steer and simulate a vessel's operations. In the Unity environment, the data was not confined to the Python simulation but was instead read and sent via CAN buses between the twin EVC installation rig and the connected computer.

3.3.1 Simulation in Python environment

To achieve an accurate environment representation, the Python environment was mapped based on the simulation environment created in Unity. This Unity environment is itself a simplified model of the real test harbor at Krossholmen, depicted in Figure 3.7.



(a) From: Google maps. Image over Krossholmen harbor.

(b) Illustration of the harbor in Unity environment.

Figure 3.7: Harbor comparison between a) actual harbor at Krossholmen and b) Unity.

Since the Unity environment consists of convex blocks, the corners of these blocks were translated into polygons within the Python environment, while trying to keep a 1:1 scale. This process was accomplished using the Python Matplotlib library.
Figure 3.8 shows the Python environment mapped out after the simulation harbor in Unity. The negative offset is to keep the same position relative to the harbor in Unity's environment.

3.3.1.1 Python environment - MPC connection

In the Python environment, simulations maintained a one-to-one connection with the MPC in and output. Where the vessel's next position and orientation are directly based on the MPC's output. The updated state is then the next input to the MPC model as seen in Figure 3.9.

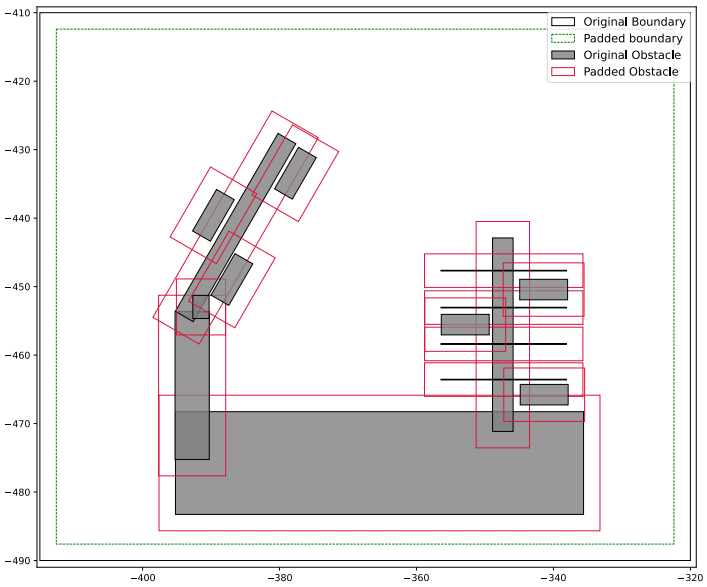


Figure 3.8: Mapped-out replica after the Unity's simulation environment.

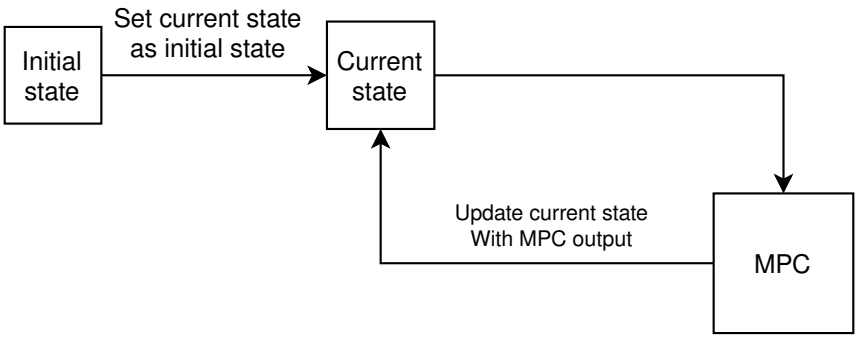


Figure 3.9: Block scheme of Python environment

The loop in 3.9 is continued until the current state has reached the final state, in other words, the docking position.

3.3.2 Simulation in Unity

The simulations in Unity had a more realistic setup, with both the vessel and environment dynamically modeled to behave similarly to a real vessel in water. The Unity simulations were interfaced with a twin EVC installation rig which in turn had CPAC's Assisted docking program integrated, which controlled the vessel's states. As the twin EVC installation rig operated using latitude and longitude coordinates and the MPC utilized cartesian coordinates, any data exchanged between the twin EVC installation rig and the MPC was converted with the equations 3.30 - 3.33.

$$\text{lat} = lat_{ref} + \left(\frac{y}{R}\right)\left(\frac{180}{\pi}\right) \tag{3.30}$$

$$\text{long} = long_{ref} + \left(\frac{x}{R\cos(lat_{ref} \cdot \frac{\pi}{180})}\right)\left(\frac{180}{\pi}\right) \tag{3.31}$$

$$x = (\text{long} - long_{ref}) \cdot \left(\frac{\pi}{180}\right) \cdot R \cdot \cos(lat_{ref} \cdot \frac{\pi}{180}) \tag{3.32}$$

$$y = (\text{lat} - lat_{ref}) \cdot \left(\frac{\pi}{180}\right) \cdot R \tag{3.33}$$

In Equations 3.30 - 3.33 lat_{ref} and $long_{ref}$ are the reference latitude and longitude values, used as fixed points for converting between Cartesian coordinates and geographic coordinates. R is the radius of the Earth in meters.

The communication between the MPC and twin EVC installation rig was made over a CAN bus, which was effective but may round off small data and lead to data loss during bit conversion. To address this issue a rounding sum was added to the conversion equation to preserve the accuracy of small values when data was transmitted over CAN. The conversion equation is shown in the equation 3.34.

$$\text{bitdata} = (\text{data} \cdot \text{scale} - \text{offset}) + \text{sgn}(\text{data} \cdot \text{scale} - \text{offset}) \cdot \frac{\text{scale}}{2} \tag{3.34}$$

In the equation, 3.34 data represents the value to be converted and transmitted over CAN. Scale and offset are parameters to ensure proper conversion, and sgn is used to determine a value's positive/negative sign.

The rounding conversation was needed due to the twin EVC installation rig operated with a proportional controller, which minimized the error between the set and the actual state. Precise control was needed in an obstacle-filled environment where small vessel movements were necessary for safe navigation. The converted values were then sent over the CAN bus as shown in Figure 3.10.

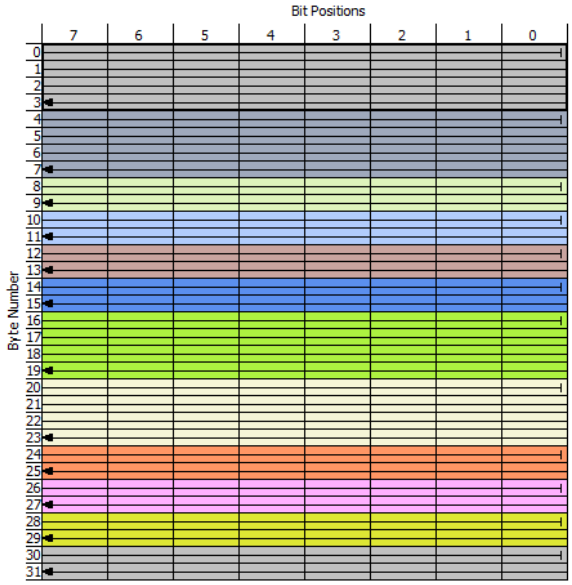


Figure 3.10: CAN bus architecture for sending MPC data. Read from right to left and state values are separated by colors.

The values sent over the CAN bus are from the top down in Figure 3.10, and each set point value is represented in that order in table 3.1 below. Zero indicates the current value of the vessel and one is the next value.

Table 3.1: Setpoint values for the states and control signals sent over the CAN-bus. 0 indicates the current state and control signal of the boat and 1 indicates the next optimum value of the state and control signal.

LatSetPoint0
LongSetPoint0
HeadingSetPoint0
LatSpeedSetPoint0
LongSpeedSetPoint0
YawRateSetPoint0
LatSetPoint1
LongSetPoint1
HeadingSetPoint1
LatSpeedSetPoint1
LongSpeedSetPoint1
YawRateSetPoint1

3.3.2.1 Unity environment - MPC connection

A distinction from the Python simulations was that the MPC did not maintain a one-to-one connection. The input state to the MPC was the actual state of the simulated vessel, read via CAN bus, see Figure 3.11. The optimal state derived from the MPC output did not directly move the current state of the vessel to the set

Figure 3.12: The Unity environment setup

state. The optimal state was sent over CAN to the twin EVC installation rig and it was the proportional controller that minimized the state error between the current and set state. After a predefined time step, during which the proportional controller should have adjusted the vessel to the desired states, a new MPC input was read over CAN from the vessel's actual states. The entire simulation setup is shown in Figure 3.12, from left to right in the Figure, it includes the Unity environment, the auxiliary Python environment that follows the Unity environment, and the twin EVC installation rig.

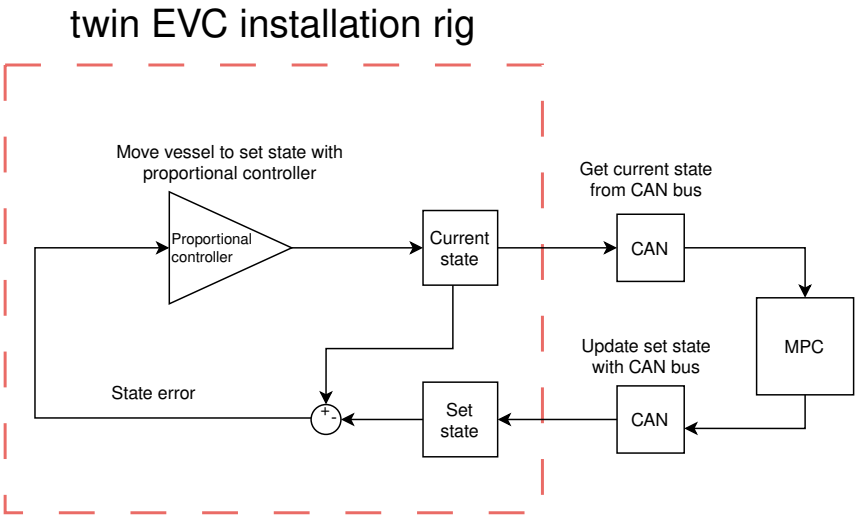
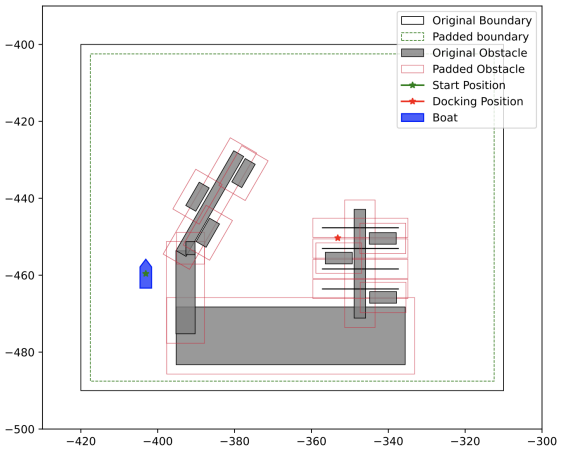


Figure 3.11: Block scheme of Unity environment

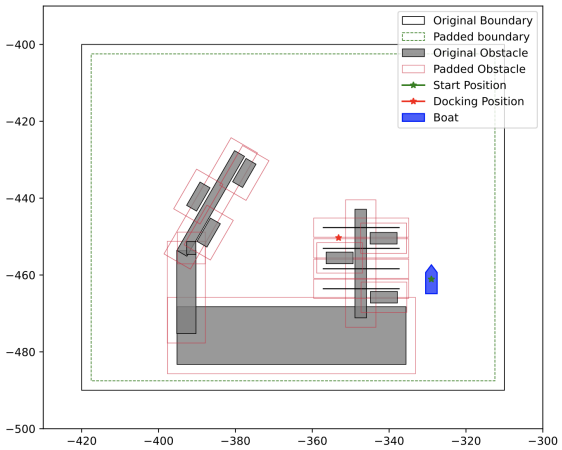
3.4 Simulation setup

To test the automatic docking algorithm, four different docking scenarios were prepared for both simulation environments, seen in Figures 3.13. Each scenario featured

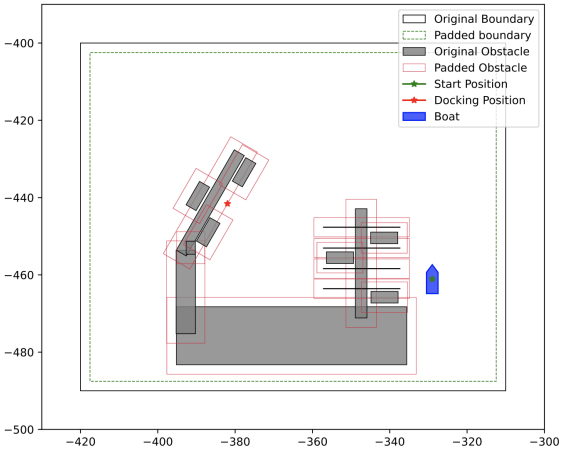
a distinct starting or docking point with varying orientations. All four scenarios were tested in both simulation environments. An additional scenario involved a dynamic obstacle that was only tested in the Python environment. The absence of simulating the last scenario with the dynamic obstacle in the Unity environment was due to time constraints limiting the implementation of dynamic obstacle simulation in the Unity environment.



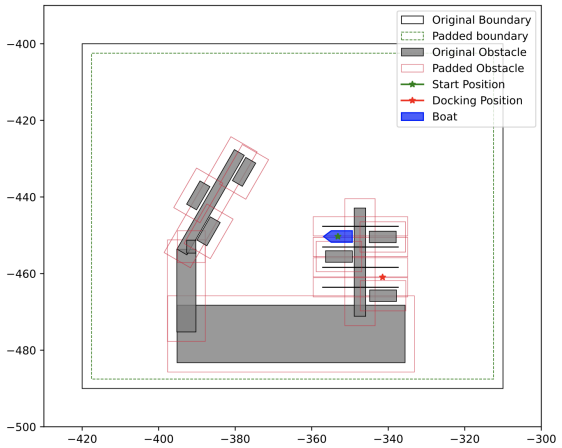
(a) Scenario 1: Drive into the harbor from the left and dock with 180° orientation.



(b) Scenario 2: Drive into the harbor from the right and dock with 180° orientation.



(c) Scenario 3: Drive into the harbor from the right and dock with 60° orientation.



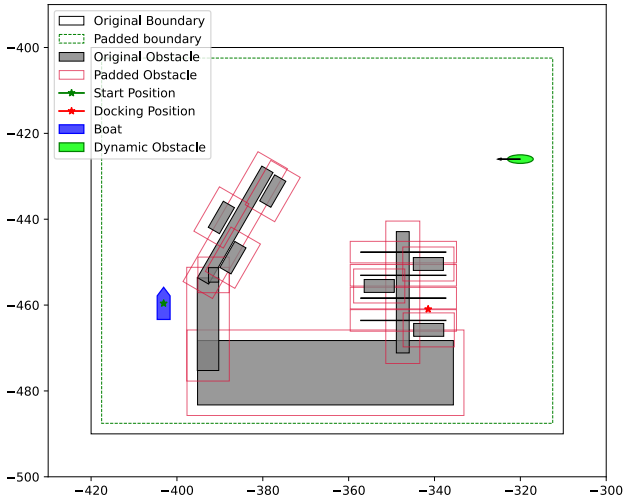
(d) Scenario 4: Redock from a 180° orientation to another docking slot at 0° orientation.

Figure 3.13: Four different docking scenarios a)-c), tested on both Python and Unity environments.

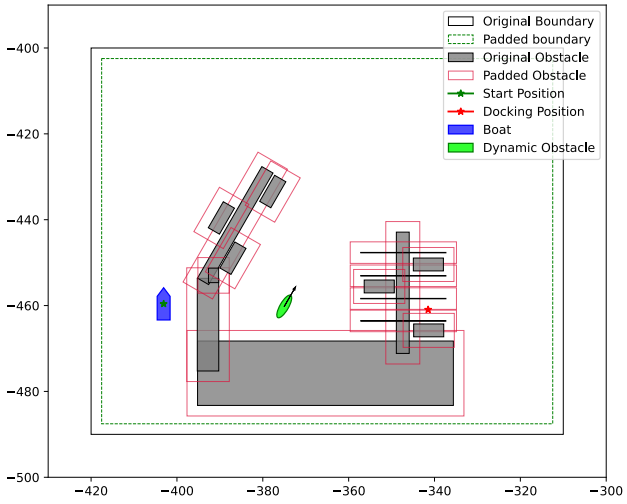
The fifth and final docking scenario involved incorporating dynamic obstacles in the environment. For this case, two different scenarios were simulated where the dynamic obstacle entered from two different positions and angles, see Figure 3.14.

3.5 Real-world testing procedure

To validate the functionality, reliability, and performance of the autonomous docking system in real-world conditions, actual testing of the algorithms on real boats is imperative. This section outlines the methodology employed for conducting real-life tests on the boat in a real-world environment, ensuring that the system meets the requirements of practical maritime applications. The system tests were performed on a Cranchi Endurance 41 with Volvo Penta D6 engines. A map over the Krossholmen harbor at Volvo Pentas marine test center was predefined as a polygon environment based on measurements and positions taken from Google Earth and Maps. Reviewing Figure 3.7 shows that the simulation environment mapped of the harbor created in Unity is not a replica of the environment at Krossholmen. Hence, a new polygon environment of the Krossholmen harbor had to be designed to ensure that the polygon environment's size and measurements coincide with the real environment at Krossholmen. Figure 3.15 shows this environment and the actual harbor.

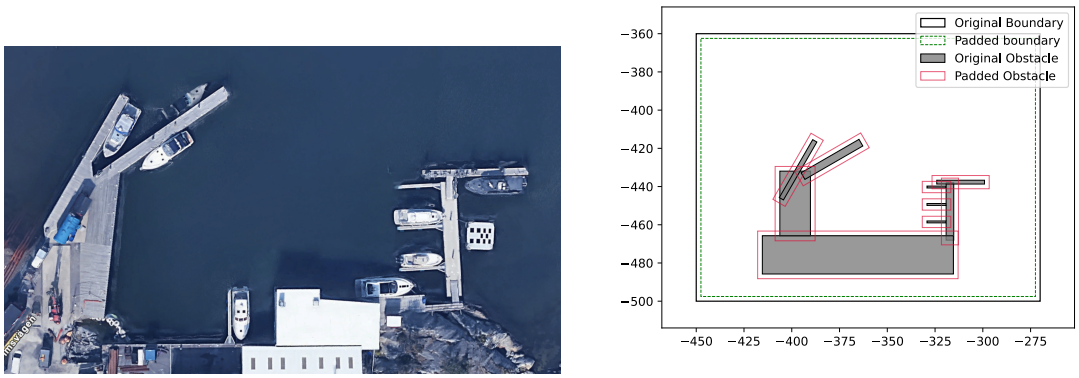


(a) Scenario 5.1: Drive into the harbor from the left and dock with 0° orientation at the right side of the dock. Dynamic obstacle entering from the left with a velocity vector at a 180° angle.



(b) Scenario 5.2: Drive into the harbor from the left and dock with 0° orientation at the right side of the dock. Dynamic obstacle entering from inside the harbor with a velocity vector at a 45° angle.

Figure 3.14: Two different scenarios including a)-b) including a dynamic obstacle. Tested on Python environment only.



(a) From: Google maps. Image over Krossholmen harbor.

(b) Illustration of the harbor as a polygon environment.

Figure 3.15: Harbor comparison between a) actual harbor at Krossholmen and b) polygon environment of the harbor at Krossholmen sent to the MPC for real-life tests.

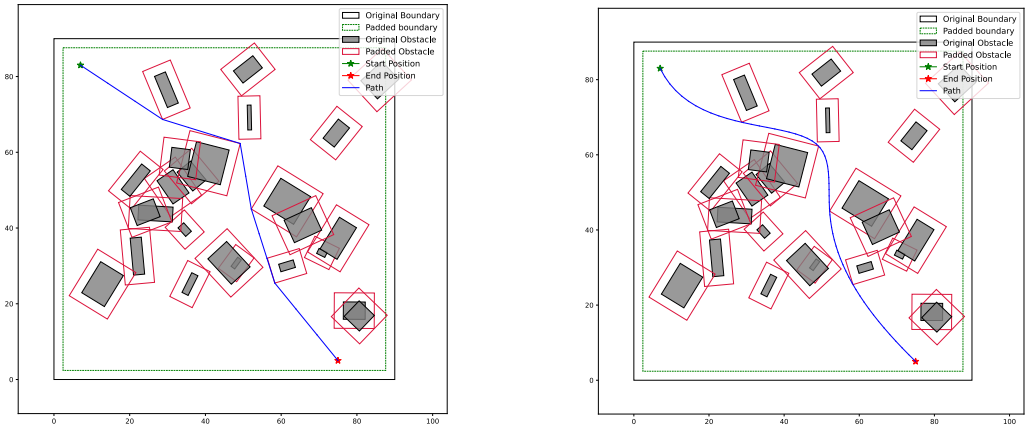
4
Results

This section presents the results of the simulations conducted to evaluate the performance of the autonomous docking system in both Python and Unity environments. Various scenarios were tested to assess the system's ability to handle different velocities, obstacle orientation, and path-following requirements. The comparison between the Python and Unity environments provides insights into how the system performs under idealized conditions versus more realistic, physically simulated conditions. This section also provides the results of initial tests of the algorithm in a real-world environment at Volvo Penta's test center at Krossholmer Harbor.

4.1 Path planner

As the modeled boat can drive in any direction, only a rough path was needed for the MPC to follow, and the implemented pathfinder did not need to consider a more typical case where the vehicle only can drive forward or backward. Before using the pathfinder, padding was added to the static obstacles to account for the size of the boat. In Figure 4.1a the mapped-out path in blue is shown avoiding the obstacles from start to dock.
To enhance users' comfort and prevent the path from making sharp turns, the path is first interpolated by adding extra coordinates between set points and then refined using a cubic spline. The interpolation maintains the rough shape of the path, while the cubic spline smooths out the corners, resulting in a more user-friendly path for the MPC to follow, as seen in Figure 4.1b.

Further tests were conducted on different scenarios to evaluate the Extremitypathfinder and the smoothing algorithm. Figure 4.2 shows the optimal paths determined by the Extremitypathfinder.



(a) A-star path **(b)** Smoothed out path

Figure 4.1: a) Using the Extremitypathfinder[26] to find the shortest path from start to dock. The pathfinder uses the corners of the padded obstacles to plan out the path. b) The path after being interpolated and cubic spline.

4.2 Trajectory planner

The trajectory planner is the main part of the autonomous docking system, responsible for determining the optimal path for the vessel to follow from its current position to the docking point. This section outlines the parameters used to ensure the trajectory planner operates effectively in both the Python and Unity simulation environments. The following subsections detail the parameter definitions and the specific configurations used to achieve stable and reliable simulations. These configurations include constraints, simulation weights, and vessel parameters, all of which are tailored to enhance the planner's performance across different scenarios.

4.2.1 Parameter definition

To ensure consistent and reliable performance for the trajectory planner, parameters were predefined for both the Python and Unity simulation environments. These parameters were deemed suitable after a process of tuning the variables until finding the parameters giving the most stable simulations. Because of this process, the parameters may differ between the two simulation environments, as both environments may need different requirements for obtaining stable simulations. These predefined parameters and their specific configurations are outlined in the tables 4.1 - 4.3.

Table 4.1 presents the constraints and model setup used in the simulations. These constraints ensure that the MPC model operates within realistic limits and has sufficient prediction horizon and path coordinates without requiring excessive data processing. In the Unity environment, a time step of 0.5 seconds was used to give the vessel time to move to the next position before getting the next input. Waiting in the

Table 4.1: Constraints and model setup for simulations

Parameter	Value	Description
u_{min}	-0.5	Vehicle constraint on the minimal velocity possible (m/s).
u_{max}	2.0	Vehicle constraint on the maximal velocity possible (m/s).
v_{min}	-1.0	Vehicle constraint on the minimal velocity possible (m/s).
v_{max}	1.0	Vehicle constraint on the maximal velocity possible (m/s).
r_{min}	0.104719755	Vehicle constraint on the maximal angular velocity (radians/s).
r_{max}	-0.104719755	Vehicle constraint on the maximal angular retardation (considered symmetric) (radians/s).
n_u	3	Number of control inputs.
n_x	3	Number of states.
t_s	0.2/0.5	Time step in Unity and Python (s).
N_O	15	Max number of static obstacles.
N	30/20	Length of the receding horizon controller in Python/Unity.
path_interval	30	Path interval points sent to the MPC.

proportional controller in the twin EVC installation rig. In the Python environment, a lower time step at 0.2 seconds was used to prevent the vessel from "jumping" to the next position when transitioning between states as the movement of the vessel had a one-to-one connection with the MPC output.
Table 4.2 details the simulation weights used in both environments. These weights adjust the importance of various factors during the optimization process, influencing the resulting trajectories.
Table 4.3 lists the specific vessel's parameters used in the trajectory planning. These parameters define the simulation conditions so that both environments are directly comparable with a real-world scenario.

In the simulations, dynamic weights were utilized to modify the cost functions, adapting to different scenarios that necessitate varying priorities. These scenarios, referred to as modes, include Driving mode, Dynamic Obstacle Avoidance mode, and Docking mode. The change in weights can be seen in table 4.4.

Driving mode is the standard mode for the autodocking function. Here all cost functions are active and the focus of the MPC is to track the reference path from the start to the docking position and avoid all obstacles close to the vessel. The focus lies on driving the boat forward toward the goal in the safest way possible which means that the cost of deviating from the reference path and the cost of driving close to an obstacle is high but also that a cost is made to prioritize forward drive.

Table 4.2: Simulation weights.

Parameter	Value	Description
$\mathcal{Q}_{\mathcal{O}}$	100	Weight on obstacle avoidance.
$\mathcal{Q}_{\mathcal{B}}$	10	Weight on the bound constraints.
$\mathcal{Q}_{\mathcal{D}}$	1000	Weight on dynamic obstacle avoidance .
$\mathcal{Q}_{CTE}$	50	Weight on the cross track error.
r	100	Weight on the deviation from the reference control input.
$\mathcal{Q}_{v}$	10	Weight on prioritizing forward drive.
q_{av}	10.0	Weight on acceleration in surge direction.
q_{au}	10.0	Weight on acceleration in sway direction.
$q_{a\psi}$	10.0	Weight on acceleration in the angular rotation.
$\mathcal{Q}_{\mathcal{N}}$	0	Weight on the terminal state.
$\mathcal{Q}_{\psi\mathcal{N}}$	0	Weight on the terminal heading.

Dynamic Obstacle Avoidance mode is triggered when a dynamic obstacle encroaches on the planned path within a predefined distance, set to two boat lengths. In the dynamic obstacle avoidance mode, the dynamic weights sent to the MPC are changed to shift the focus to avoiding the approaching obstacle. This means that the cost for deviating from the reference path will be set to zero such that the focus lies only on avoiding the obstacle, allowing the MPC to choose the next set point freely. When the obstacle has been avoided a new reference path will be calculated using the path planning algorithm from the current position to the docking position and sent to the MPC before entering driving mode again.

Docking mode is activated when the vessel approaches the goal position. In this mode, the cost of deviating from the path is nullified by setting its dynamic weight to zero. In contrast, the weight for the cost of deviating from the final position is increased. This adjustment ensures that the MPC prioritizes minimizing the deviation from the docking position. Additionally, cost functions for obstacle avoidance and maintaining the reference velocity remain active to guarantee a safe docking procedure.

4.3 Simulations

The simulations were conducted on a system with the following specifications:

- Processor: 13th Gen Intel(R) Core(TM) i7-1355U, 1700 MHz, 10 Cores, 12 Logical Processors

Table 4.3: Simulation parameters.

Parameter	Value	Description
L	7.5	Length of the boat (meters).
W	3	Width of the boat (meters).
offset	1.875	Offset from the center of the boat to extra reference points (meters).
padding	2.46	Size of the padding of each obstacle and boundary.
goal_x	0.5	Margin of errors (meters).
goal_y	0.5	Margin of errors (meters).
goal_theta	0.5	Margin of errors (degrees).

Table 4.4: Simulation weights across the three modes: driving, dynamic obstacle avoidance, and docking mode.

Parameter	Driving	Dynamic Obstacle Avoidance	Docking
$\mathcal{Q}_{\mathcal{O}}$	100	100	100
$\mathcal{Q}_{\mathcal{B}}$	10	10	10
$\mathcal{Q}_{\mathcal{D}}$	1000	1000	1000
$\mathcal{Q}_{CTE}$	50	0	0
r	100	100	500
$\mathcal{Q}_{v}$	10	10	0
q_{av}	10.0	10.0	10.0
q_{au}	10.0	10.0	10.0
$q_{a\psi}$	10.0	10.0	10.0
$\mathcal{Q}_{\mathcal{N}}$	0	0	30
$\mathcal{Q}_{\psi\mathcal{N}}$	0	0	30

- Operating System: Microsoft Windows 11 Enterprise

The simulations depict five cases representing typical boat maneuvering scenarios in a docking environment. The first four cases are simulated in the Python environment described in 3.3.1 and the unity environment in 3.3.2, whereas the fifth case is only simulated in the Python environment. The simulations conducted in the Unity environment aimed to evaluate the performance of the trajectory planner in a more realistic environment.

For each simulated case, graphs of how the velocity and acceleration variate throughout the simulation are depicted to measure user comfort, i.e., to observe whether the acceleration and velocity indicate stable vessel motion or not. The solver time and cost are visualized and show how long it takes for the MPC to find a solution in each time step and the minimized cost in each time step.

The following subsections present the simulation results for each case in the Python and Unity environment.

4.3.1 Case 1

Case 1 focused on examining the trajectory planner's performance in a straightforward scenario. The case focuses on showing how well the MPC follows the planned path through complex turns and a typical docking scenario where the boat will reverse dock into the final position. The start and docking positions for this case are shown in Figure 3.13a.

4.3.1.1 Simulations in Python environement

The simulated trajectory is depicted in Figure 4.3 with the planned trajectory in blue and the actual trajectory in dashed red. Figure 4.4 and 4.5 depict the actual velocities and acceleration for the vessel throughout the simulation. The solver time and cost are visualized in Figure 4.6.

4.3.1.2 Simulations in Unity environment

The simulated trajectory, depicted in figure 4.7, illustrates the planned trajectory in blue and the actual trajectory in dotted red. Figure 4.8 and Figure 4.9 display the velocities and accelerations, respectively, with their simulated actual counterparts. The solver time and cost for case 1 are shown in Figure 4.10.

4.3.2 Case 2

In the second case, the trajectory planner had a similar task as in the first scenario but in this case, the boat enters from the other side of the harbor. The start and docking positions are shown in Figure 3.13b.

4.3.2.1 Simulations in Python environment

The simulated trajectory is depicted in figure 4.11 with the planned trajectory in blue and the actual trajectory in dashed red. Figure 4.12 and 4.13 depict the actual velocities and acceleration for the vessel throughout the simulation. The solver time and cost are visualized in Figure 4.14.

4.3.2.2 Simulations in Unity environment

The trajectory simulated in this case is depicted in Figure 4.23. Figures 4.24, 4.25, and 4.26 shows the velocities, accelerations, and solver values.

4.3.3 Case 3

Case 3 involved exploring the trajectory planner's performance in a more challenging scenario by docking with an angle using only the vessel's sway velocity. I.e., having the boat slide sideways into the docking position. The start and docking positions are shown in Figure 3.13c.

4.3.3.1 Simulations in Python environment

The simulated trajectory is depicted in figure 4.19 with the planned trajectory in blue and the actual trajectory in dashed red. Figure 4.20 and 4.21 depict the actual velocities and acceleration for the vessel throughout the simulation. The solver time and cost are visualized in Figure 4.22.

4.3.3.2 Simulations in Unity environment

The trajectory simulated in this case is depicted in Figure 4.23. Figures 4.24, 4.25, and 4.26 show the velocities, accelerations, and solver time and cost.

4.3.4 Case 4

Case 4 presented a more challenging scenario where the vessel starts in a docking spot, i.e. a narrow space, with the end position in a new docking position at the other side of the dock. The start and docking positions are shown in Figure 3.13d.

4.3.4.1 Simulations in Python environment

The simulated trajectory is depicted in Figure 4.27 with the planned trajectory in blue and the actual trajectory in dashed red. Figure 4.28 and 4.29 depict the actual velocities and acceleration for the vessel throughout the simulation. The solver time and cost are visualized in Figure 4.30.

4.3.4.2 Simulations in Unity environment

The initial simulation in the Unity environment gave an unexpectedly strange trajectory figure, see Figure 4.31. Figure 4.32, 4.33, and 4.34 depicts the trajectory, velocities, accelerations, and solver metrics for this suboptimal trajectory, respectively. The vessel was steered back to the starting position and the simulations were made again, resulting in a more stable and optimal trajectory, as showcased in Figures 4.31, 4.35, 4.36, and 4.37.

4.3.5 Case 5

In the fifth case, the environment is set up with a dynamic obstacle interfering with the planned path for the boat. Here, two scenarios in the Python environment are simulated, where the obstacle interferes from different angles and directions, see Figure 3.14.

In the first scenario, the dynamic obstacle enters with a 180 ° degree angle according to Figure 4.38. In the Figure, the numbers in the green ellipse and the boat represent the objects at different times during the simulation. During step 3 in the figure, the MPC detected the dynamic obstacles and went into dynamic obstacle avoidance mode, meaning that it deviated from the reference path to avoid the obstacle. When the dynamic obstacle was safely avoided, see step 4 in the Figure, the path-planning algorithm planned a new reference path from the current position to the docking

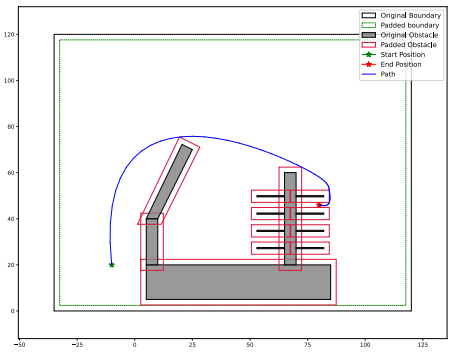
position which the MPC followed to the goal. Figure 4.39, 4.40 and 4.41 depict the variations in velocity, acceleration for the vessel during the simulation and the solver time and total cost at each time step.
In the second scenario, the dynamic obstacle starts within the harbor with a 45° angle heading out from the harbor, see figure 4.42. At step 5 during the simulation, the MPC detected the dynamic obstacle and started deviating from the reference path until the obstacle was safely avoided, at step 5. The path planner then planned a new reference path for the MPC to follow until reaching the docking mode. Figure 4.43, 4.44 and 4.45 depict the variations in velocity, acceleration for the vessel during the simulation and the solver time and total cost at each time step.

4.3.6 Initial tests in real-world environment

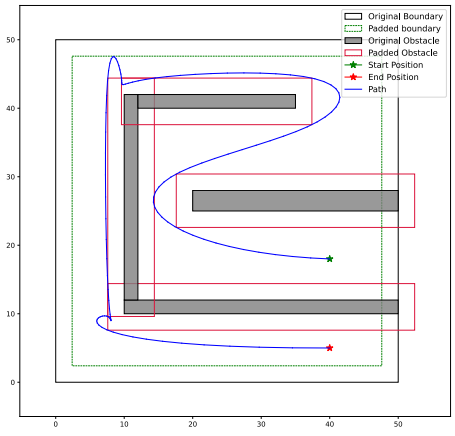
Initial tests of the system in a real-world environment required some adjustments to system parameters, these changes are visible in table 4.5. Due to the project's time constraints, real-life tests were limited to a single afternoon. Consequently, the focus was primarily on verifying the compatibility of the MPC with the boat's proportional control system and not on testing actual docking scenarios. Hence, the ability to obtain comprehensive results regarding the system's performance in a real-life setting was restricted. While some initial behavior was observed, such as abrupt accelerations similar to those seen in Unity simulations, further testing is needed to draw meaningful conclusions.

Table 4.5: Parameters altered for testing in a real-life scenario.

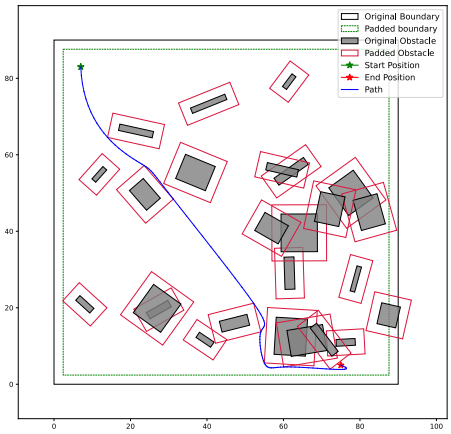
Parameter	Value	Description
u_{min}	-0.5	Vehicle constraint on the minimal velocity possible (m/s).
u_{max}	0.5	Vehicle constraint on the maximal velocity possible (m/s).
v_{min}	-0.5	Vehicle constraint on the minimal velocity possible (m/s).
v_{max}	0.5	Vehicle constraint on the maximal velocity possible (m/s).
r_{min}	0.05	Vehicle constraint on the maximal angular velocity (radians/s).
r_{max}	-0.05	Vehicle constraint on the maximal angular retardation (considered symmetric) (radians/s).
L	12	Length of the boat (meters).
W	4	Width of the boat (meters).
offset	3	Offset from the boat's center to extra reference points (meters).
padding	2.46	Size of the padding of each obstacle and boundary.
t_s	0.5	Time step in for real-life tests (s).
q_{av}	100.0	Weight on acceleration in surge direction.
q_{au}	100.0	Weight on acceleration in sway direction.
$q_{a\psi}$	1000.0	Weight on acceleration in the angular rotation.



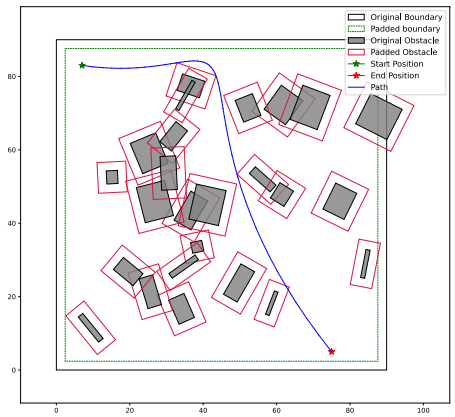
(a) Scenario 1: Typical case when going into a docking position.



(b) Scenario 2: A maze-like environment.



(c) Scenario 3: Random generated obstacle-filled environment



(d) Scenario 4: Random generated obstacle-filled environment.

Figure 4.2: Four different path scenarios: a) Harbour path, b) Maze path, two random generated obstacle-filled environments c) and d).

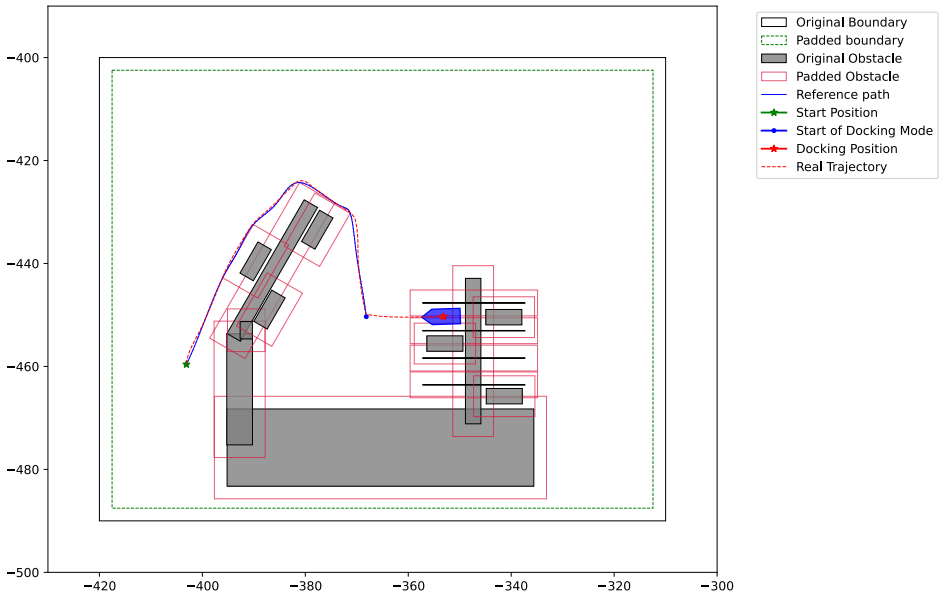


Figure 4.3: Real trajectory in dashed red and planned trajectory in blue for case 1 in the simulated Python environment.

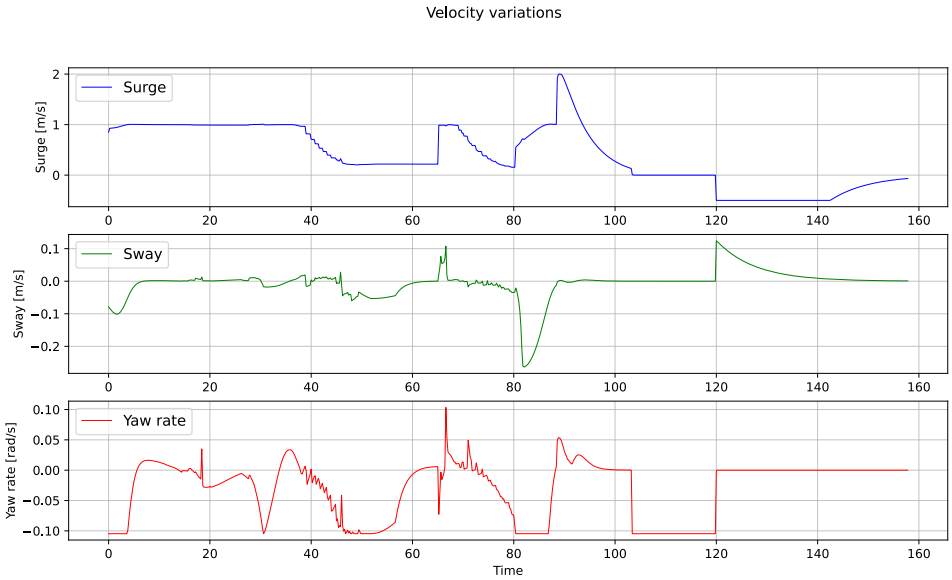


Figure 4.4: Linear and angular velocity for the simulated boat in the Python environment for case 1.

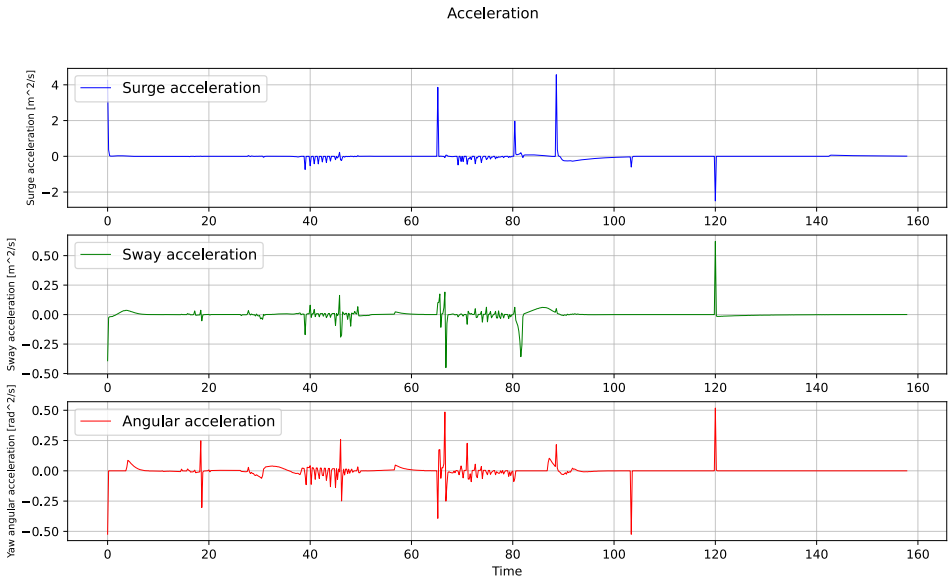


Figure 4.5: Linear and angular acceleration for the simulated boat in the Python environment in case 1.

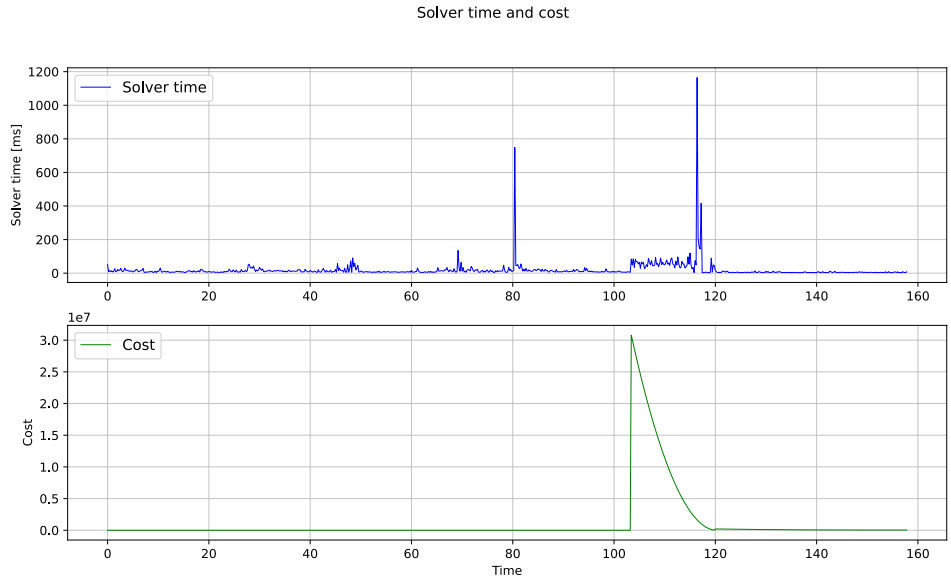


Figure 4.6: Solver time and cost at each time step during the simulation in the Python environment for case 1.

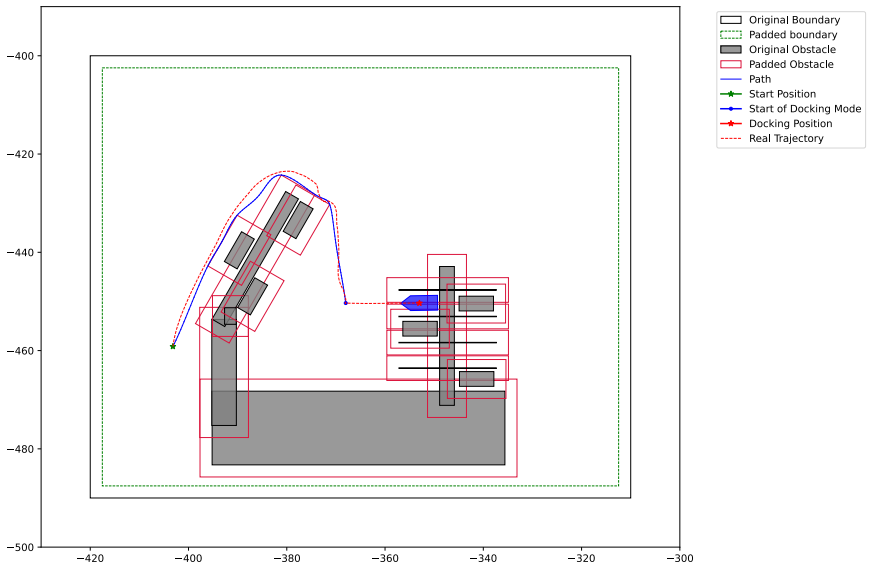


Figure 4.7: Real trajectory in dashed red and planned trajectory in blue for case 1 in the simulated Unity environment.

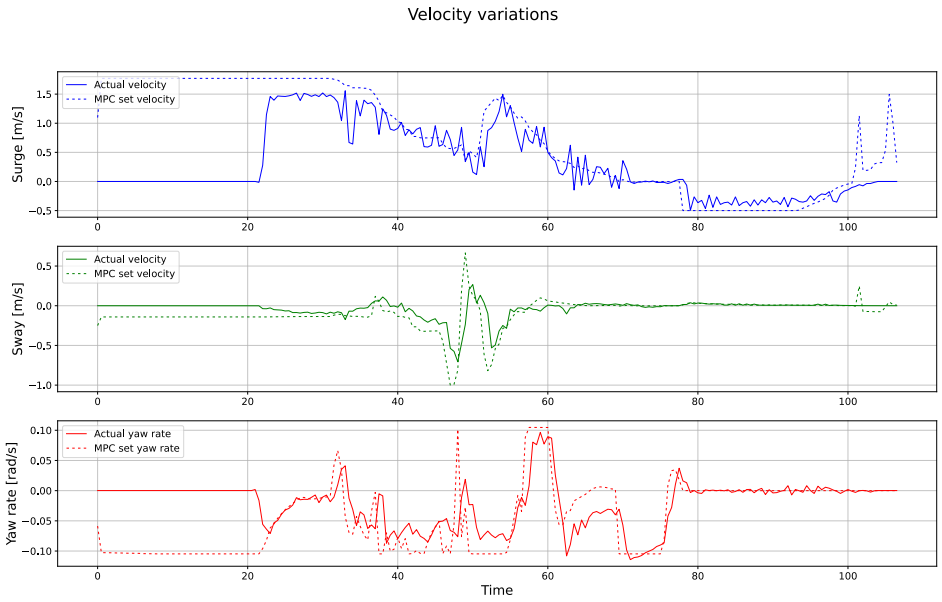


Figure 4.8: The set and simulated actual linear and angular velocity of the boat in the Unity environment for case 1.

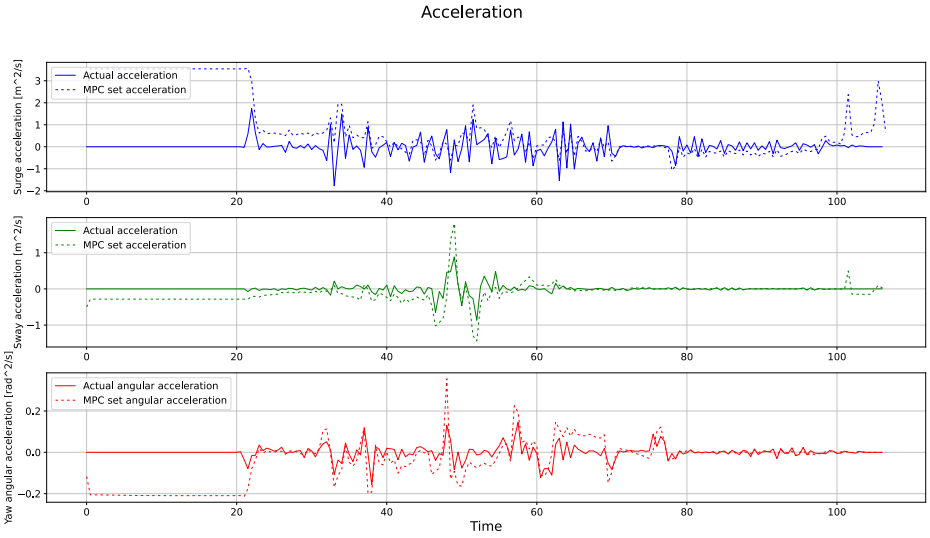


Figure 4.9: The set and simulated actual linear and angular acceleration of the boat in the Unity environment for case 1.

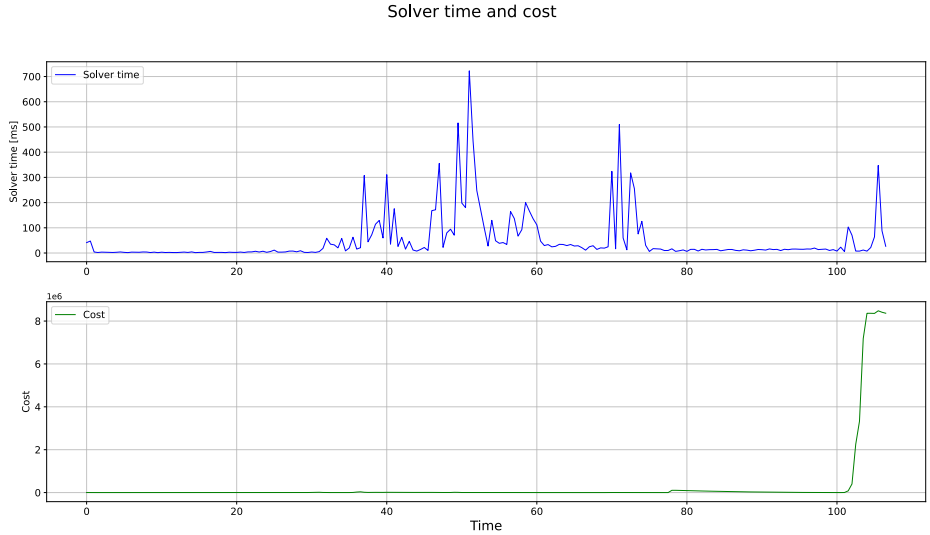


Figure 4.10: Solver time and cost at each time step during the simulation in the Unity environment for case 1.

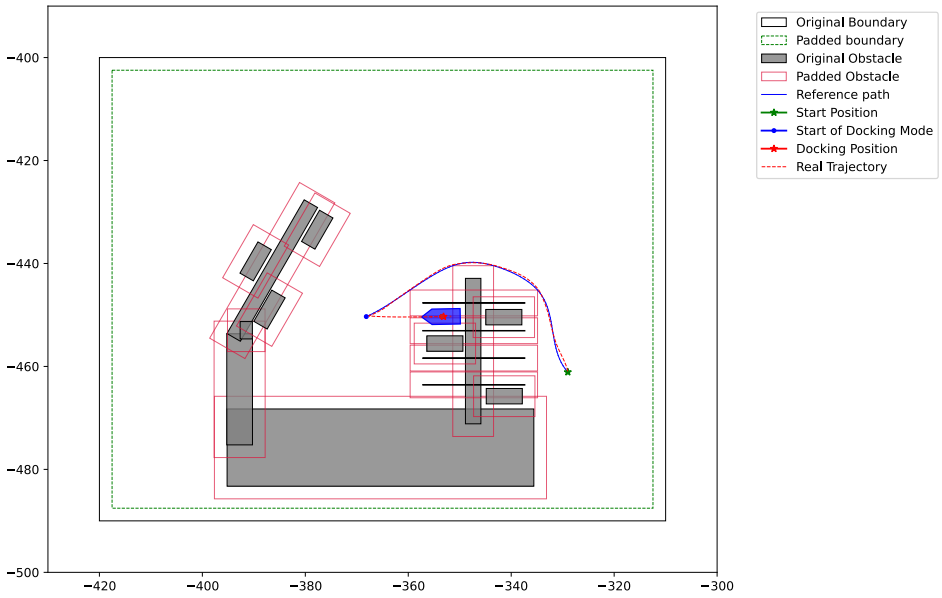


Figure 4.11: Real trajectory in dashed red and planned trajectory in blue for case two in the simulated Python environment.

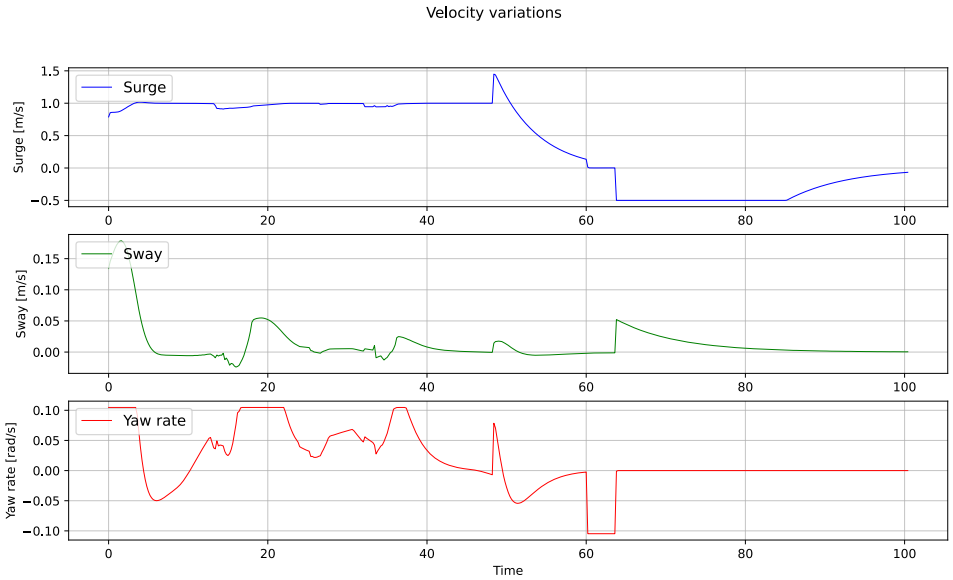


Figure 4.12: Linear and angular velocity for the simulated boat in the Python environment for case 2.

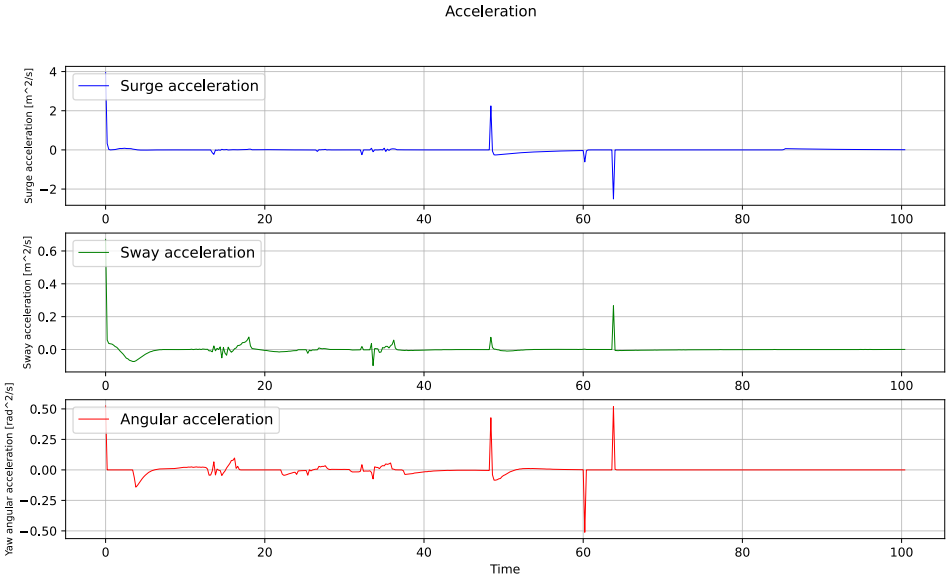


Figure 4.13: Linear and angular acceleration for the simulated boat in the Python environment for case 2.

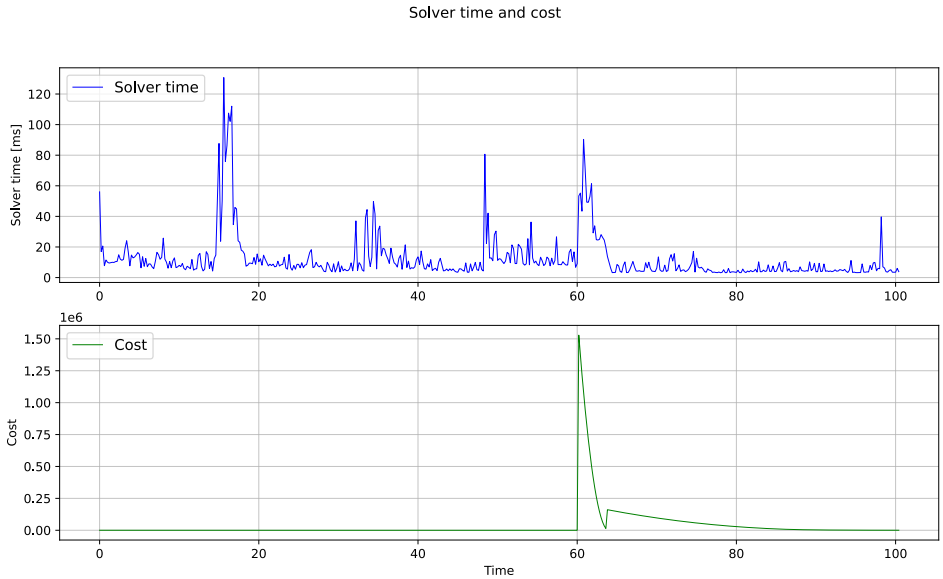


Figure 4.14: Solver time and cost at each time step during the simulation in the Python environment for case 2.

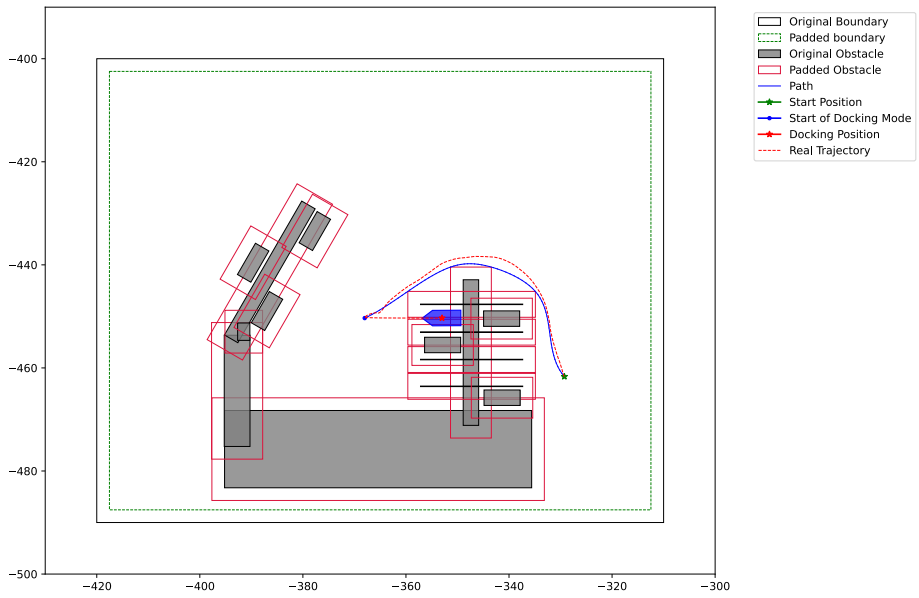


Figure 4.15: Real trajectory in dashed red and planned trajectory in blue for case 2 in the simulated Unity environment.

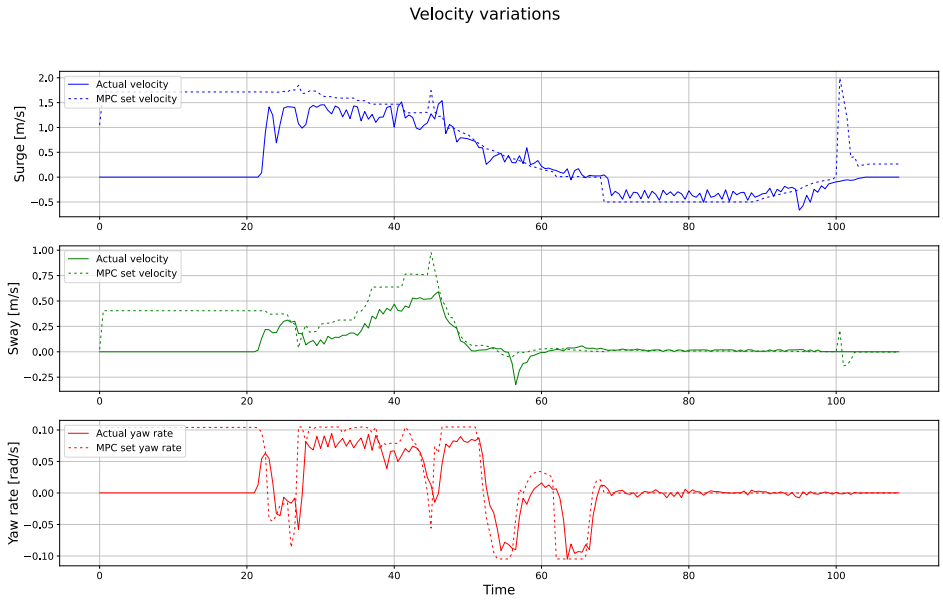


Figure 4.16: The set and simulated actual linear and angular velocity of the boat in the Unity environment for case 2.

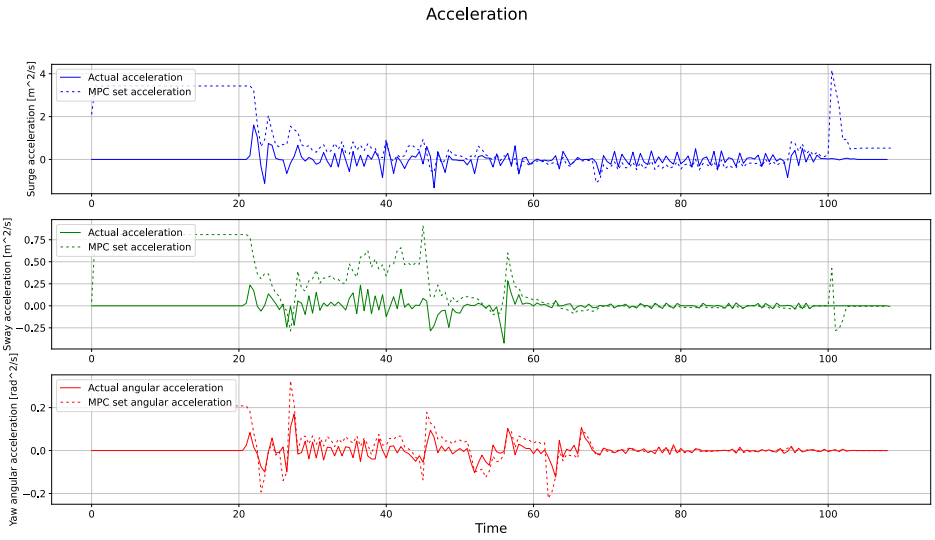


Figure 4.17: The set and simulated actual linear and angular acceleration of the boat in the Unity environment for case 2.

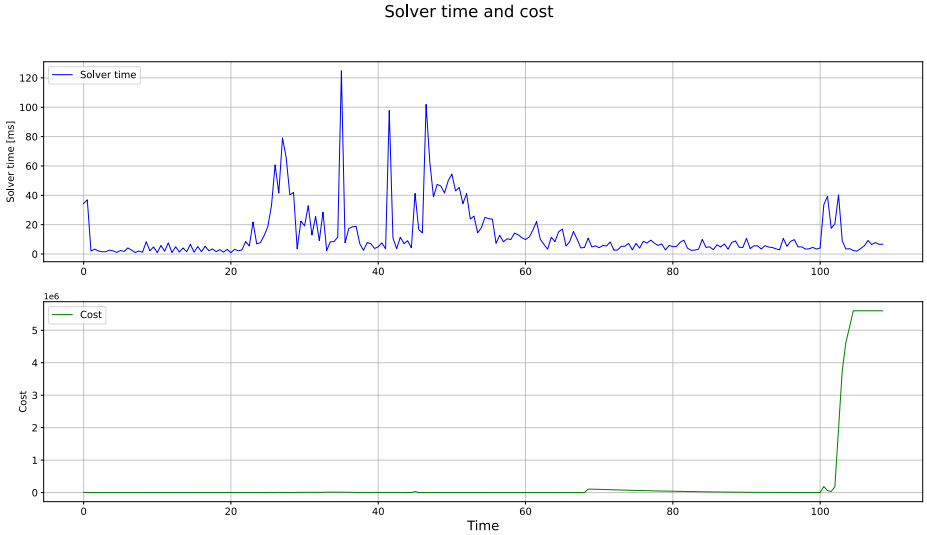


Figure 4.18: Solver time and cost at each time step during the simulation in the Unity environment for case 2.

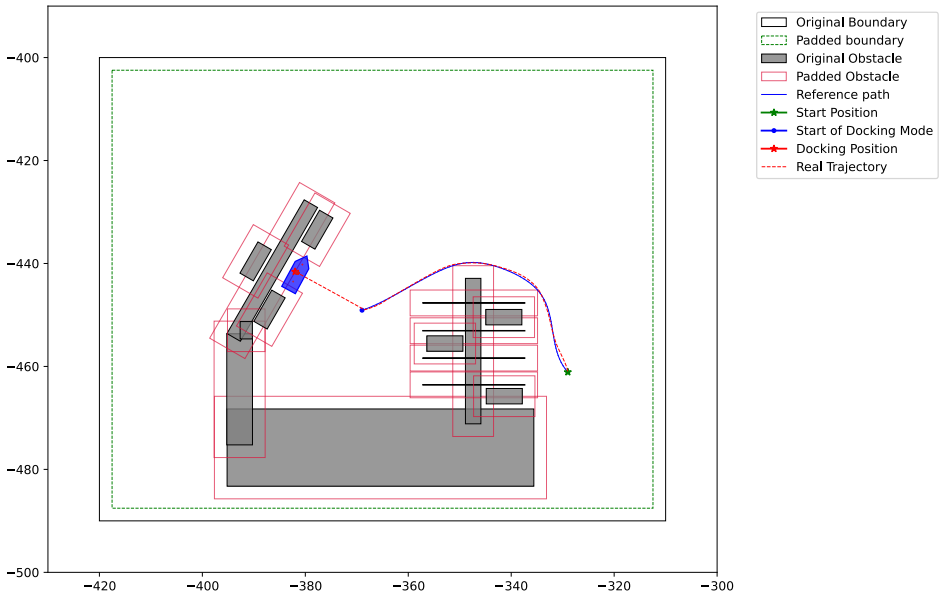


Figure 4.19: Real trajectory in dashed red and planned trajectory in blue for case 3 in the simulated Python environment.

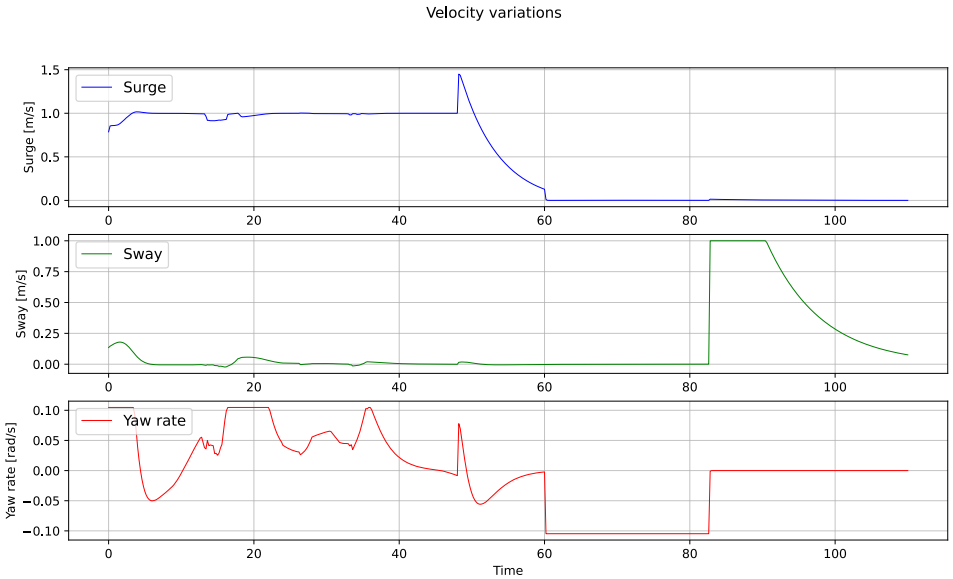


Figure 4.20: Linear and angular velocity for the simulated boat in the Python environment for case 3.

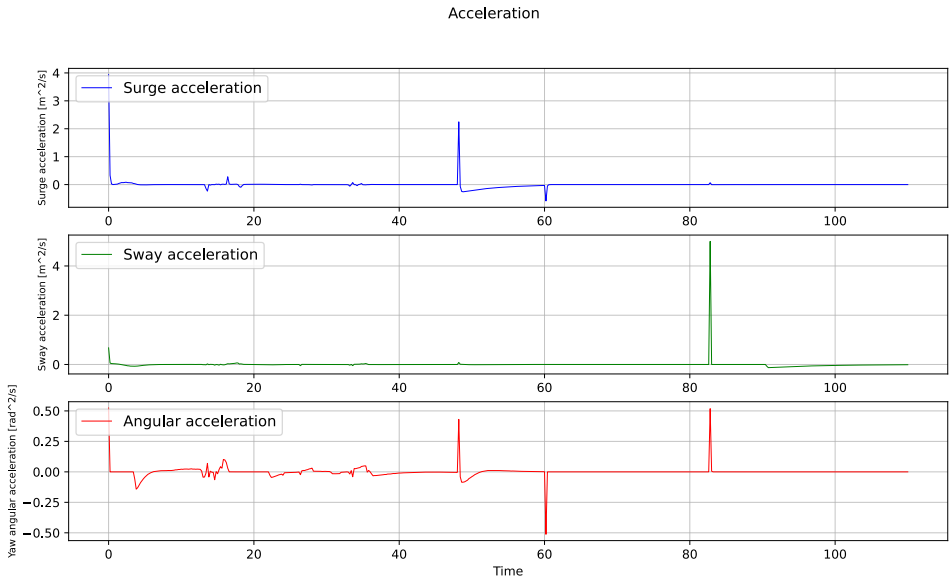


Figure 4.21: Linear and angular acceleration for the simulated boat in the Python environment for case 3.

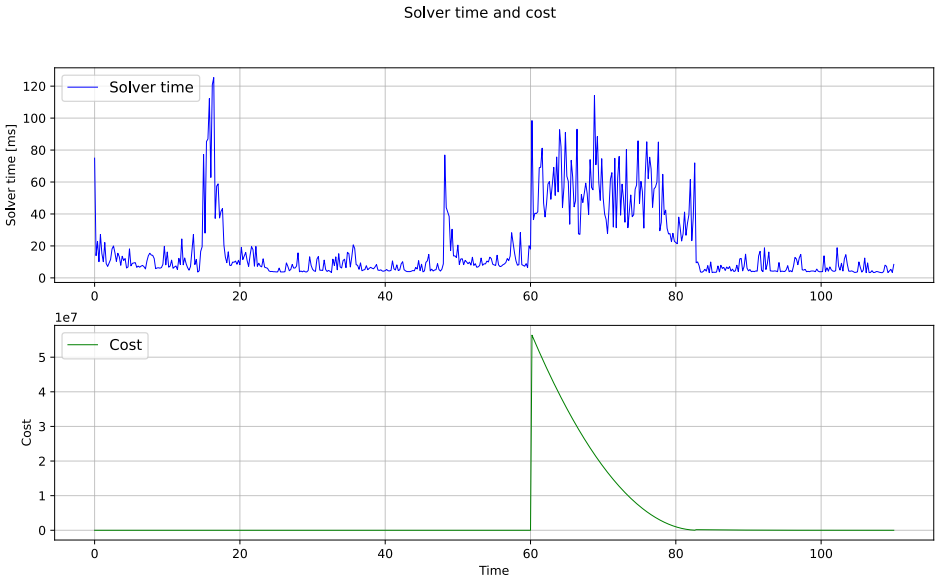


Figure 4.22: Solver time and cost at each time step during the simulation in the Python environment for case 3.

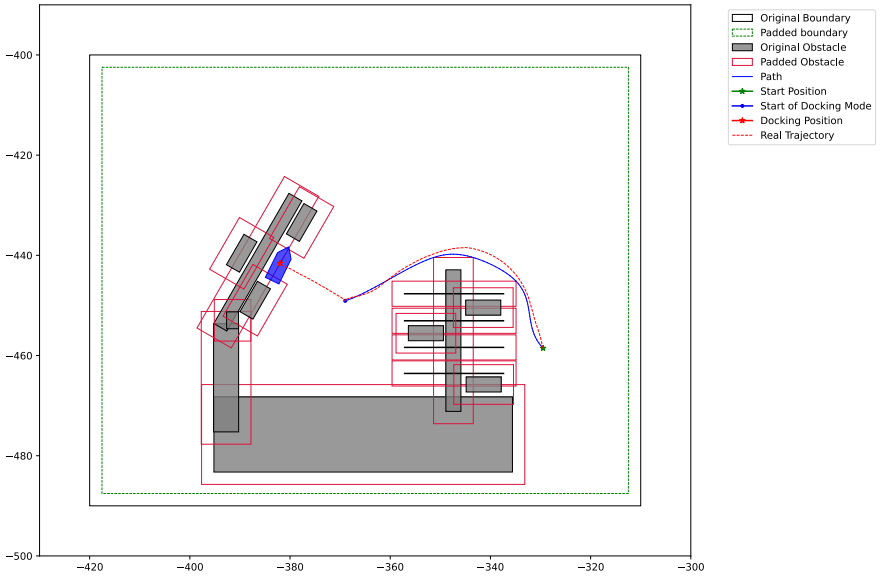


Figure 4.23: Real trajectory in dashed red and planned trajectory in blue for case 3 in the simulated Unity environment.

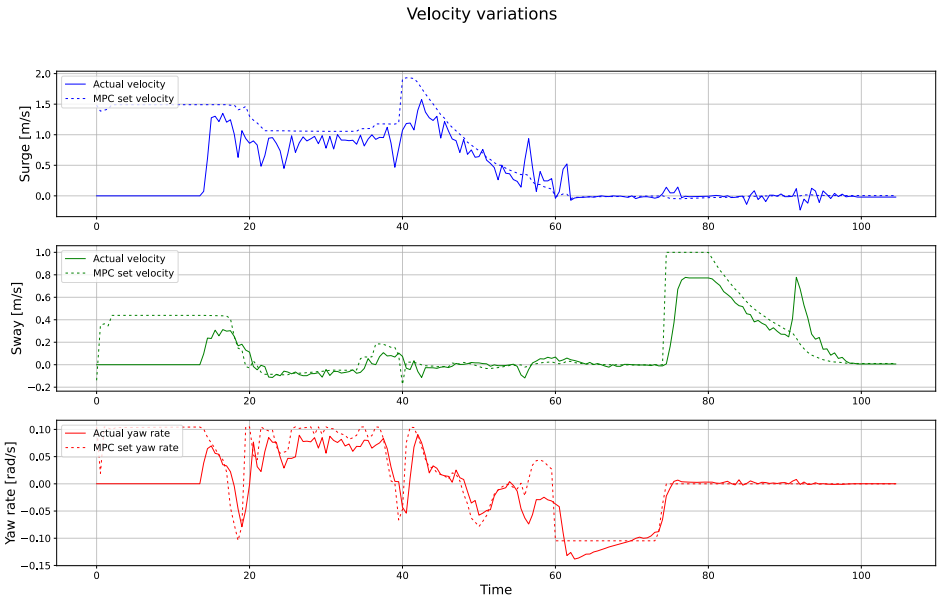


Figure 4.24: The set and simulated actual linear and angular velocity of the boat in the Unity environment for case 3.

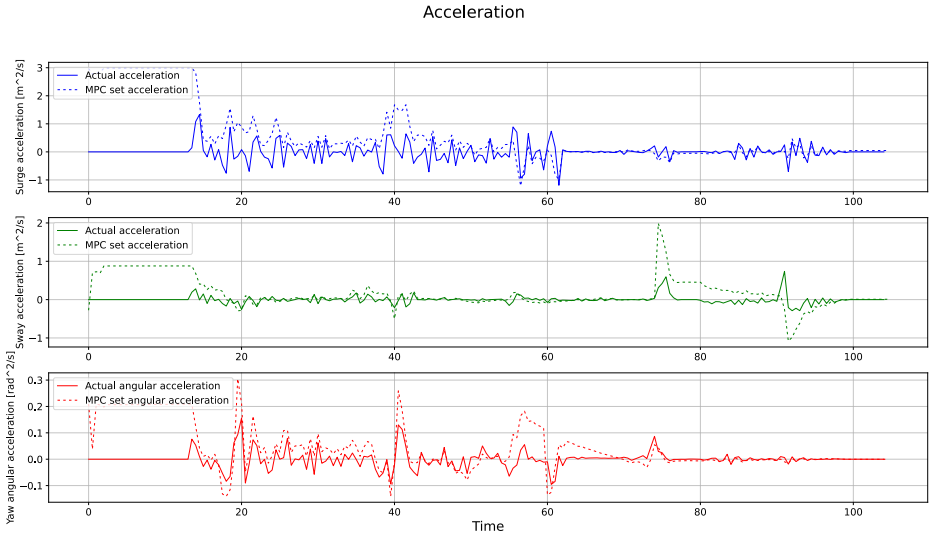


Figure 4.25: The set and simulated actual linear and angular acceleration of the boat in the Unity environment for case 3.

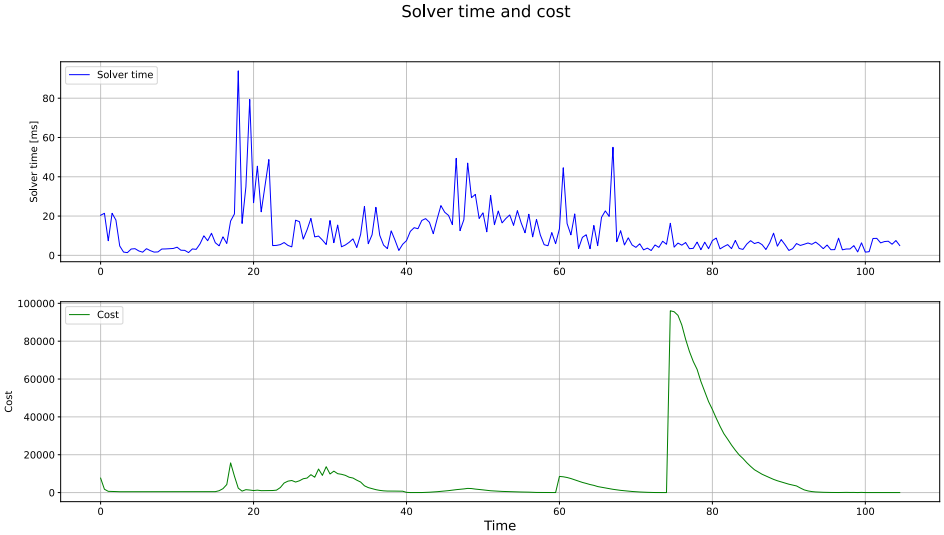


Figure 4.26: Solver time and cost at each time step during the simulation in the Unity environment for case 3.

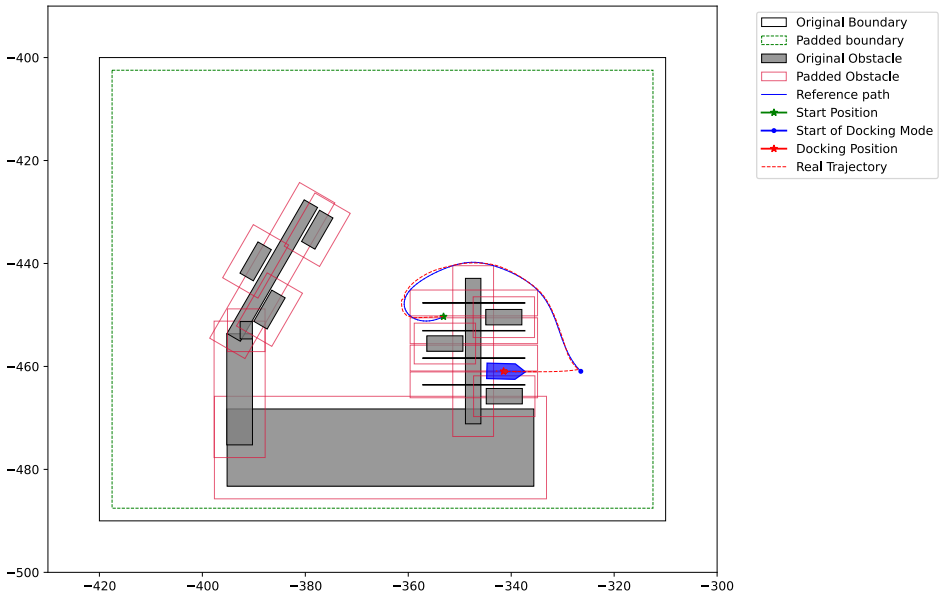


Figure 4.27: Real trajectory in dashed red and planned trajectory in blue for case 4 in the simulated Python environment.

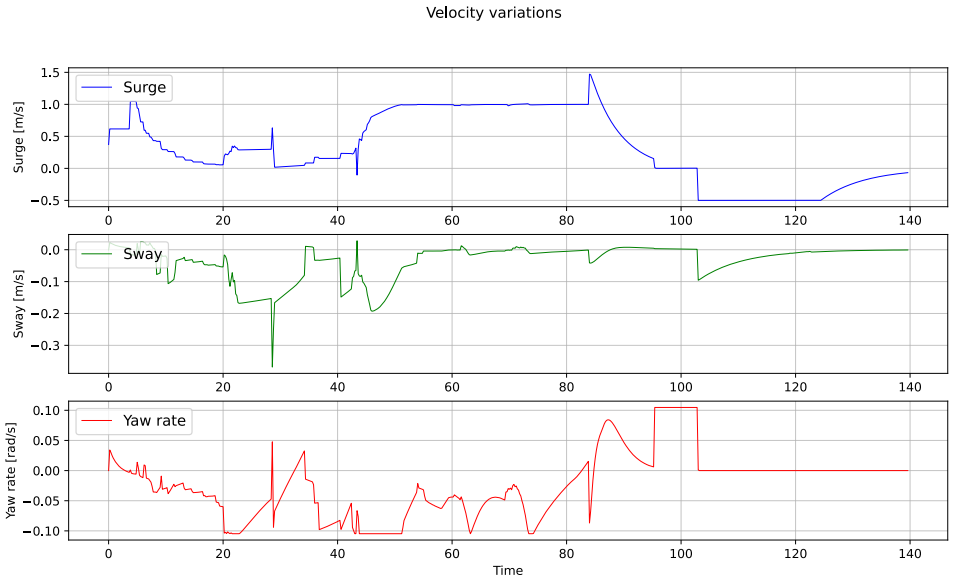


Figure 4.28: Linear and angular velocity for the simulated boat in the Python environment for case 4.

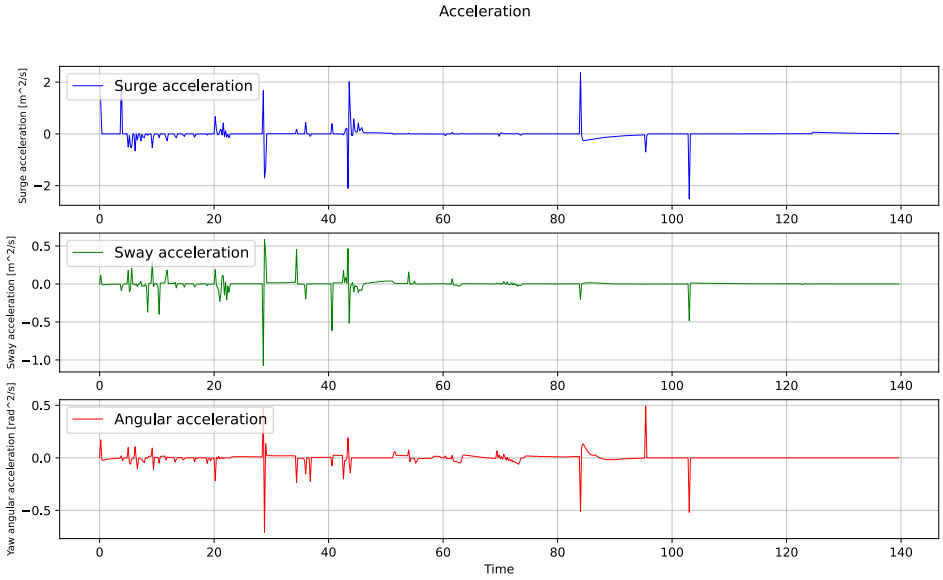


Figure 4.29: Linear and angular acceleration for the simulated boat in the Python environment for case 4.

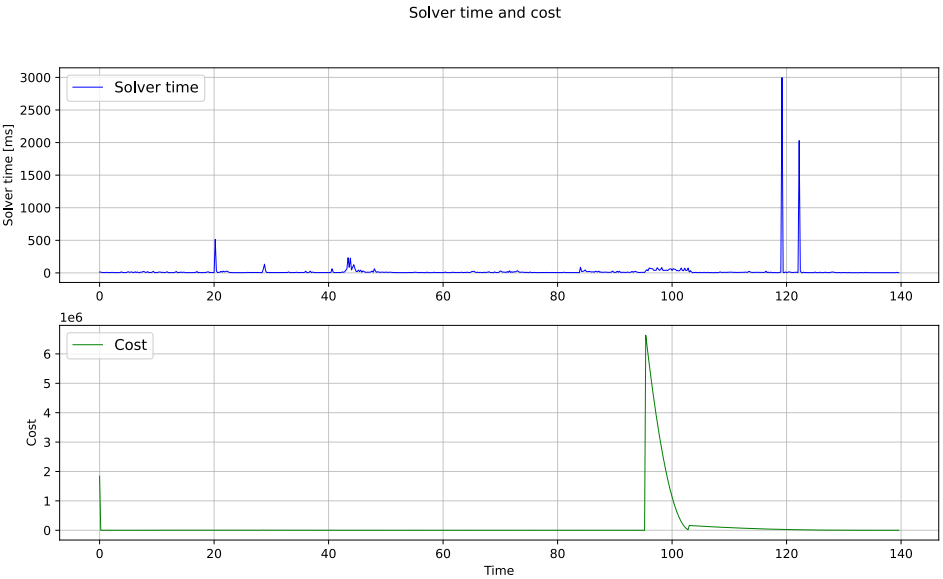
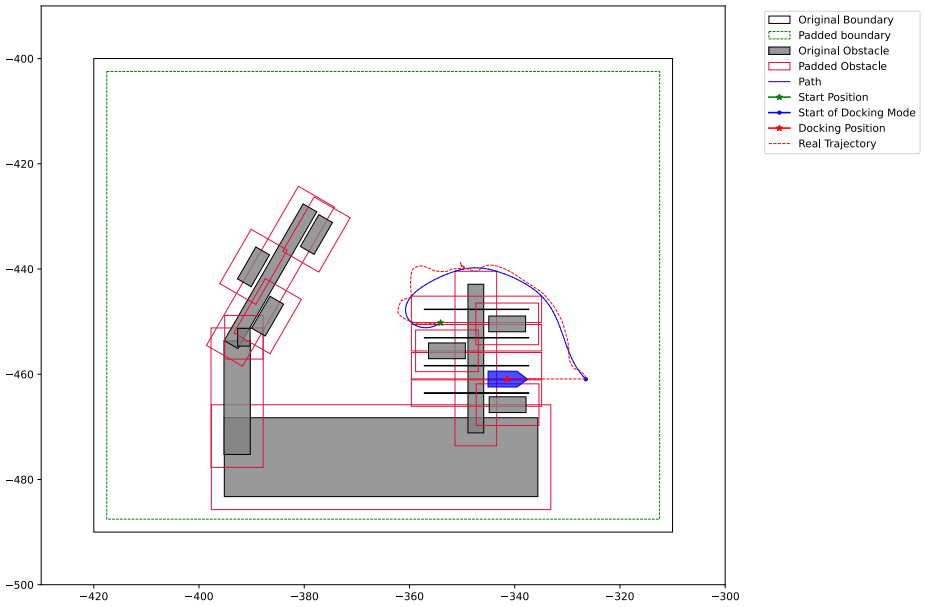
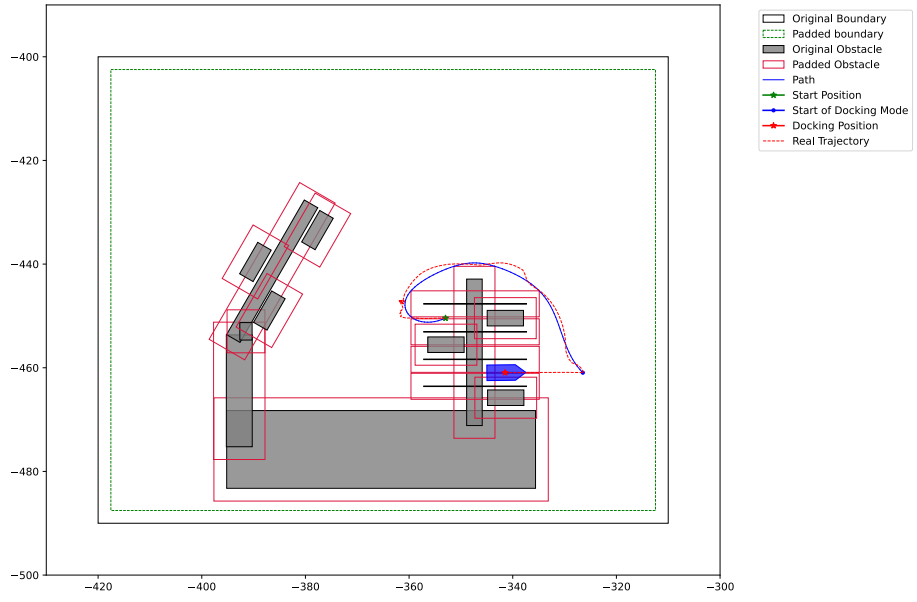


Figure 4.30: Solver time and cost at each time step during the simulation in the Python environment for case 4.



(a) The suboptimal trajectory in dashed red and planned trajectory in blue for case 4 in the simulated Unity environment.



(b) The stable and acceptable trajectory in dashed red and planned trajectory in blue for case 4 in the simulated Unity environment.

Figure 4.31: A comparison between the suboptimal and acceptable trajectory of case 4 in the Unity environment.

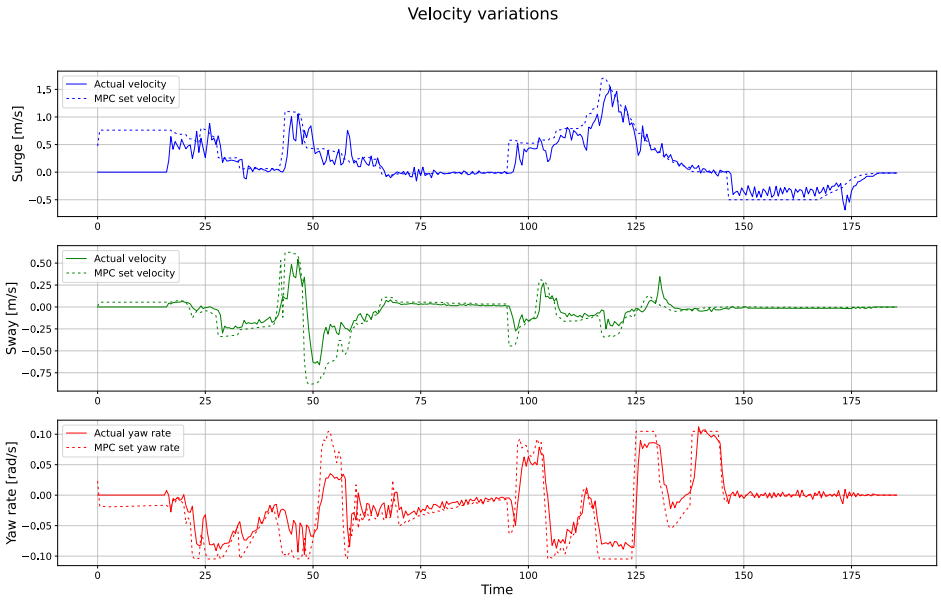


Figure 4.32: The set velocity and simulated actual velocity for the suboptimal trajectory of the boat in case 4 in the Unity environment.

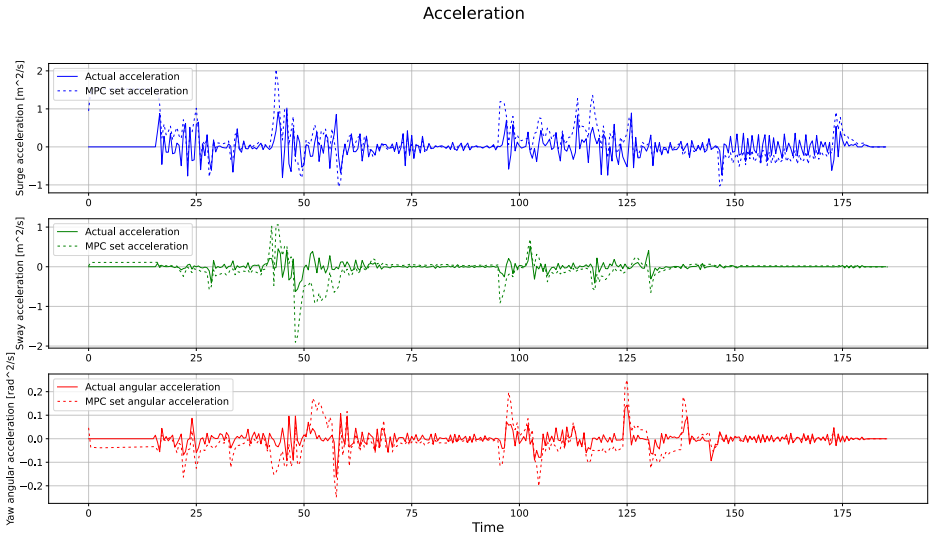


Figure 4.33: The set acceleration and simulated actual acceleration for the suboptimal trajectory of the boat in case 4 in the Unity environment.

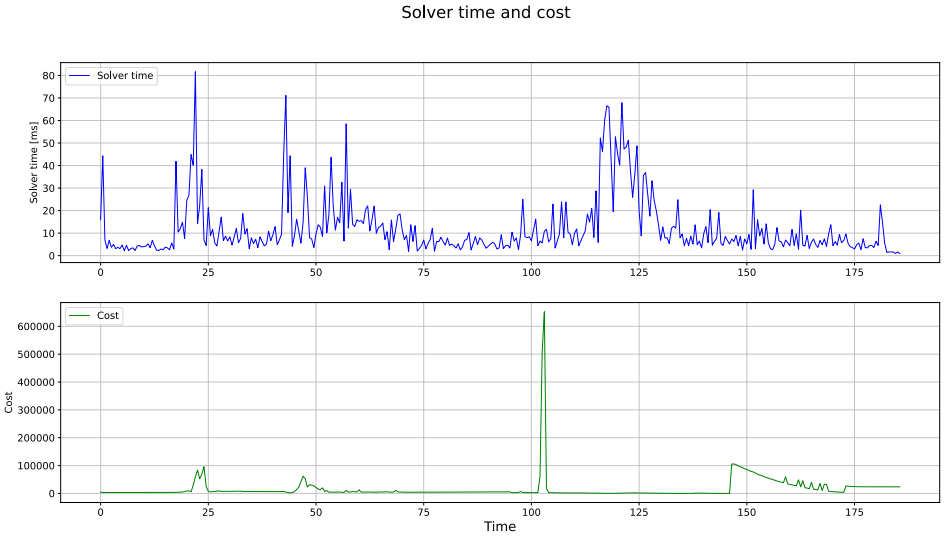


Figure 4.34: Solver time and cost at each time step for the suboptimal boat trajectory during the simulation in the Unity environment for case 4.

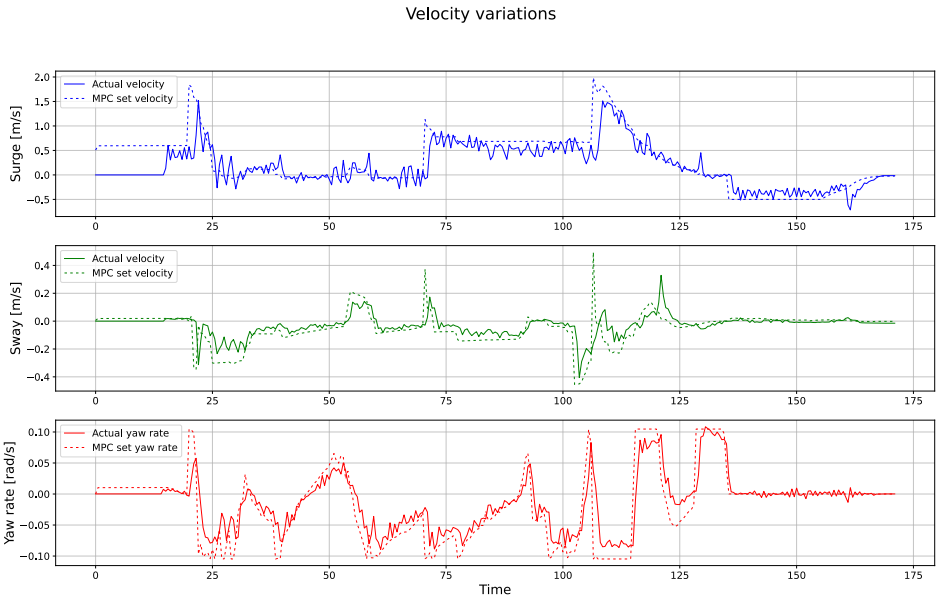


Figure 4.35: The set velocity and simulated actual velocity for the stable and acceptable trajectory of the boat in case 4 in the Unity environment.

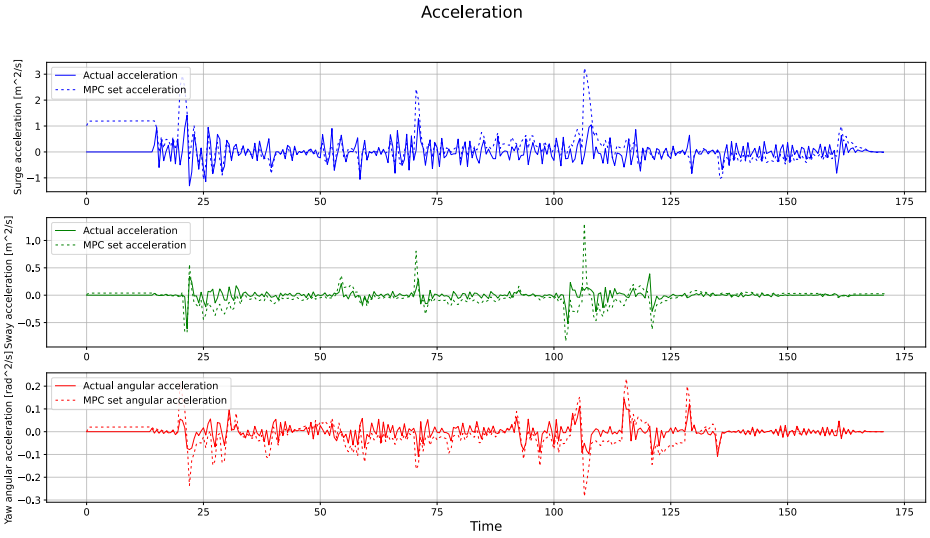


Figure 4.36: The set acceleration and simulated actual acceleration for the stable and acceptable trajectory of the boat in case 4 in the Unity environment.

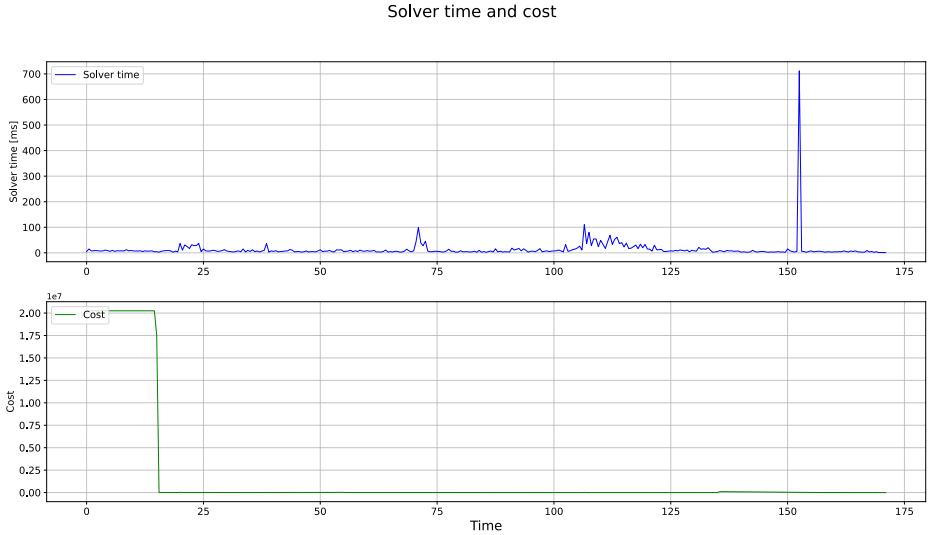


Figure 4.37: Solver time and cost at each time step for the stable and acceptable boat trajectory during the simulation in the Unity environment for case 4.

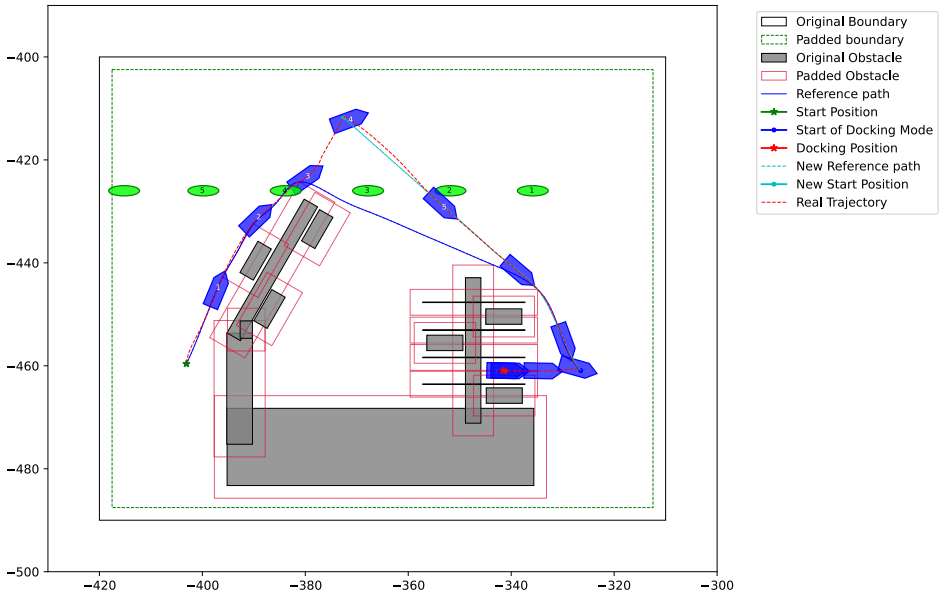


Figure 4.38: Simulated trajectory in dashed red and planned trajectory in blue for case 5 with a dynamic obstacle (green ellipse) entering from the right interfering with the planned path.

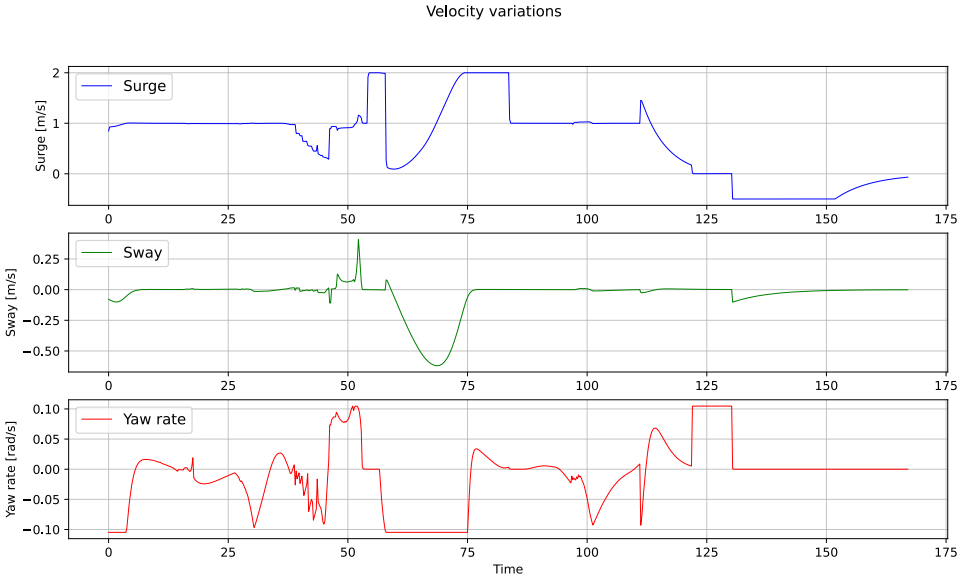


Figure 4.39: Linear and angular velocity for the simulated boat in case 5 for scenario 1.

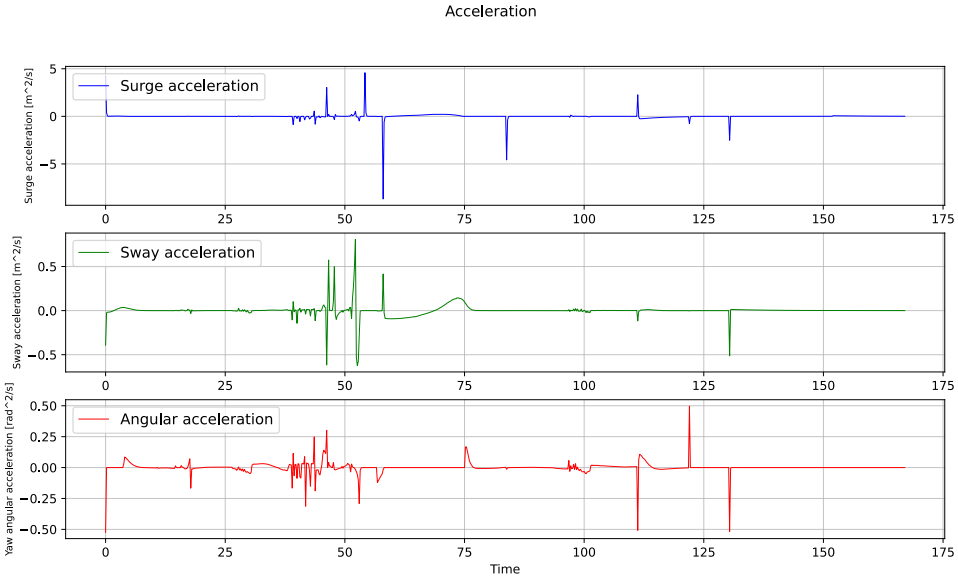


Figure 4.40: Linear and angular acceleration for the simulated boat in case 5 for scenario 1.

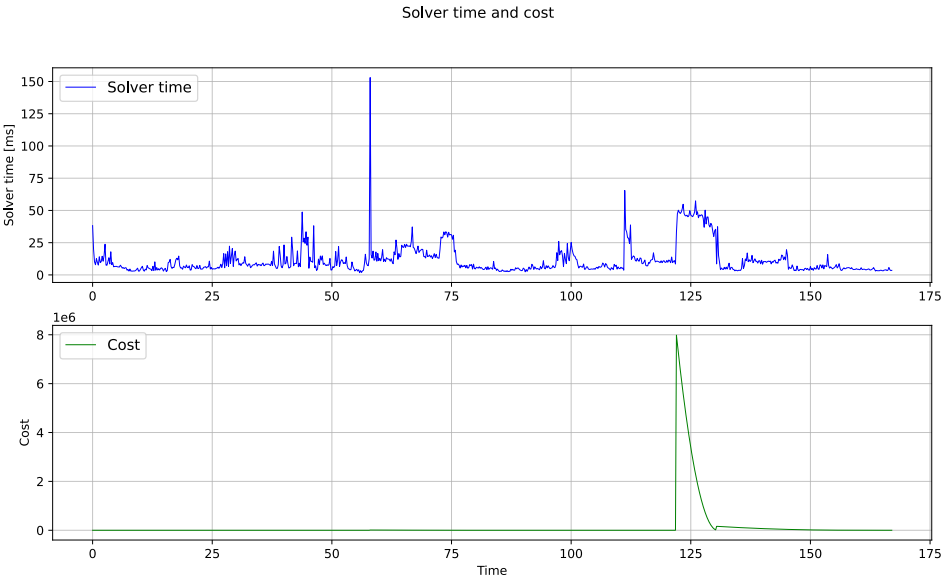


Figure 4.41: Solver time and cost at each time step during the simulation for case 5 scenario 1.

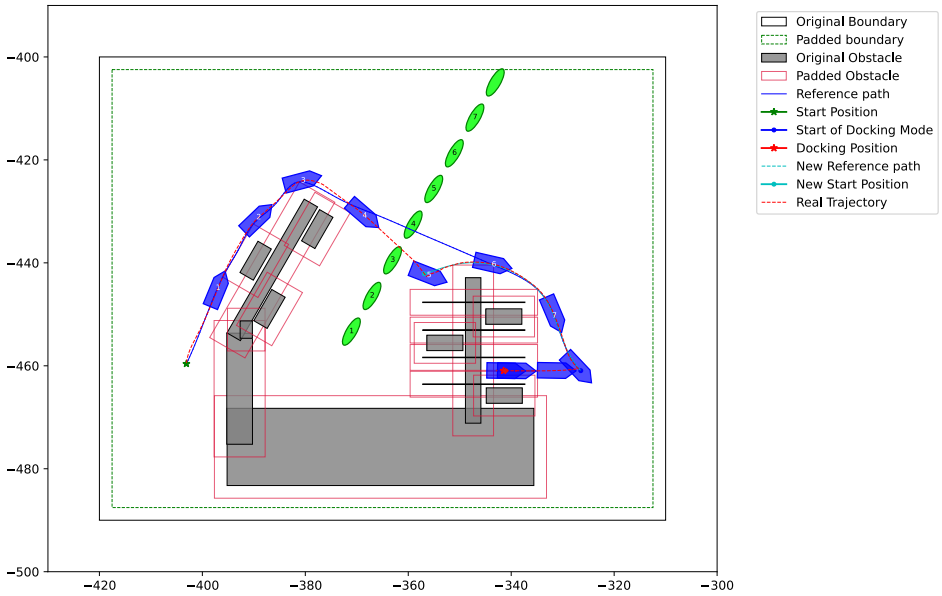


Figure 4.42: Simulated trajectory in dashed red and planned trajectory in blue for case 5 with a dynamic obstacle (green ellipse) entering from the left with an angle of 45° interfering with the planned path.

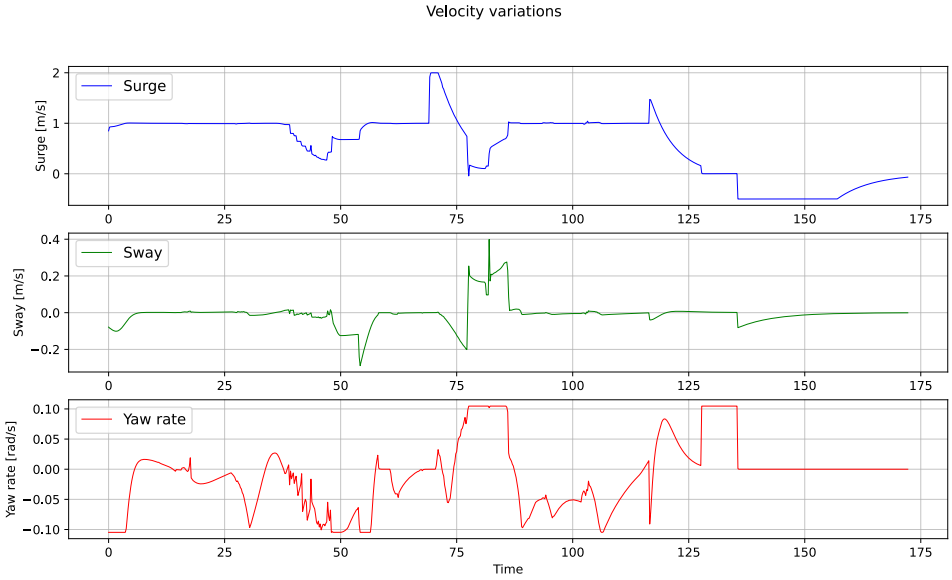


Figure 4.43: Linear and angular velocity for the simulated boat in case 5 for scenario 2.

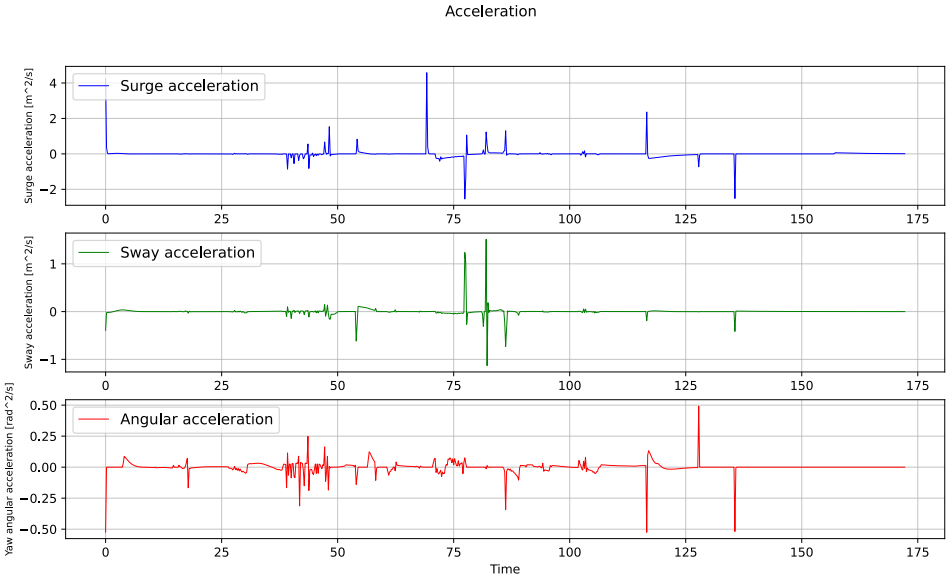


Figure 4.44: Linear and angular acceleration for the simulated boat in case 5 for scenario 2.

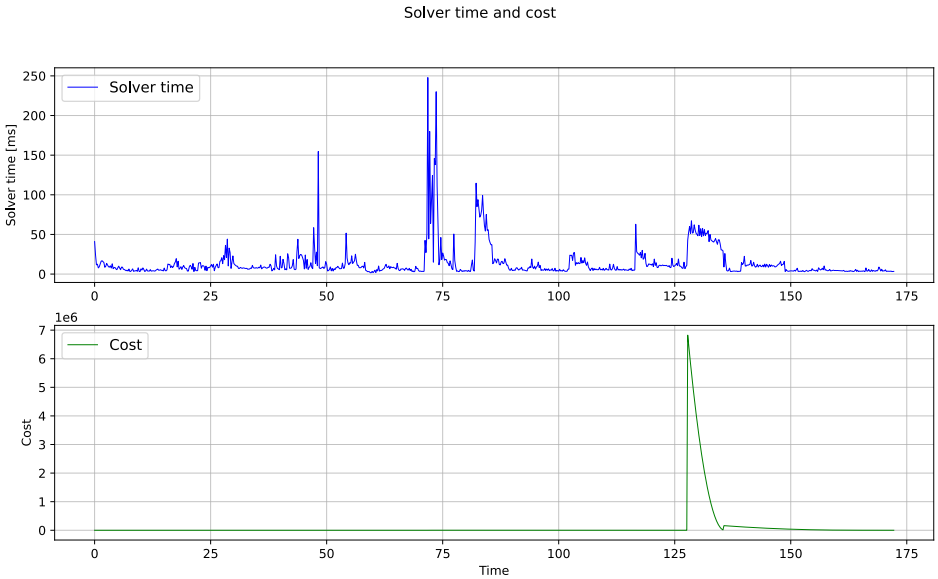


Figure 4.45: Solver time and cost at each time step during the simulation for case 5 scenario 2.

5
Discussion

The following chapter aims to analyze and reflect upon the performance of the path and trajectory planning systems implemented in this thesis. By evaluating the efficiency, limitations, and potential improvements of the path planner and the MPC across the different simulation environments, this chapter seeks to provide a comprehensive understanding of the autonomous docking system's capabilities and future areas of improvement.

5.1 Path planner

The Extremitypathfinder functioned as expected, by finding the shortest path from start to end. However, it did not take the dimensions of the vessel into account, potentially resulting in path choices unsuitable for vessel navigation. Figures 4.2c and 4.2d exemplify this issue, where the vessel would have been stuck in its starting position due to its inability to navigate between the obstacles identified by the path planner. Another issue is that since the path planner relies on a graph-based approach with connections made in straight lines, it overlooks dynamics. This means that the chosen paths required smoothing out for a more seamless and continuous trajectory. While the implemented smoothing algorithm addressed this, it occasionally exaggerated curves and produced rough edges as shown in Figure 4.2b.
These extravagant curves from the smoothing algorithm were the main reasons the autonomous docking algorithm was implemented with the docking mode separated from the reference path. This was to make the docking procedure more robust, as a docking scenario could involve scenarios, as in cases 1 and 2, where the docking spot is in a narrow space between two objects. The smoothing algorithm could make a curve too close to an object, making the cost of approaching the docking position too high for the MPC to finalize the docking. A more suitable path planner could operate using rapidly exploring random trees or Dubins curves which take the dynamics of the vessel into account, returning an appropriate path that is both user-oriented and trajectory-efficient.

5.2 Trajectory planner

The simulation results demonstrate a promising potential for the autonomous docking concept. However, key discrepancies emerged when comparing the Python and Unity environments, which need further investigation and refinement.

5.2.1 Comparative performance analysis of simulations in Python and Unity environments

Analyzing the trajectory following the performance of the vessel in both the Python and Unity environments reveals several differences. One major issue that can be observed in case 1, between the simulations in the Python and Unity environment, 4.3 and 4.7, is the MPC tendency to overly prioritize following the path rather than reaching the set goal destination. This results in the vessel halting its forward motion and moving in the vessel's lateral direction to realign with the path. The Python environment did not exhibit this problem as the MPC output is directly connected with the movement of the vessel and was able to follow the path with ease. In contrast, the Unity environment had a proportional controller in between and had to account for drag in the water, making the Unity environment more prone to drift away from the set path. The MPC was designed to minimize the deviation from the reference path, hence it would stop forward movement until the path had been reached again.

When comparing the performance of the vessel in different simulation environments, several key observations emerge; in general, the set and actual actions of the vessel were similar, but the actual actions exhibited more fluctuations and irregularities, as shown in the graph; for instance in Figure 4.8 and 4.16. These fluctuations and irregularities could be an inertia when switching directions; for example, if the propeller changes direction it takes roughly half a second before the new force is achieved. If the setpoint changes back during that time, causing the propeller to switch directions again this further extends the time required to stabilize. These fluctuations and irregularities could also be that the vessel's state 0 and the set state 1 as in table 3.1 are too close to each other such as the state error for the proportional controller was too small to be regulated. The difference in trajectories, where the Unity environment struggled to turn and drifted away from the set path, can be observed in all simulations, as shown in Figures 4.7, 4.15, 4.23, 4.31a and 4.31b. Notably, both environments handled case 3 with ease, where the task was to dock at an angle using only the lateral motion of the vessel to reach the docking spot, as shown in Figures 4.19 and 4.23.

When comparing the results between the two environments in the fourth case, a few differences can be observed. In case 4, the trajectory from the simulation in the Python environment follows the path well and docks effortlessly, see Figure 4.27. In contrast, the trajectories from the simulations in the Unity environment are less optimal as demonstrated in Figures 4.31a and 4.31b. Several factors can cause this difference. In case 4, the set surge velocity is relatively low at approximately 0.75 m/s, see Figure 4.35 and 4.32, whereas in case 1 and case 2, the surge velocities are 2.0 m/s and 1.5 m/s respectively, see Figures 4.8 and 4.16. By starting inside a dock position as in case 4, the vessel's starting point is in a narrow and obstacle-dense space and hence needs to turn slowly to get out safely. This means that for the more real-life setting simulated in Unity, the yaw set values were low, causing the vessel to miss turns and deviate from the path as the proportional controller in the twin

EVC installation was fed small errors to compensate for due to the low velocities, resulting in ineffective adjustments and low actual vessel velocities. This causes the vessel to not turn in time and deviate from the path, prompting the MPC to overcompensate in an attempt to get the vessel to return to the reference path. This overcompensation led to the vessel getting too close to objects, setting the trajectory on a downward spiral with low signals and worse compensations. The small errors in the proportional controller further contributed to the low velocities, making the MPC struggle to correct the course effectively.

By reviewing the plots of the cost for each simulation made in both environments, a distinct trend is observed. The high increase in the cost at the end of each simulation occurs when reaching the start of the docking mode. This spike occurs as the vessel approaches the docking position and initiates a rotation to align with the desired docking orientation. At this point, the MPC focuses on minimizing the angular difference between the vessel's current orientation and the target docking angle. Consequently, the cost surges at the beginning of the docking mode, reflecting the substantial deviation from the desired angle. This adjustment ensures that the MPC effectively prioritizes achieving an optimal docking orientation.

Further analysis of strange behaviors needs to be addressed, for example, the difference in the set and actual velocities/acceleration at the beginning of each simulation. The reason the actual value is zero at the beginning of all simulations, while the other MPC set signals remain constant, is that the automatic docking mode was not yet activated on the Unity-simulated vessel. Once the automatic docking mode was active, the simulated actual states were able to change.

Another notable observation is regarding the acceleration for all simulations across both environments. For the simulations in the Python environment, see Figure 4.5, 4.13, 4.21, 4.29, 4.40 and 4.44, distinct spikes in the acceleration are visible in all three directions, indicating abrupt changes in vessel motion. These abrupt accelerations are even more pronounced in the Unity simulations, see Figure 4.9, 4.17, 4.25, 4.36, and 4.33, which aim to represent real-life scenarios. These abrupt changes in acceleration contribute to unstable vessel motion, potentially compromising user comfort. The MPC is designed to mitigate these abrupt surges in acceleration by imposing a cost on the acceleration and control actions, providing smoother transitions, and reducing abrupt changes in velocity, see section 3.2.4. However, by reviewing the acceleration and velocity variations, it becomes evident that these cost functions may not be sufficient. To achieve smoother transitions and enhance vessel motion stability, further tuning of the acceleration and control costs is necessary. This can involve adjusting weight parameters to penalize changes in velocity more effectively or introducing hard constraints on acceleration to minimize abrupt fluctuations.

Despite these challenges, the vessel successfully docks in all simulations, albeit sometimes taking longer than desired. The docking time could potentially be decreased by implementing techniques such as minimum time control, however, due to the

time constraints of this research, this has not been tested and verified. Overall, the interplay between slow set velocities, obstacle proximity, and compensatory adjustments in the Unity environment results in less optimal trajectories compared to the Python environment, but still functional.

5.2.2 Performance analysis of dynamic obstacle avoidance

As no simulations were conducted of case 5 with dynamic obstacles in a more real-life setting such as the Unity environment, this analysis of the results is limited to the MPC's ability to avoid the dynamic obstacles in the Python environment. For this purpose, two scenarios were set up, as shown in Figure 3.14, to assess the boat's capability to safely divert from the reference path when a dynamic obstacle approaches the planned trajectory from different angles.
When entering the dynamic obstacle avoidance mode, the MPC starts avoiding the obstacle and planning an alternative path deviating from the reference after the obstacle has been diverted. This mode is activated when the obstacle is within a distance of two boat lengths of the vessel. As mentioned in the results, in this mode, the penalty weight for the cross-track error is set to zero allowing the MPC to focus solely on avoiding the dynamic obstacle, while still accounting for static obstacles and boundaries.

In both scenarios, the MPC successfully avoids the obstacle in time, preventing a collision, as demonstrated in Figures 4.38 and 4.42. In the second scenario, depicted in Figure 4.42, the vessel avoids the obstacle by deviating from the reference path by adjusting its heading and accelerating toward the dock. Despite this deviation, the vessel slows down sufficiently before reaching the dock, thereby avoiding a collision with the static obstacle, showcasing MPCs ability to avoid collision with both dynamic and static obstacles.
However, an examination of the surge velocity and acceleration of the vessel in Figures 4.43 and 4.44 reveals a significant spike in acceleration, and consequently an increase in velocity, at approximately 70 seconds. This indicates that the vessel accelerates abruptly to evade the dynamic obstacle, and its turn a more unstable vessel motion that may not provide much user comfort.
For a more stable control action, the MPC should ideally slow down the vessel, allowing the obstacle to pass. Once the obstacle is clear, the vessel should either resume following the reference path or, if necessary, allow the path planner to replan the reference path and proceed toward the goal. This approach would result in a more stable and smoother driving experience, enhancing comfort and safety for the user.

By reviewing the solver time plots for case 5, both scenarios reveal a significant increase in the solver time at specific intervals, approximately at 50-60 seconds in Figure 4.45 and shortly after 70 seconds in Figure 4.45. This increase indicates that it takes longer for the MPC to find the optimal solution to the optimization problem, i.e., the next control action. This is deemed a reasonable result, as these

intervals correspond to the moments in the simulation when the dynamic obstacle is safely diverted in each scenario. Consequently, the path planner begins replanning the reference path, and the vessel exits the dynamic obstacle avoidance mode to follow the newly planned path. In summary, the observed increase in solver time aligns with the critical moment of path replanning, confirming the MPC's handling of dynamic obstacle avoidance and subsequent path adjustment.

When analyzing the cost during the simulation for both scenarios, there is, similar to cases 1-4 in both simulation environments, a significant increase in cost at the end of each simulation. As written in 5.2.1, this increase occurs when reaching the docking mode and the vessel initiates the rotation to align with the desired docking orientation. If the angular difference between the current and desired docking orientations is large, then the cost will also be high. This is to get the next control actions from the MPC to minimize the angular difference and achieve the right orientation necessary to start the docking procedure.

The analysis of the dynamic obstacle avoidance technique for the autonomous docking system demonstrates the MPC's capability to successfully navigate around dynamic obstacles in the Python environment. The simulations of both scenarios confirm that the MPC effectively deviates from the reference path to avoid collisions, adjusting the vessel's heading and speed accordingly. However, the analysis also highlights certain drawbacks, such as abrupt acceleration and increased solver time during critical moments in the dynamic obstacle avoidance mode. These findings suggest that while the system can avoid dynamic and static obstacles, there are areas of improvement to enhance user comfort and control stability. Unlike cases 1-4, no simulations of case 5 with dynamic obstacles were tested in a real-life setting, such as the Unity environment, because of the time constraints of the thesis. Such simulations are necessary to better assess and do a comparative analysis of the dynamic obstacle avoidance performance and to refine the system's performance in more realistic scenarios.

5.2.3 Performance analysis of initial testing in real-world environment

Given the limited testing time frame, the conclusions of the system's performance in a real-life setting were constrained. However, some observations of the system's potential performance were made. The initial testing involved seeing how well the boat could follow a straight trajectory. These tests showcased high and abrupt accelerations, similar to the observations made from the Unity simulations, but even more pronounced in the real-life scenario. When tasked with moving between two points along a straight trajectory, the system encountered difficulties. While the path and trajectory planner performed as expected and successfully planned a reference path and suitable control actions, the issue arose when translating velocities to vessel motion. Specifically, the rotational velocities exceeded acceptable limits, causing the vessel to overshoot its intended heading. Consequently, the vessel rotated back and forth between rotations. On-site weight adjustments were attempted, as seen

in table 4.5, but they proved insufficient. To enhance the system's performance and ensure a smoother journey, more penalties or hard constraints need to be put on the accelerations and stricter velocity constraints are necessary. Furthermore, to be able to draw significant conclusions about the system's applicability in real life, more extensive testing of the algorithm is necessary.

5.2.4 Future works

The findings and analysis presented in this report underscore the potential of autonomous docking and collision avoidance using MPC. The thesis can contribute to several research areas, offering a foundation for further advancements and integration with other technologies, especially within the field of autonomous maritime navigation. The development and optimization of MPC-based autonomous docking systems contribute to the broader field of autonomous navigation, enhancing the safety and efficiency of maritime vessel operations. To further improve the stability and reliability of the autonomous docking system, several possibilities can be considered. Integrating sensor data from LiDAR or other advanced sensors, and by using information from nautical charts, a thorough mapping of the surrounding environment could be made. Such as measurements from sensors to get the distance from the vessel to each obstacle in the environment, both static and dynamic, and the use of nautical charts for identifying non-navigable areas such as shallow waters, shorelines, and rocks. This could be implemented with the current system by using the measurement data of the environment from sensors and nautical charts and converting it into lists of convex polygons of the obstacles that hinder navigable paths such that it is suitable to send to the MPC. Incorporating this into the MPC-based automated docking system could provide accurate and real-time updates about the vessels' surroundings and hence ensure a safer docking procedure.

The simulation results presented show that the MPC-based system shows potential in the field of autonomous docking in maritime environments. Even though some future work should be addressed for a more robust and practical automatic docking algorithm, the system holds the potential for adding more features, such as the ones mentioned above, to get broader applicability. However, the analysis of the results also shows that several areas require further research and development to enhance system performance, user comfort, and real-world applicability. These improvements can be made in each building block for the whole algorithm, path planner, trajectory planner, and dynamic object avoidance. Below, some of the key future research areas and possibilities for improvement of these blocks are presented.

5.2.4.1 Path planner

While the proposed path planner works for the current model, it is not an ideal fit. The graph-based planner returns the shortest Euclidean distance, which complicates the trajectory planner's task of enforcing a smooth and continuous trajectory. This requires the vessel to stop at each path planner point to stay on course. Although the proposed method for smoothing the path helps, it sometimes results in overly exaggerated curves or sharp edges.

A more suitable path planner would consider incorporating the vessel dynamics. For example, using a Rapidly Exploring Random Tree (RRT) or Dubins path could yield a more appropriate trajectory. Investigating and developing these models could produce more feasible paths by accounting for the vessel's dimensions to avoid generating paths that are impossible for the vessel to navigate. These methods can find paths that, while not necessarily the shortest, feature smooth curves and can accommodate complex maneuvers such as changing from forward motion to reverse, facilitating easier docking.

5.2.4.2 Trajectory planner

The vessel model's flexible movements, including sideways motion in the vessel's lateral direction, presented challenges in driving mode, which prioritizes forward motion to align with typical vessel operation. The trajectory planner often attempted to use both forward and sideways movements, i.e. surge and sway motions, simultaneously increasing the total velocity. Despite the cost function prioritizing forward motion, velocities in the vessel's lateral direction were still occasionally used in driving mode. A proposed solution is to disable this motion in driving mode, ensuring purely forward motion in the boat's longitudinal direction, and re-enable movement in the vessel's lateral direction when entering docking mode for more precise maneuvers.

Another area for improvement is the total time needed from start to stop. Although the algorithm successfully docks the vessel, it is somewhat slow. Implementing a technique called minimum time control could address this issue by optimizing the vessel's trajectory to achieve docking in the shortest possible time. This approach ensures that the vessel operates at its maximum permissible speeds and accelerations within the given constraints, reducing unnecessary delays and enhancing the efficiency of the docking process.

To increase the control stability and user comfort of the automated docking system, further research on incorporating control strategies allowing for smoother velocity control is necessary. Implementing mechanisms to reduce the abrupt changes in acceleration and velocity seen in the dynamic obstacle avoidance mode could provide a smoother and more comfortable experience for the users of the systems. Enhancing the path planners to more effectively replan the reference path post-obstacle avoidance could help ensure smoother transitions back to the desired trajectory and optimizing the MPC solver to reduce the computational time during critical intervals could also ensure more reliable and stable control actions. Optimizing the tuning of the proportional controller could also help to better handle small errors and improve the vessel's response to control signals, especially in the presence of inertia and drag.

To further enhance the system's applicability, incorporating nautical laws and charts into the algorithm holds the potential for expanding its capabilities. By integrating nautical laws, a cost function could be constructed to ensure that the trajectory planner follows the set laws and regulations, promoting safe and legal navigation when operated. Utilizing nautical charts to map the environment, cost functions can be modeled to penalize proximity to hazards such as shallow waters, safety zones, or

restricted areas. This information would be distinct from obstacle avoidance costs, allowing different weights for various non-navigable areas. A higher cost would be assigned to avoid collision with an obstacle compared to entering a restricted zone, enhancing overall safety and compliance.

Finally, a transition from simulations to conducting further and more extensive real-world testing of the algorithm is necessary to validate the system's performance in actual maritime environments, accounting for real-world factors such as wind, currents, and varying water conditions.

6
Conclusion

The research conducted in this thesis addresses the challenges in the field of autonomous docking of marine vessels, particularly focusing on the integration of path planning and trajectory planning in an environment with static and dynamic obstacles and narrow passages. The proposed system demonstrates the potential to automate the docking procedure while ensuring safety and efficiency by combining path planning algorithms with MPC.

The analysis of the path planner's performance revealed its effectiveness in identifying optimal routes. However, the A* algorithm's inability to account for the vessels's dynamics and dimensions resulted in impractical paths in the presence of narrow passages. An additional smoothing algorithm to improve upon the A*-path, while helpful, occasionally introduced exaggerated curves, necessitating a more sophisticated path planning approach for further improvement. Future research could explore alternatives such as RRT or Dubins curve, which both incorporate the vessel dynamics and do not only search for the shortest path toward the goal position.

The performance of the trajectory planner highlighted the system's capability to navigate in a busy marine environment by following the reference path, avoiding obstacles, and docking a marine vessel in narrow spaces. However, discrepancies between the Python and Unity simulations, particularly regarding path deviations, underscore the need for further refinements to increase the stability and functionality of the algorithm's performance in real-world scenarios. The tendency to overly prioritize the reference path in the Unity environment suggests that incorporating the more robust control strategies and further tuning of the controller could enhance the stability and responsiveness. The MPC's ability to safely react to dynamic obstacles shows promising results, although the system's abrupt accelerations indicate areas of improvement. More stable control actions could enhance user comfort and overall system stability.

Further research should focus on refinements to the path and trajectory planning algorithms to allow for more complex maritime environments and vessel dynamics. This also includes further and more extensive testing in complex real-world scenarios to validate the system's functionality in actual maritime environments, accounting for real-world factors such as wind, currents, and varying water conditions.

Finally, this thesis contributes to the research and advancement of autonomous docking for marine vessels, providing a foundation for further research and development

in this area. The findings highlight the importance of integrating advanced path planning algorithms and control strategies, such as MPC, to achieve robust and stable autonomous docking solutions. With continued research and further real-world validation, the proposed solution holds promise for enhancing the safety and efficiency of maritime operations and the integration of additional technology, such as sensor inputs for environmental mapping.

Bibliography

[1] Ade Candra, Mohammad Andri Budiman, and Kevin Hartanto. "Dijkstra's and a-star in finding the shortest path: A tutorial". In: *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*. IEEE. 2020, pp. 28–32.

[2] Max Lutz and Thomas Meurer. "Optimal trajectory planning and model predictive control of underactuated marine surface vessels using a flatness-based approach". In: *2021 American Control Conference (ACC)*. IEEE. 2021, pp. 4667–4673.

[3] Andreas B Martinsen, Anastasios M Lekkas, and Sebastien Gros. "Autonomous docking using direct optimal control". In: *IFAC-PapersOnLine* 52.21 (2019), pp. 97–102.

[4] Dae Jung Kim, Yong Woo Jeong, and Chung Choo Chung. "Lateral Vehicle Trajectory Planning Using a Model Predictive Control Scheme for an Automated Perpendicular Parking System". In: *IEEE Transactions on Industrial Electronics* 70.2 (2023), pp. 1820–1829. DOI: 10.1109/TIE.2022.3163567.

[5] Simon Helling and Thomas Meurer. "Dual collision detection in model predictive control including culling techniques". In: *IEEE Transactions on Control Systems Technology* (2023).

[6] Muhammad Awais Abbas, Ruth Milman, and J Mikael Eklund. "Obstacle avoidance in real time with nonlinear model predictive control of autonomous vehicles". In: *Canadian journal of electrical and computer engineering* 40.1 (2017), pp. 12–22.

[7] Muhammad Rhifky Wayahdi, Subhan Hafiz Nanda Ginting, and Dinur Syahputra. "Greedy, A-Star, and Dijkstra's algorithms in finding shortest path". In: *International Journal of Advances in Data and Information Systems* 2.1 (2021), pp. 45–52.

[8] Alessandro Gasparetto et al. "Path planning and trajectory planning algorithms: A general overview". In: *Motion and Operation Planning of Robotic Systems: Background and Practical Approaches* (2015), pp. 3–27.

[9] Min Luo, Xiaorong Hou, and Jing Yang. "Surface optimal path planning using an extended Dijkstra algorithm". In: *Ieee Access* 8 (2020), pp. 147827–147838.

[10] Amit Patel. *Introduction to A**. 30/6. 2024. URL: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[11] Peter E. Hart, Nils J. 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. DOI: 10.1109/TSSC.1968.300136.

[12] Marija Seder and Ivan Petrovic. “Dynamic window based approach to mobile robot motion control in the presence of moving obstacles”. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. 2007, pp. 1986–1991. DOI: 10.1109/ROBOT.2007.363613.

[13] P. Ogren and N.E. Leonard. “A convergent dynamic window approach to obstacle avoidance”. In: *IEEE Transactions on Robotics* 21.2 (2005), pp. 188–195. DOI: 10.1109/TRO.2004.838008.

[14] David Q Mayne et al. “Constrained model predictive control: Stability and optimality”. In: *Automatica* 36.6 (2000), pp. 789–814.

[15] James Blake Rawlings, David Q. Mayne, and Moritz M. Diehl. *Model predictive control: Theory, computation, and design*. 2nd. Nob Hill Publishing, 2019.

[16] Bo Egardt. *Model Predictive Control*. 2020.

[17] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. *Predictive control for linear and hybrid systems*. Cambridge University Press, 2017.

[18] Xin-She Yang. *Engineering mathematics with examples and applications*. Academic Press, 2016.

[19] Bo Edgart and Sebastian Gros. *Modelling And Simulation*. 2022.

[20] Frank Allgower, Rolf Findeisen, Zoltan K Nagy, et al. “Nonlinear model predictive control: From theory to application”. In: *Journal-Chinese Institute Of Chemical Engineers* 35.3 (2004), pp. 299–316.

[21] Jiayu Fan, Nikolce Murgovski, and Jun Liang. *Efficient optimization-based trajectory planning*. 2023. arXiv: 2312.17440 [cs.RO].

[22] Optimization Engine. *The Structure of OpEn*. 04/11. 2024. URL: https://alphaville.github.io/optimization-engine/docs/open-intro#the-structure-of-open.

[23] Lorenzo Stella et al. “A Simple and Efficient Algorithm for Nonlinear Model Predictive Control”. In: *arXiv preprint* (Sept. 2017). arXiv: 1709.06487v1 [math.OC].

[24] COIN-OR. *IPOPT Documentation*. 04/11. 2024. URL: https://coin-or.github.io/Ipopt/.

[25] Ben Hermans, Panagiotis Patrinos, and Goele Pipeleers. “A Penalty Method Based Approach for Autonomous Navigation using Nonlinear Model Predictive Control”. In: (May 2018).

[26] GitHub-name MrMinimal64. *Extremitypathfinder*. Accessed: May 03, 2024. 2024. URL: %5Curl%7Bhttps://pypi.org/project/extremitypathfinder/%7D.

A

Appendix 1

A.1 Pseudocode for the A* algorithm

Algorithm 1 A* Algorithm

```
1: function A*(start, goal)
2:     closedset ← the empty set                ▷ The set of nodes already evaluated.
3:     openset ← {start}                        ▷ The set of nodes to be evaluated.
4:     came_from ← the empty map                ▷ The map of navigated nodes.
5:
6:     estimated total cost from start to goal.
7:     g[start] ← 0                             ▷ Cost from start along best known path.
8:     f[start] ← g[start] + h_estimate(start, goal)
9:
10:    while openset is not empty do
11:        current ← the node in openset having the lowest f
12:        if current = goal then
13:            return RECONSTRUCT_PATH(came_from, goal)
14:        end if
15:        remove current from openset
16:        add current to closedset
17:        for each neighbor in NEIGHBOR_NODES(current) do
18:            if neighbor in closedset then
19:                continue
20:            end if
21:
22:            g_tentative ← g[current] + dist_between(current, neighbor)
23:
24:            if neighbor not in openset or g_tentative < g[neighbor] then
25:                came_from[neighbor] ← current
26:                g[neighbor] ← g_tentative
27:                f[neighbor] ← g[neighbor]+heuristic_cost_estimate(neighbor, goal)
28:                if neighbor not in openset then
29:                    add neighbor to openset
30:                end if
31:            end if
32:        end for
33:    end while
34:    return failure
35: end function
36:
37: function RECONSTRUCT_PATH(came_from, current)
38:    total_path ← [current]
```

A.2 Pseudocode for the Djikstra algorithm

Algorithm 2 Dijkstra Algorithm

```
1:  function DIJKSTRA(Graph, Source)
2:      create vertex set D
3:      for each vertex v in Graph do:
4:          distance[v] ← INFINITY
5:          previous[v] ← UNDEFINED
6:          add v to D
7:      end for
8:      distance[source] ← 0
9:
10:     while D is not empty do
11:         u ← vertex in D with min distance[u]
12:         remove u from D
13:         for each neighbor v if u still in D do
14:             alt ← distance[u] + length(u, v)
15:             if alt < distance[v] then
16:                 distance[v] ← alt
17:                 previous[v] ← u
18:             end if
19:         end for
20:     end while
21:     return distance[], previous[]
22: end function
```