Bachelor's thesis

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

Lauritz Rismark Fosso
Kristian Aasmundstad Johannesen
Pål Kristoffer Kjærem
Tor Harald Staurnes

Decentralized Model Predictive Control for Increased Autonomy in Fleets of ROVs

Bachelor's thesis in Electrical Engineering
Supervisor: Josef Matous
May 2023

NTNU
Norwegian University of
Science and Technology

Lauritz Rismark Fosso
Kristian Aasmundstad Johannesen
Pål Kristoffer Kjærem
Tor Harald Staurnes

Decentralized Model Predictive Control for Increased Autonomy in Fleets of ROVs

Bachelor's thesis in Electrical Engineering
Supervisor: Josef Matous
May 2023

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

NTNU
Norwegian University of
Science and Technology

Acknowledgement

This thesis marks the culmination of a three-year journey, filled with valuable experiences and exciting challenges.

We would like to show gratitude to Damiano Varagnolo for challenging us along the way and inspiring us with his ambitious ideas and broad experience.

We would also like to thank Josef Matous for his guidance throughout this project, and for giving us sound advice and inspiring us.

A special thanks is in order to Mikael Medina, for giving us access to his Gazebo simulator environment and setting aside time to assist us. His help has been invaluable to this thesis.

And for our friends and family who have been supporting us along the way. Thank you.

Abstract

This thesis aims to document the design and implementation of a decentralized model predictive control architecture in a set of BlueROV2 Heavys. The thesis covers the mathematical modelling of the ROV, the design of the MPC, its implementation in ROS 2, and simulations of the final product.

The mathematical model is in the form of Fossen's robot-inspired matrix-vector model for marine craft. The model is described matrix by matrix, and parametrized using both quaternions and Euler angles. An alternative model that accounts for irrotational ocean currents is also presented.

The ROVs communicate through optical sensors, and therefore the line of sight between the ROVs is a prerequisite for communication. In that context, a model predictive controller is designed that achieves this by knowing the other ROV's position. An alternative cost function, that locates all ROVs at the edge of a circle, defined by a radius and a point is also derived.

Collision avoidance and precise trajectory following are also a necessity to enable cooperation in multi-agent systems, where several agents are in close proximity to one another. The controller was designed to handle this, in addition to maintaining line of sight.

The implementation was done in ROS 2 and a graphical user interface was developed with two main control modes, one for *trajectory planning* and one for *joystick control*. A mode for running standardised tests was also developed.

The simulations were performed both for Python and ROS 2 with Gazebo, and a set of four tests were designed to evaluate the controllers' performance. These tests were run under different conditions, such as ocean currents, packet loss, and modifications of system parameters. Each test was conducted 100 times with two agents in Gazebo.

The test results proved the controller capable. With less than 1% failure rate in collision avoidance, and path-following capabilities. However the results show that the controller did experience challenges with maintaining line of sight, and thus complete robustness in was not achieved. The Python simulator, however, did not experience any violation of the constraints, with the trajectory of the ROVs closely following the reference.

Sammendrag

Denne oppgaven dokumenterer design og implementering av en desentralisert modellprediktiv reguleringsarkitektur i et sett med BlueROV2 Heavy. Oppgaven dekker matematisk modellering av undervannsfarkosten, design av MPC, implementering i ROS 2 og simulering av sluttproduktet.

Den matematiske modellen er uttrykt på form av Fossens robotinspirerte matrise-vektormodell for marinefartøy. Modellen blir gjennomgått matrise for matrise, og er parametrisert både med kvaternioner og Euler vinkler. En alternativ modell som tar hensyn til ikke roterende havstrømmer presenteres også.

De fjernstyrte undervannsfarkostene kommuniserer gjennom optiske sensorer, og derfor er siktlinje mellom farkostene en forutsetning for kommunikasjon. I den sammenhengen er en modellprediktiv regulator utformet som oppnår dette kun ved å ha tilgang til den andre farkostens posisjon. En alternativ kostfunksjon, som plasserer alle farkostene ved randen av en sirkel, definert av en radius rundt ett punkt, utledes også.

Antikollisjonssystem og presis banefølging er også en nødvendighet i fleragentsystemer, der flere enheter er i umiddelbar nærhet av hverandre. Regulatoren var designet for å håndtere dette, i tillegg til å holde siktlinjen mellom undervannsfarkostene.

Systemet ble implementert i ROS 2, og det ble utviklet en grafisk brukergrensesnitt med to hovedkontrollmoduser, én for baneplanlegging og én for joystick-kontroll. I tillegg ble det satt opp en modus for diverse standard tester.

Simuleringene ble utført i både Python og ROS 2 med Gazebo, og fire tester ble designet for å evaluere regulatoren ytelse. Disse testene ble kjørt under forskjellige forhold, som for eksempel med havstrømmer, pakketap, og modifikasjoner av systemparametere. Hver test ble utført 100 ganger med to agenter i Gazebo.

Resultatene viste at regulatoren fungerte. Antikollisjonssystemet sviktet i mindre enn 1% av testene, og regulatoren fulgte banene gitt. Likevel viste resultatene at regulatoren strevde med å opprettholde siktlinje, og fullstendig robusthet ble dermed ikke oppnådd. Python-simulatoren opplevde imidlertid ingen brudd på begrensningene som ble satt, og banen til undervannsfarkostene fulgte referansen.

Contents

Terminology and Glossary

Glossary

BlueROV2 ROV produced by Blue Robotics.

BlueROV2 Heavy BlueROV2 with extra thrusters.

do-mpc An open-sourced python package used for model predictive control.

Gazebo An third-party application used for simulation in robotics.

Raspberry Pi 4 A small Linux-based computer.

Acronyms

CB Center of Buoyancy.

CG Center of Gravity.

DoF Degrees of Freedom.

ECEF Earth-Centered Earth-Fixed.

ECI Earth-Centered Inertial.

FOV Field of view.

GUI Graphical User Interface.

ITK Department of Engineering Cybernetics (Institutt for Teknisk Kybernetikk).

LED Light Emitting Diode.

MPC Model Predictive Control.

MSROV Mid-sized Remote Operated underwater Vehicle.

NED North-East-Down.

NTNU Norwegian University of Science and Technology (Norges teknisk-naturvitenskapelige universitet).

OCROV Observational Class Remote Operated underwater Vehicle.

ROS Robot Operating System.

ROV Remotely Operated underwater Vehicle.

UUV Unmanned Underwater Vehicle.

WCROV Working Class Remote Operated underwater Vehicle.

Figures

Tables

Chapter 1

Introduction

1.1 Background

The ocean covers over 70% of the Earth‘s surface, yet more than 80% of the ocean remains unobserved, unmapped, and unexplored [1]. Exploring the ocean will bring about increased knowledge of its ecosystem, and can lead to new discoveries, that can be the basis for new inventions within a wide range of areas, such as medicine and technology that mimic adaptations of deep-sea animals. The use of underwater drones, known as Unmanned Underwater Vehicles (UUVs) play a vital role in tackling this challenging and dangerous environment. For example, the immense pressures at the ocean floor in and of itself render it unreachable by humans alone.

A popular class of Unmanned Underwater Vehicles are Remotely Operated underwater Vehicles, or ROVs. These are used in a variety of industries such as oil and gas, fishing, exploration and scientific research [2, p. 16-19]. ROVs perform tasks such as inspecting underwater infrastructure, photographing and recording underwater life, ship wreckages etc.

Many ROVs currently on the market are operated by a pilot that controls the vehicle manually. The next step in their evolution is to achieve a greater degree of autonomy and to enable intelligent cooperation to handle more complicated tasks, for example picking up larger objects, like loose fishing nets or other items drifting with the current.

This thesis explores a method for increasing autonomy and cooperation between ROVs and is part of a larger research project currently running at the Department of Engineering Cybernetics (Institutt for Teknisk Kybernetikk), NTNU, with the goal of obtaining increased knowledge and developing methods for increasing the autonomy of ROVs.

1.2 Problem Statement

In order to achieve increased autonomy in fleets of ROVs and effectively handle complex assignments, it is crucial for the units to operate as a cohesive whole. With this objective in mind, the controller was assigned the following tasks:

1. Communication between ROVs
2. Collision avoidance between ROVs
3. Path following

Communication between ROVs: The ROVs associated with the research project was planned to communicate through optical communication, which requires a line of sight to exchange data. Line of sight is therefore a prerequisite for communication.

Collision avoidance between ROVs: To ensure that the ROVs do not collide when operating as a multi-agent unit, collision avoidance is a necessity for the ROVs to be able to safely operate in proximity to one another.

Path following: The ROVs should be able to follow a predetermined path and generate trajectories from waypoints given by the operator.

The controller proposed in this thesis will perform the aforementioned tasks on a set of two ROVs in a decentralized control system.

The performance of the controller is then assessed by running numerous simulations under various conditions, to analyse its robustness.

1.3 Report structure

This report is divided into chapters, each focusing on a distinct area of the project. They are designed to be readable independently for easier reference in the future.

Chapter 2 Describes the work process, different challenges encountered, and solutions to said challenges.

Chapter 3 Introduces ROVs, first explaining what they are and the different classifications, and then presenting the specific vehicle that this project is utilising.

Chapter 4 Contains the mathematical framework necessary to create a model of the ROV's dynamics. The Mathematical prerequisites required are presented before the ROV's equations of motion are derived. Lastly, uncertainties surrounding the model are discussed.

Chapter 5 Introduces the applied method of control, Model Predictive Control, describes the design of the MPC and concludes with discussion of the design choices

Chapter 6 Describes the project's implementation in ROS 2. First, some necessary concepts regarding the use of ROS 2 are explained. Then, the project architecture is presented with results, before analysing and discussing about its implementation.

Chapter 7 Details the simulators used in this thesis and a set of standardised tests then used to analyse the robustness of the simulators.

Chapter 8 Concludes the thesis with the group's final thoughts and gives recommendations for future work.

1.4 Source Code

The code and data connected to this thesis are available on GitHub[3].
BlueROV2 Garden [4], a Gazebo workspace used in this thesis, is not available to the public as it is used in research at ITK.

Chapter 2

Project Development

To acquire fundamental knowledge on ROVs and underwater robotics, the project began with literary research of the relevant topics. Additionally, the group intended to replicate the results of the previous year's Bachelor thesis titled *Implementation of a quaternion-based PD controller in ROS2 for a generic underwater vehicle with six degrees of freedom* [5] using a ROV in a testing pool. This would have provided a starting point for the current project and insight into the larger ongoing research project. In preparation to reproduce the result, the group read the thesis and held a meeting with one of the authors. However, access to a ROV was not granted at this point, resulting in the re-creation not taking place.

The next step was selecting a specific theme for the project. The group was presented with two options: either replace the single-agent controller used in the previous year's project [5], or expand the control to a multi-agent system. Following consultations with the supervisor and client and internal group discussions, the latter option was chosen.

After deciding on the direction for the project, the group began literary research on multi-agent control to find a possible solution for the problem. During this phase, with a recommendation from the client, it was decided to create a Model Predictive Controller (MPC) with the do-mpc toolbox. The group was not familiar with MPC, so in-depth research on MPC was needed.

At this stage, the end objective was to evaluate the system using physical ROVs. However, the availability of the vehicles was uncertain, posing a significant challenge to the system's development and result collection. To address this issue, the group reached out to Mikael Andreas Medina. Medina had, at the time of writing, been developing an environment for Gazebo to simulate BlueROV2 Heavy submerged in water [4]. By gaining access to this environment, alongside a do-mpc-based Python simulator created by the group, the system could get tested during the development phase and a contingency plan for result collection was in place.

To optimise the development process, the group was divided into two teams and assigned each team specific tasks, allowing each team to focus on one or two aspects of the project at a time. Additionally, the group held weekly meetings with the supervisor and client to review progress and discuss potential solutions to any issues that surfaced. If needed, the group held additional meetings with the supervisor or other members of ITK for specific issues and possibilities.

Towards the end of the project, the group started the design of various tests, which were then tested with the Gazebo simulation. However, an attempt was made to facilitate a test of the system with physical ROVs. The initial proposal was to test the system with two vehicles, but it became clear that gaining access to even one vehicle was uncertain, and certainly not two. If access to a single vehicle were granted, the group acknowledged one vehicle would not adequately test the multi-agent part of the project. To solve this issue the group proposed to test the system with one physical vehicle and one virtual vehicle in the simulator. The primary obstacle for accessing the ROVs is that the department's vehicles have incomplete hardware modifications. These challenges and a recommendation from the supervisor and client led to the decision to continue testing the system with the simulation and only focus on that.

To test the system the group developed a GUI and finished the design of multiple test conditions for the simulation. Each test condition was conducted numerous times to collect enough data to adequately represent the system developed. This data was processed and the resulting graphs are presented in this thesis.

Chapter 3

Remotely Operated Underwater Vehicles

3.1 Introduction

Remotely Operated underwater Vehicles (ROVs) are a type of Unmanned Underwater Vehicle (UUV), more specifically they are highly manoeuvrable, remotely controlled, tethered underwater robots [2, p. 3-5]. The ROVs are used in many different fields such as scientific research, the fishing industry and the oil and gas industry [2, p. 17-18], to mention a few. This chapter introduces the important aspects of how ROVs are built and how they function. Following the general description of ROVs, the BlueROV2 and its' capabilities are presented. The vehicle this thesis is based on is the BlueROV2.

3.2 ROVs in General

3.2.1 Size-Classification

ROVs come in a variety of different sizes and specifications that can be grouped into four main classes according to *The ROV Manual, A User Guide for Remotely Operated Vehicles* [2, p. 5-8]. These classes are:

- *Observational Class Remote Operated underwater Vehicle (OCROV)*: Contains the smallest micro-ROVs to vehicles weighing up to 100kg.
- *Mid-sized Remote Operated underwater Vehicle (MSROV)*: Contains vehicles weighing between 100 kg and 1000 kg.
- *Working Class Remote Operated underwater Vehicle (WCROV)*: These vehicles generally are connected to high-voltage AC power supplies from the surface.
- *Special-use vehicles*: This group contains vehicles that do not fall under the three aforementioned classes.

3.2.2 Communication

The vast majority of ROVs use a tether for communication and data flow. The main reason for this is that only lower-frequency waves penetrate water, and the decreased frequency negatively affects the data rate [2, p. 66]. ROVs are dependent on a high data rate to get a high-resolution live video feed to the operator. However, it is worth mentioning that there are options to get tetherless ROVs such as acoustic and optical modems [6].

Optical modems function by emitting light from a light emitter, usually a laser diode or LED, to a light receiver [7]. For bidirectional communication between ROVs each vehicle must have an emitter and a receiver [7]. Since the communication is through light, the modems must be in the line of sight. This is the bases of this project's FOV constraint which will be explored further in Chapter 5.1.9 and 5.2.2.

3.2.3 Propulsion

The propulsion is a significant part of the ROV. The most common form of propulsion is electrically or hydraulically driven propellers, but there are other types as well such as jets [2, p. 122-124] and belts for crawling on the seafloor [2, p. 7] [8]. The type of thruster used depends on the ROVs use case. For example, ROVs with heavy-duty tooling, hydraulically driven propellers might be advantageous [2, p. 123], but smaller observational vehicles have electrically driven propellers. The number of thrusters also varies, where more thrusters generally lead to higher manoeuvrability.

3.2.4 Buoyancy

Since the ROVs are submerged in water, the vehicles' buoyancy becomes an important factor. Archimedes' principle states that *An object immersed in a fluid experiences a buoyant force that is equal in magnitude to the force of gravity on the displaced fluid* [2, p. 109]. It is common for the ROV to be slightly positively buoyant to ensure the vehicle returns to the surface if a power failure occurs. Positive buoyancy also allows for near-bottom operation because the ROV does not need to thrust upwards, forcing water down and stirring up sediment that can cover the camera [2, p. 119].

The buoyant force has a Center of Buoyancy (CB) which it acts upon, the same way gravity acts upon Center of Gravity (CG). ROVs are designed with positively buoyant material on top and negatively buoyant material on the bottom. This puts the CB high and CG low to make the vehicle stable [2, p. 74 and 116-117].

3.3 BlueROV2

The BlueROV2 is a highly customizable and manoeuvrable OCROV. The vehicle is available with two possible thruster configurations, the base model with six thrusters and the heavy configuration with eight [9]. This project uses the version with heavy configuration therefore the rest of the information given is based on that configuration.

3.3.1 Thruster Configuration and DoF

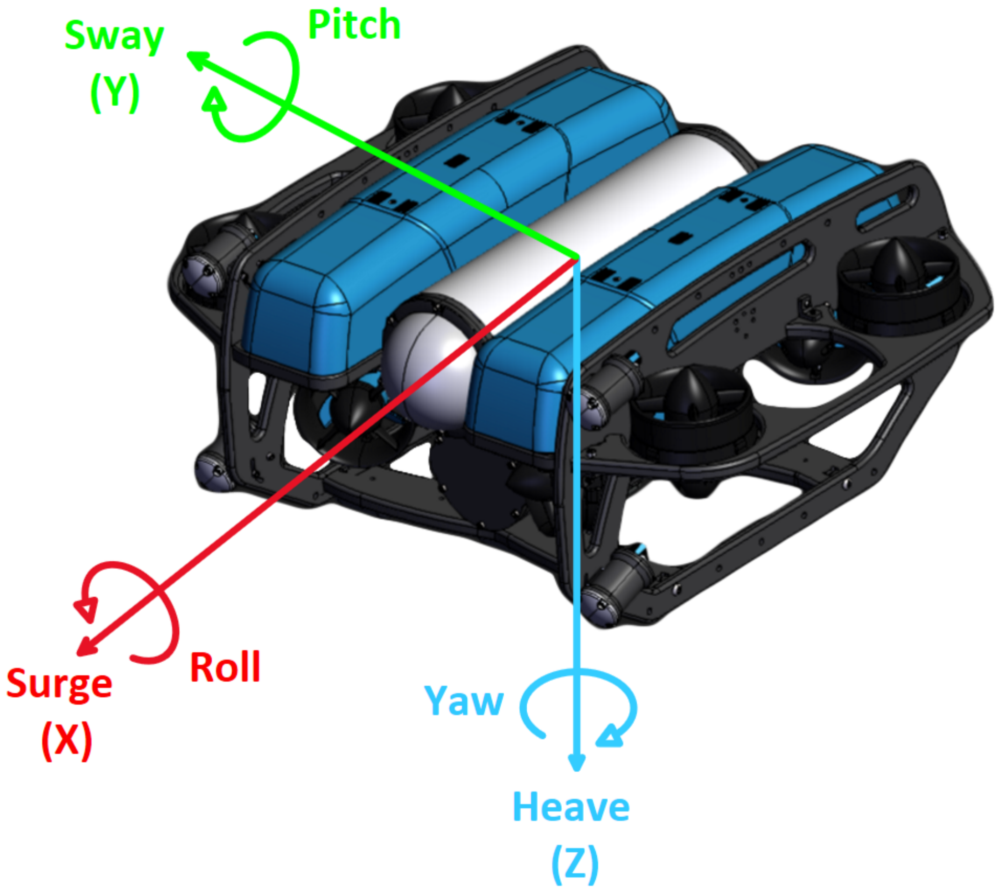


Figure 3.1: BlueROV2 Heavy [10] 6 DoF coordinate system

The BlueROV2 Heavy configuration has eight thrusters, four vectored thrusters and four vertical thrusters [9, 11]. The vectored thrusters are installed in the xy-plane of the vehicle allowing for linear movement in the surge and sway direction as well as yaw rotation. The vertical thrusters allow linear movement in the heave direction, and roll and pitch rotation. This configuration gives the vehicle 6 DoF as seen in Figure 3.1. [9, 11]. The thrusters produce a maximum forward thrust of 88.3 N, which allows the vehicles to reach a maximum velocity of 1.5 m/s [9]. The maximum vertical velocity is not listed, however, in this project it is assumed the same velocity vertically as horizontally. This assumption is based on the higher vertical thrust of 137.3 N [9] counteracting the drag from the larger surface area.

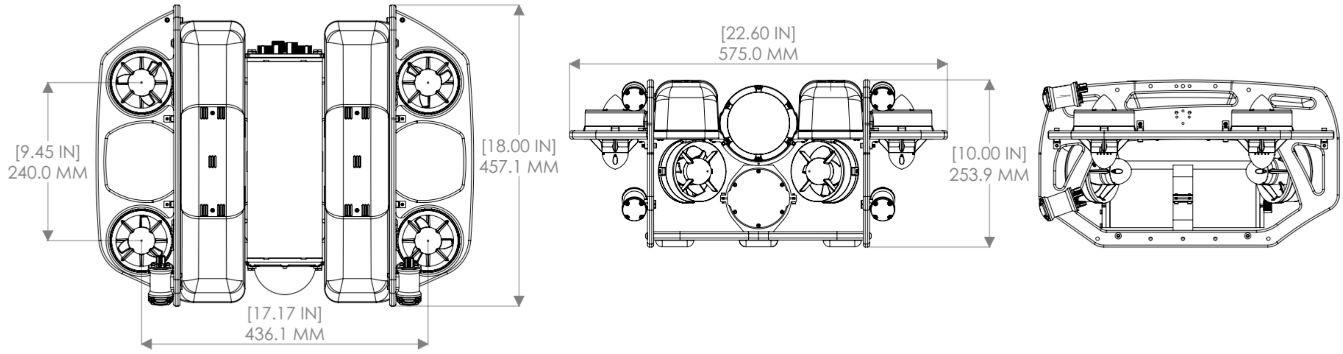


Figure 3.2: 2D drawings of the BlueROV2 Heavy configuration [11]

3.3.2 Components and Sensors

The BlueROV2 has a 1080p HD camera with 110° Field of view and ± 90° tilt for live video streaming to the operator. The ROV is equipped with a gyroscope, accelerometer and magnetometer where each sensor has 3 DoF, totalling 9 DoF for positioning [12]. All processing and computing are done by a Raspberry Pi 4 which runs on the open-source software BlueOS. The ROV uses a tether with ethernet communication between the vehicle and the operator [9]. It is worth mentioning that the BlueROV2 Heavy belonging to ITK is modified, however, this does not affect this project since all tests are done in a simulator.

Chapter 4

Mathematical Model

4.1 Introduction

The mathematical model is essential to the implementation of model-based controllers such as MPC, and its accuracy is a critical factor in the performance of the controller. This chapter presents the mathematical framework needed to model ROVs.

Firstly, some mathematical prerequisites are introduced, namely *Euler angles*, *quaternions* and *skew-symmetry*. Then the relevant reference frames, and a list of the notation used throughout. Lastly, the equations of motion, which are in the form of Fossen's robot-inspired model for marine craft are introduced [13, p. 15]. The equations of motion are presented and explained matrix by matrix, and relevant assumptions and simplifications are made. The uncertainties associated with the constants used will also be discussed.

The model implemented is a modified version of the one presented by Wu [14]. The details surrounding the modifications are described in 4.8 and 4.7.4.

4.2 Euler Angles

Euler angles are used to parametrize a 3D rotation with three distinct rotations performed in a sequence. There are two main categories of Euler angles: Proper Euler angles, and Tait-Bryan angles. These two methods differ in the number of distinct axes used. Proper Euler angles are only parametrized around two distinct axes, whereas Tait-Bryan angles are parametrized around three. Equation (4.1) displays a Tait-Bryan Z-Y-X parametrization. That is, first a rotation ψ around the z-axis, then a rotation θ around the y-axis and finally, a rotation ϕ around the x-axis. These rotations define the roll ϕ, pitch θ, and yaw ψ [15, p. 56].

$$Z_\psi Y_\theta X_\phi = \begin{bmatrix} c_\psi c_\theta & c_\psi s_\theta s_\phi - c_\phi s_\psi & s_\psi s_\phi + c_\psi c_\phi s_\theta \\ c_\theta s_\psi & c_\psi c_\phi + s_\psi s_\theta s_\phi & c_\phi s_\psi s_\theta - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{4.1}$$

Here, s and c are shorthand for sine and cosine.

4.3 Quaternions

Quaternions, first described in 1843 by the Irish mathematician William Rowan Hamilton, are an expansion of two-dimensional complex numbers into four dimensions. Quaternions are widely used in applications such as 3D graphics rendering [16] and attitude tracking since, unlike Euler angles, they do not contain singularities, which cause a phenomenon known as “gimbal lock” when the ROV is oriented at certain angles [13, p. 32]. Gimbal lock causes the craft to lose a degree of freedom, or in 3D graphics can cause peculiar rotations. Another distinct advantage quaternions have over Euler angles is that they are computationally more efficient.

A quaternion is described by a real number, and three imaginary numbers referred to as a *vector*. A quaternion can be written on the form:

$$q = \eta + \epsilon_1 i + \epsilon_2 j + \epsilon_3 k \tag{4.2}$$

Or, alternatively on vector form:

$$q = \begin{bmatrix} \eta & \epsilon_1 & \epsilon_2 & \epsilon_3 \end{bmatrix}^\top \tag{4.3}$$

Through this, it is possible to describe a 3D rotation by letting the vector describe the axis of rotation and the real part describe the angle of rotation.

4.3.1 Unit Quaternions

Unit quaternions, by definition, satisfy Equations (4.4), (4.5) and (4.6)

$$\eta^2 + \epsilon_1^2 + \epsilon_2^2 + \epsilon_3^2 \triangleq 1 \tag{4.4}$$

$$\eta \triangleq cos\left(\frac{\beta}{2}\right) \tag{4.5}$$

$$\epsilon \triangleq \lambda sin\left(\frac{\beta}{2}\right) \tag{4.6}$$

where λ is a unit vector, and β is the angle of rotation [17].

4.3.2 Hamilton Product

Quaternion multiplication, also called the *Hamilton product* is defined in (4.7).

$$\begin{aligned} q_1 \otimes q_2 = & \left(\eta_1\eta_2 - \epsilon_{1,1}\epsilon_{2,1} - \epsilon_{1,2}\epsilon_{2,2} - \epsilon_{1,3}\epsilon_{2,3}\right) \\ & + \left(\eta_1\epsilon_{2,1} + \epsilon_{1,1}\eta_2 + \epsilon_{1,2}\epsilon_{2,3} - \epsilon_{1,3}\epsilon_{2,2}\right) i \\ & + \left(\eta_1\epsilon_{2,2} - \epsilon_{1,1}\epsilon_{2,3} + \epsilon_{1,2}\eta_2 + \epsilon_{1,3}\epsilon_{2,1}\right) j \\ & + \left(\eta_1\epsilon_{2,3} + \epsilon_{1,1}\epsilon_{2,2} - \epsilon_{1,2}\epsilon_{2,1} + \epsilon_{1,3}\eta_2\right) k \end{aligned} \tag{4.7}$$

Where $q_i = \begin{bmatrix} \eta_i & \epsilon_{i,1} & \epsilon_{i,2} & \epsilon_{i,3} \end{bmatrix}$. Alternatively, if q_1 and q_2 are unit quaternions, on the vector form (4.8)

$$q_1 \otimes q_2 = \begin{bmatrix} \eta_1\eta_2 - \boldsymbol{\epsilon_1}^\top \boldsymbol{\epsilon_2} \\ \eta_1\boldsymbol{\epsilon_2} + \eta_2\boldsymbol{\epsilon_1} + S\left(\epsilon_1\right)\boldsymbol{\epsilon_2} \end{bmatrix} \tag{4.8}$$

where $\boldsymbol{\epsilon_1}$, and $\boldsymbol{\epsilon_2}$ are respectively the vector part of q_1 and q_2 [13, p. 33].

4.3.3 Quaternion Inverse

The inverse of a quaternion, $q^{-1} = \bar{q}$, is defined as

$$q^{-1} = \frac{q^*}{||q||^2} \tag{4.9}$$

where q^* is the conjugate of q defined as $q^* = \begin{bmatrix} \eta & -\epsilon_1 & -\epsilon_2 & -\epsilon_3 \end{bmatrix}^\top$ and $||q||^2 = \sqrt{\eta^2 + \epsilon_1^2 + \epsilon_2^2 + \epsilon_3^2}$ For unit quaternions $||q||^2 = 1 \Rightarrow q^{-1} = q^*$

4.3.4 Quaternion Rotations

Quaternion rotations are written in the form of Equation (4.10), where $\tilde{q}$ is the rotated quaternion, v is a *pure* quaternion (its real part equals zero) that one wishes to rotate, q is the rotation, and $\bar{q} = q^{-1}$.

$$\tilde{q} = qv\bar{q} \tag{4.10}$$

4.3.5 Attitude Control Using Quaternions

Controlling attitude[1] using quaternions requires a different, perhaps less intuitive approach than one would employ when using Euler angles. Equations (4.11) and (4.12) display the approach to quaternion attitude control, where q is the attitude and q_d is the desired attitude. Equation (4.11) expresses the error between q_d and q. Perfect setpoint regulation is expressed in Equation (4.12), when the vector of $\tilde{q} = 0_{3\times1}$ [18].

$$\tilde{q} = \bar{q}_d q \tag{4.11}$$

$$q = q_d \iff \tilde{q} = \begin{bmatrix} \pm1 \\ 0 \end{bmatrix} \tag{4.12}$$

4.4 Skew-symmetric Operator

Skew symmetric matrices are matrices in which

$$A = -A^\top \tag{4.13}$$

holds. An alternative method to compute the cross-product of two vectors is to use the skew-symmetric operator, $S(a)$, $S \in SS(3)$ [13, p. 24].

$$a \times b \triangleq S(a)b \tag{4.14}$$

$$S(a) = -S(a)^\top = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{4.15}$$

Skew-symmetric matrices can also be used in the context of expressing the derivative of a rotation matrix, $\dot{R}(\Theta)$

$$\dot{R}(\Theta) = R(\Theta)S(\omega) \tag{4.16}$$

where ω are the angular velocities and Θ are the Euler angles .

[1] Here, attitude carries the same meaning as *orientation*, which includes roll, pitch and yaw.

4.5 Reference Frames

It is necessary to define two reference frames in order to describe the ROV's dynamics, and more are needed to track its position and attitude in space. In this project, the BODY and NED frames are utilised, but the reader is introduced to the ECEF and ECI frames in order to get a broader picture of the hierarchy of reference frames.

BODY Frame The equations of motion describe the ROV's dynamics in its BODY frame. Which is a local frame that is fixed to the geometrical centre of the ROV, with its x-, y- and z-axis always pointing in the surge, sway and heave direction respectively [13, p. 20], as shown in Figure 4.1

NED Frame The NED frame or the North-East-Down, which, as its naming would suggest, is a Geographic reference frame located tangential to the surface of the earth, with its x-axis pointing true North, y-axis pointing East, and z-axis pointing down towards the centre of the earth [13, p. 20].

ECEF Frame The Earth-Centered Earth-Fixed frame has its origin at the centre of the Earth and follows the Earth's rotation. Through the ECEF frame, the NED frame's origin is expressed with longitude and latitude [13, p. 19].

ECI Frame The Earth-Centered Inertial frame has, like the ECEF frame, its origin at the centre of the Earth, but it does not follow the Earth's rotation, it is rather fixed in space [13, p. 19].

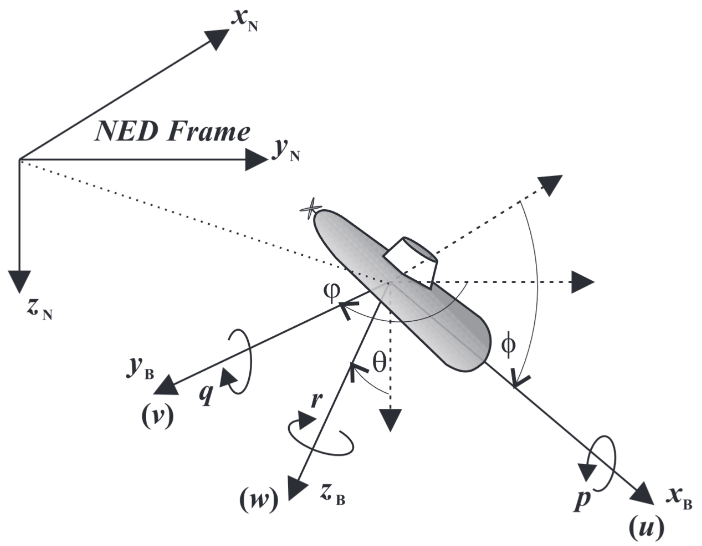


Figure 4.1: Illustration of NED and BODY reference frames [19]

4.6 Notation

The notation used to describe the ROV's dynamics in this report follows the SNAME convention, shown in Table 4.1

Table 4.1: SNAME naming convention for marine craft [13, p. 18]

BODY			NED
DoF	Forces and moments	Linear and angular velocities	Position and Euler angles
Surge	X	u	x
Sway	Y	v	y
Heave	Z	w	z
Roll	K	p	ϕ
Pitch	M	q	θ
Yaw	N	r	ψ

Table 4.2: Notation

Description	Parameter
NED frame position	$p = \begin{bmatrix} x & y & z \end{bmatrix}^\top$
NED frame Euler angles	$\Theta = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^\top$
NED frame quaternion orientation	$q = \begin{bmatrix} \eta & \epsilon_1 & \epsilon_2 & \epsilon_3 \end{bmatrix}^\top$
NED frame position and orientation	$\eta = \begin{bmatrix} p & q \end{bmatrix}^\top$
BODY frame linear velocity	$v = \begin{bmatrix} u & v & w \end{bmatrix}^\top$
BODY frame angular velocity	$\omega = \begin{bmatrix} p & q & r \end{bmatrix}^\top$
BODY frame velocity vector	$\nu = \begin{bmatrix} v & \omega \end{bmatrix}^\top$
Gravitational force	W
Buoyancy force	B
Origin of the body frame	**CO**
Center of Gravity relative to CO	$\mathbf{CG} = r_g = \begin{bmatrix} x_g & y_g & z_g \end{bmatrix}^\top$
Center of Buoyancy relative to CO	$\mathbf{CB} = r_b = \begin{bmatrix} x_b & y_b & z_b \end{bmatrix}^\top$

Table 4.3: Constants [14]

Linear damping			Quadratic damping			Misc.		
Parameter	Value	Unit	Parameter	Value	Unit	Parameter	Value	Unit
X_u	-4.03	$\frac{\text{Ns}}{\text{m}}$	$X_{u\|u\|}$	-18.18	$\frac{\text{Ns}^2}{\text{m}^2}$	m	11.5	kg
Y_v	-6.22	$\frac{\text{Ns}}{\text{m}}$	$Y_{v\|v\|}$	-21.66	$\frac{\text{Ns}^2}{\text{m}^2}$	W	112.8	N
Z_w	-5.18	$\frac{\text{Ns}}{\text{m}}$	$Z_{w\|w\|}$	-36.99	$\frac{\text{Ns}^2}{\text{m}^2}$	B	114.8	N
K_p	-0.07	$\frac{\text{Ns}}{\text{rad}}$	$K_{p\|p\|}$	-1.55	$\frac{\text{Ns}^2}{\text{rad}^2}$	I_x	0.16	kgm^2
M_q	-0.07	$\frac{\text{Ns}}{\text{rad}}$	$M_{q\|q\|}$	-1.55	$\frac{\text{Ns}^2}{\text{rad}^2}$	I_y	0.16	kgm^2
N_r	-0.07	$\frac{\text{Ns}}{\text{rad}}$	$N_{r\|r\|}$	-1.55	$\frac{\text{Ns}^2}{\text{rad}^2}$	I_z	0.16	kgm^2
						r_g	$\begin{bmatrix}0 & 0 & 0.02\end{bmatrix}^\top$	m
						r_b	$\begin{bmatrix}0 & 0 & 0\end{bmatrix}^\top$	m

Added mass		
Parameter	Value	Unit
$X_{\dot{u}}$	-5.5	Kg
$Y_{\dot{v}}$	-12.7	Kg
$Z_{\dot{w}}$	-14.57	Kg
$K_{\dot{p}}$	-0.12	$\frac{\text{Kgm}^2}{\text{rad}}$
$M_{\dot{q}}$	-0.12	$\frac{\text{Kgm}^2}{\text{rad}}$
$N_{\dot{r}}$	-0.12	$\frac{\text{Kgm}^2}{\text{rad}}$

4.7 Equations of Motion

To derive the equations of motion, one must study the ROV's rigid-body kinetics, hydrodynamics and hydrostatics. The goal is to write the equations of motion on the form:

$$M_{RB}\dot{\nu} + M_A\dot{\nu} + C_{RB}(\nu)\nu + C_A(\nu)\nu + D_L(\nu)\nu + D_{NL}(\nu)\nu + g(\eta) = \tau \tag{4.17}$$

where M_{RB} is the rigid-body system inertia matrix, $C_{RB}(\nu)$ is the rigid-body Coriolis-centripetal matrix, M_A and C_A are, respectively, the added mass and Coriolis-centripetal matrices. $D_L(\nu)$ and $D_{NL}(\nu)$ are the matrices that describe the linear and the nonlinear damping. Finally, τ is the vector forces and moments acting upon the ROV [13, p. 15]. Equation (4.17) is in the form of Fossen's robot-inspired matrix-vector model for marine craft, based on the robot model

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} = \tau \tag{4.18}$$

to create a compact form to describe the dynamics of a 6-DoF marine craft. Here, q denotes the joint angles of the robot[13, p. 15].

4.7.1 Rigid-body Kinetics

The rigid-body system inertia matrix, M_{RB}, is shown in (4.19) [13, p. 64]. Refer to Chapter 3 of Fossen's Marine Craft Hydrodynamics and Motion Control for the derivation of M_{RB}.

$$M_{RB} = \begin{bmatrix} mI_3 & -mS(r_b) \\ mS(r_b) & I_b \end{bmatrix} \tag{4.19}$$

Where I_b is the inertia dyadic

$$I_b = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{yx} & I_y & -I_{yz} \\ -I_{zx} & -I_{zy} & I_z \end{bmatrix} \tag{4.20}$$

with $I_{ab} = \int_V ab\rho_m dV$, where ρ_m is the density. I_a is the moment of inertia about the axis a [13, p. 59]. Expanding (4.19) yields (4.21).

$$M_{RB} = \begin{bmatrix} m & 0 & 0 & 0 & mz_g & -my_g \\ 0 & m & 0 & -mz_g & 0 & mx_g \\ 0 & 0 & m & my_g & -mx_g & 0 \\ 0 & -mz_g & my_g & I_x & -I_{xy} & -I_{xz} \\ mz_g & 0 & -mx_g & -I_{yx} & I_y & -I_{yz} \\ -my_g & mx_g & 0 & -I_{zx} & -I_{zy} & I_z \end{bmatrix} \tag{4.21}$$

The ROV can be considered symmetric along both the xz-plane and xy-plane. Furthermore, the CO is placed at the geometrical centre of the ROV. The products of inertia I_{xy}, I_{xz} and I_{yz}, will become zero because of these symmetries [14], [13, p. 201]. Then $r_g = \begin{bmatrix} x_g & y_g & z_g \end{bmatrix}^\top = \begin{bmatrix} 0 & 0 & z_g \end{bmatrix}^\top$ as a result of these symmetries. M_{RB} can be simplified accordingly, as shown in Equation (4.22) [13, p. 201] [14].

$$M_{RB} = \begin{bmatrix} m & 0 & 0 & 0 & mz_g & 0 \\ 0 & m & 0 & -mz_g & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & -mz_g & 0 & I_x & 0 & 0 \\ mz_g & 0 & 0 & 0 & I_y & 0 \\ 0 & 0 & 0 & 0 & 0 & I_z \end{bmatrix} \tag{4.22}$$

The Coriolis-centripetal matrix, C_{RB}, as its name suggests, expresses how the Coriolis and centripetal forces affect the ROV's kinetics. Deriving this matrix can be done using an energy-based approach: $T = \frac{1}{2}\nu^\top M \nu$, where T is the kinetic energy, $\nu = \begin{bmatrix} \nu_1 & \nu_2 \end{bmatrix}^\top$. Differentiating with respect to ν_1 and ν_2, and finally using Kirchoff's equation [13, p. 66] gives:

$$M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \tag{4.23}$$

$$C(\nu) = \begin{bmatrix} 0_{3\times3} & -S(M_{11}\nu_1 + M_{12}\nu_2) \\ -S(M_{11}\nu_1 + M_{12}\nu_2) & -S(M_{21}\nu_1 + M_{22}\nu_2) \end{bmatrix} \tag{4.24}$$

Using the Lagrangian parametrization shown in Equation (4.24), C_{RB} is computed from M_{RB}, resulting in Equation (4.25)

$$C_{RB}(\nu) = \begin{bmatrix} 0 & 0 & 0 & 0 & mw & mpz_g - mv \\ 0 & 0 & 0 & -mw & 0 & mu + mqz_g \\ 0 & 0 & 0 & mv - mpz_g & -mu - mqz_g & 0 \\ 0 & mw & mpz_g - mv & 0 & I_z r & -I_y q - muz_g \\ -mw & 0 & mu + mqz_g & -I_z r & 0 & I_x p - mvz_g \\ mv - mpz_g & -mu - mqz_g & 0 & I_y q + muz_g & mvz_g - I_x p & 0 \end{bmatrix} \tag{4.25}$$

It should be mentioned that this parametrization is not unique, there are other matrices, $C(\nu)$ that will yield the same product $C(\nu)\nu$ [13, p. 67]. Alternatively, it is also possible to parametrize C_{RB} such that it is independent of the linear velocities of the ROV. This is particularly useful when representing the equations of motion in terms of the relative velocities ν_r. This is elaborated further upon in 4.7.7.

$$C_{RB}(\nu) = \begin{bmatrix} mS(\nu_2) & -mS(\nu_2)S(r_g) \\ mS(r_g)S(\nu_2) & -S(I_b\nu_2) \end{bmatrix} \tag{4.26}$$

Here, $\nu_2 = \omega$. Expanding (4.26) and using the same symmetry assumptions as in (4.22), yields (4.27)

$$C_{RB}(\nu) = \begin{bmatrix} 0 & -m\,r & m\,q & m\,r\,z_g & 0 & 0 \\ m\,r & 0 & -m\,p & 0 & m\,r\,z_g & 0 \\ -m\,q & m\,p & 0 & -m\,p\,z_g & -m\,q\,z_g & 0 \\ -m\,r\,z_g & 0 & m\,p\,z_g & 0 & I_z\,r & -I_y\,q \\ 0 & -m\,r\,z_g & m\,q\,z_g & -I_z\,r & 0 & I_x\,p \\ 0 & 0 & 0 & I_y\,q & -I_x\,p & 0 \end{bmatrix} \tag{4.27}$$

Both (4.25) and (4.27) result in the same product $C_{RB}(\nu)\nu$ and as such, either of these parametrizations can be utilised.

4.7.2 Hydrostatics

The generalised restoring force vector $g(\eta)$ expresses how the gravitational force, W and buoyancy force, B affect the craft. It is, as such, dependent on η: the vector for position and attitude. $g(\eta)$ for a submerged vehicle can be expressed as shown in Equation [13, p. 72].

$$g(\eta) = -\begin{bmatrix} R^{\top}(q)\,(f_g + f_n) \\ r_g \times R^{\top}(q)f_g + r_b \times R^{\top}(q)f_b \end{bmatrix} \tag{4.28}$$

Where

$$f_g = \begin{bmatrix} 0 & 0 & W \end{bmatrix}^{\top},\ f_b = -\begin{bmatrix} 0 & 0 & B \end{bmatrix}^{\top} \tag{4.29}$$

Expanding this expression yields 4.30.

$$g(\eta) = \begin{bmatrix} (B-W)\,\sigma_2 \\ (B-W)\,\sigma_3 \\ -(B-W)\,\sigma_1 \\ W\,y_g\,\sigma_1 - B\,z_b\,\sigma_3 - B\,y_b\,\sigma_1 + W\,z_g\,\sigma_3 \\ B\,x_b\,\sigma_1 + B\,z_b\,\sigma_2 - W\,x_g\,\sigma_1 - W\,z_g\,\sigma_2 \\ B\,x_b\,\sigma_3 - B\,y_b\,\sigma_2 - W\,x_g\,\sigma_3 + W\,y_g\,\sigma_2 \end{bmatrix} \tag{4.30}$$

where $\sigma_1 = 2\,\epsilon_1{}^2 + 2\,\epsilon_2{}^2 - 1,\ \ \sigma_2 = 2\,\epsilon_1\,\epsilon_3 - 2\,\epsilon_2\,\eta,\ \ \sigma_3 = 2\,\epsilon_2\,\epsilon_3 + 2\,\epsilon_1\,\eta,$

Since $r_b = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^{\top}$ and $r_g = \begin{bmatrix} 0 & 0 & z_g \end{bmatrix}^{\top}$ Equation (4.30) can be simplified, as expressed in Equation (4.31).

$$g(\eta) = \begin{bmatrix} (B-W)\big(2\,\epsilon_1\,\epsilon_3 - 2\,\epsilon_2\,\eta\big) \\ (B-W)\big(2\,\epsilon_2\,\epsilon_3 + 2\,\epsilon_1\,\eta\big) \\ -(B-W)\big(2\,\epsilon_1{}^2 + 2\,\epsilon_2{}^2 - 1\big) \\ W\,z_g\,\big(2\,\epsilon_2\,\epsilon_3 + 2\,\epsilon_1\,\eta\big) \\ -W\,z_g\,\big(2\,\epsilon_1\,\epsilon_3 - 2\,\epsilon_2\,\eta\big) \\ 0 \end{bmatrix} \tag{4.31}$$

4.7.3 Hydrodynamics

As the ROV moves, its dynamics are affected by the fluid it moves through. The motion of the ROV will induce a motion in the fluid, as it has to move aside for the ROV to pass and fill the vacuum left behind it. The fluid, as a consequence, possesses kinetic energy that affects the ROV's dynamics [13, p. 143 - 144] The relationship between the kinetic fluid energy T_A is expressed in

(4.32), where M_A is the added mass matrix. Deriving M_A can be done by inserting (4.32) into Kirchoff's equation [13, p. 145-146]

$$T_A = \frac{1}{2}\nu^{\top} M_A \nu \tag{4.32}$$

$$M_A = -\begin{bmatrix} X_{\dot{u}} & X_{\dot{v}} & X_{\dot{w}} & X_{\dot{p}} & X_{\dot{q}} & X_{\dot{r}} \\ Y_{\dot{u}} & Y_{\dot{v}} & Y_{\dot{w}} & Y_{\dot{p}} & Y_{\dot{q}} & Y_{\dot{r}} \\ Z_{\dot{u}} & Z_{\dot{v}} & Z_{\dot{w}} & Z_{\dot{p}} & Z_{\dot{q}} & Z_{\dot{r}} \\ K_{\dot{u}} & K_{\dot{v}} & K_{\dot{w}} & K_{\dot{p}} & K_{\dot{q}} & K_{\dot{r}} \\ M_{\dot{u}} & M_{\dot{v}} & M_{\dot{w}} & M_{\dot{p}} & M_{\dot{q}} & M_{\dot{r}} \\ N_{\dot{u}} & N_{\dot{v}} & N_{\dot{w}} & N_{\dot{p}} & N_{\dot{q}} & N_{\dot{r}} \end{bmatrix} \tag{4.33}$$

The added mass terms are highly coupled and non-linear, especially when the ROV moves at high velocities. However, when the ROV moves at lower velocities, combined with the symmetries of the ROV, the off-diagonal elements of M_A can be neglected. In practice, the diagonal approximation is found to be reasonably accurate [13, p. 148]. The elements in (4.34) can be read as $Y_{\dot{u}}$: the change in added mass force Y in direction y, with respect to the change of velocity u (velocity in the x direction) [13, p. 144].

$$M_A = -diag(X_{\dot{u}}, Y_{\dot{v}}, Z_{\dot{w}}, K_{\dot{p}}, M_{\dot{q}}, N_{\dot{r}}) \tag{4.34}$$

Similarly to C_{RB}, C_A, the hydrodynamic Coriolis-centripetal matrix can be computed through M_A, using the Lagrangian parametrization shown in (4.24) [13, p. 146].

$$C_A(\nu) = \begin{bmatrix} 0 & 0 & 0 & 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v \\ 0 & 0 & 0 & Z_{\dot{w}}w & 0 & -X_{\dot{u}}u \\ 0 & 0 & 0 & -Y_{\dot{v}}v & X_{\dot{u}}u & 0 \\ 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v & 0 & -N_{\dot{r}}r & M_{\dot{q}}q \\ Z_{\dot{w}}w & 0 & -X_{\dot{u}}u & N_{\dot{r}}r & 0 & -K_{\dot{p}}p \\ -Y_{\dot{v}}v & X_{\dot{u}}u & 0 & -M_{\dot{q}}q & K_{\dot{p}}p & 0 \end{bmatrix} \tag{4.35}$$

Linear hydrodynamic damping for underwater vehicles is mainly caused by skin friction. Similarly to M_A, the off-diagonal elements of the linear damping matrix, D_L can be neglected for craft moving at low speeds [13, p. 151].

$$D_L(\nu) = -diag(X_u, Y_v, Z_w, K_p, M_q, N_r) \tag{4.36}$$

Non-linear damping is caused by phenomena such as vortex-shedding. Equation (4.37) is a rough approximation of the non-linear damping matrix for an ROV, neglecting the off-diagonal elements [13, p. 196].

$$D_{NL}(\nu) = -diag(X_{u|u|}\,|u|, Y_{v|v|}\,|v|, Z_{w|w|}\,|w|, K_{p|p|}\,|p|, M_{q|q|}\,|q|, N_{r|r|}\,|r|) \tag{4.37}$$

4.7.4 Control Forces and Moments

The control input vector τ represents the forces and moments acting upon the ROV from its thrusters [13, p. 233]. For a craft with 6 DoF, τ for one thruster can be expressed as shown in (4.38).

$$\tau = \begin{bmatrix} f_t \\ r_t \times f_t \end{bmatrix} = \begin{bmatrix} F_x \\ F_y \\ F_z \\ l_y F_z - l_z F_y \\ l_z F_x - l_x F_z \\ l_x F_y - l_y F_x \end{bmatrix} \tag{4.38}$$

Where $f_t = \begin{bmatrix} F_x & F_y & F_z \end{bmatrix}^\top$ is the thrust vector, and $r_t = \begin{bmatrix} l_x & l_y & l_z \end{bmatrix}^\top$ is the vector of thruster arms, the distance from the thrusters, to the CO along the x-, y- and z-axes. The approach used and parameters chosen are taken from Wu [14], with the only modification being the positive thruster direction for thrusters 3 and 4. This was done so that the positive thruster direction would correspond with the Gazebo simulator. The control input vector, τ can be further broken down into (4.39)

$$\tau = Bu_e, \qquad B = T_e K_e \tag{4.39}$$

where T_e is the extended thrust configuration matrix, K_e is the extended thrust coefficient matrix and u_e is the vector of inputs. T_e is found by using vector decomposition for each thruster. Thrusters 1 and 4 are angled at $\frac{\pi}{4} rad$, while Thrusters 2 and 3 are angled at $-\frac{\pi}{4} rad$. Thrusters 5, through 8 are mounted orthogonally to the xy-plane, with thrusters 6 and 7's positive direction defined along the positive z-axis, while thrusters 5 and 8 have defined positive direction along the negative z-axis.

$$T_e = \begin{bmatrix} 0.7071 & 0.7071 & 0.7071 & 0.7071 & 0 & 0 & 0 & 0 \\ -0.7071 & 0.7071 & 0.7071 & -0.7071 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 \\ 0.0601 & -0.0601 & -0.0601 & 0.0601 & -0.2180 & -0.2180 & 0.2180 & 0.2180 \\ 0.0601 & 0.0601 & 0.0601 & 0.0601 & 0.1200 & -0.1200 & 0.1200 & -0.1200 \\ -0.1888 & 0.1888 & -0.1888 & 0.1888 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.40}$$

$K_e \in \mathbb{R}^{8\times 8}$ is a diagonal matrix that contains the gain of each input. For simulation purposes using MPC, this matrix is less critical, as the purpose of this matrix is to map the thruster input signal to the thruster output force. When simulating, there are no physical restrictions to u_e. Thus, the identity matrix $K_e = I_{8\times 8}$ was chosen, with saturation $u_{e,sat} \pm 6.2$ as this resulted in a top speed of 1.5 m/s. This corresponds with the information in the datasheet for the ROV. For real systems, K_e should be mapped such that $K_i\, u_{e,max} = F_{i,max}$, where $F_{i,max}$ is the maximum force thruster i can produce.

4.7.5 Tranformation From BODY To NED

Currently, the dynamics of the ROV are expressed in its own BODY frame, and it is necessary to express the dynamics in the NED frame, two matrices are required to accomplish this.

Linear Velocity Transformation

The linear velocity transformation from the BODY frame to the NED frame is expressed in (4.41), where $\dot{p}$ is the change in position in the NED frame, $R(q)$ is the quaternion derived rotation matrix, (4.43) [13, p. 34], and v is the linear velocity vector in the BODY frame.

$$\dot{p} = R(q)v \tag{4.41}$$

$$R(q) = I_3 + 2\eta S(\epsilon) + 2S(\epsilon)^2 \tag{4.42}$$

$$R(q) = \begin{bmatrix} 1-2(\epsilon_2^2+\epsilon_3^2) & 2(\epsilon_1\epsilon_2-\epsilon_3\eta) & 2(\epsilon_1\epsilon_3+\epsilon_2\eta) \\ 2(\epsilon_1\epsilon_2+\epsilon_3\eta) & 1-2(\epsilon_1^2+\epsilon_3^2) & 2(\epsilon_2\epsilon_3-\epsilon_1\eta) \\ 2(\epsilon_1\epsilon_3-\epsilon_2\eta) & 2(\epsilon_2\epsilon_3+\epsilon_1\eta) & 1-2(\epsilon_1^2+\epsilon_2^2) \end{bmatrix} \tag{4.43}$$

Angular Velocity Transformation

The angular velocity rotation matrix, $T(q)$ is used to transform the angular velocity from BODY to NED. ω is the angular velocity vector. A definition for $T(q)$ is given in 4.44 [13, p. 35].

$$T(q) = \frac{1}{2}\begin{bmatrix} -\epsilon^{\top} \\ \eta I_3 + S(\epsilon) \end{bmatrix} \tag{4.44}$$

Expanding (4.44) yields (4.45)

$$T(q) = \frac{1}{2}\begin{bmatrix} -\epsilon_1 & -\epsilon_2 & -\epsilon_3 \\ \eta & -\epsilon_3 & \epsilon_2 \\ \epsilon_3 & \eta & -\epsilon_1 \\ -\epsilon_2 & \epsilon_1 & \eta \end{bmatrix} \tag{4.45}$$

Finally, $\dot{q}$, the change in attitude in the NED frame, is expressed in (4.46)

$$\dot{q} = T(q)\omega \tag{4.46}$$

Linear and Angular Transformation

The linear and angular transformation matrix, J_q, which combines (4.43) and (4.45) is expressed in Equation (4.47).

$$J_q(\eta) = \begin{bmatrix} R(q) & 0_{3\times3} \\ 0_{4\times3} & T(q) \end{bmatrix} \tag{4.47}$$

$$\dot{\eta} = \begin{bmatrix} \dot{p} \\ \dot{q} \end{bmatrix} = J_q(\eta)\nu \tag{4.48}$$

4.7.6 Complete Model

The end result becomes (4.49)

$$\begin{bmatrix} \dot{\eta} \\ \dot{\nu} \end{bmatrix} = \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{\nu} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} J_q(\eta)\nu \\ -(M_{RB} + M_A)^{-1}(C_{RB}(\nu)\nu + C_A(\nu)\nu + D_L(\nu)\nu + D_{NL}(\nu)\nu + g(\eta) - \tau) \end{bmatrix} \tag{4.49}$$

The vectors and matrices in (4.49) are specified in Table 4.4

Table 4.4: Overview of vectors and matrices used in (4.49).

Vector/Matrix	Equation
M_{RB}	(4.22)
M_A	(4.34)
$C_{RB}(\nu)$	(4.25)
$C_A(\nu)$	(4.35)
$D_L(\nu)$	(4.36)
$D_{NL}(\nu)$	(4.37)
$g(\eta)$	(4.31)
τ	(4.39)
$J_q(\eta)$	(4.47)

4.7.7 Ocean Currents

Underwater marine crafts, like ROVs will often be exposed to currents, and it is, therefore, useful to include this in the mathematical model. For an irrotational constant ocean current (no angular component), the relative velocity vector is expressed in Equation (4.50)

$$\nu_r = \begin{bmatrix} \nu - \nu_c \\ \omega \end{bmatrix} \tag{4.50}$$

where ν_c is the current velocity vector expressed in the BODY frame. Finding the ocean current expressed in the Body frame, from the NED frame can be done as follows:

$$\nu_c = R(q)^\top \nu_c^n \tag{4.51}$$

or, if the current velocity vector is defined from another frame such as the ENU (East North Up) frame, as it is in the Gazebo simulator.

$$\nu_c = R(q)^\top R_n^e \nu_c^e, \qquad R_n^e = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{4.52}$$

Where ν_c^e is the current velocity vector expressed in the ENU frame and R_n^e is the rotation matrix from ENU to the NED frame.

If C_{RB} is parametrized using a linear velocity-independent parametrization, the equations of motion can be expressed in terms of ν_r, as shown in Equation (4.53) [13, p. 301-302].

$$\begin{aligned} \dot{\eta} &= J_q(\eta)(\nu_r + \nu_c) \\ \dot{\nu}_r &= (M_{RB} + M_A)^{-1}(\tau - C_{RB}(\nu_r)\nu_r - C_A(\nu_r)\nu_r - D_L(\nu_r)\nu_r - D_N L(\nu_r)\nu_r - g(\eta)) \end{aligned} \tag{4.53}$$

By applying the result $\dot{\nu}_c = -S(\omega)\nu_c$ [13, p. 301], it is possible to express the equations of motion with respect to the absolute velocity-vector ν.

$$\begin{aligned} \dot{\eta} &= J_q(\eta)\nu \\ \dot{\nu} &= \begin{bmatrix} -S(\omega)\nu_c \\ 0_{3\times1} \end{bmatrix} \\ &\quad + (M_{RB} + M_A)^{-1}(\tau - C_{RB}(\nu_r)\nu_r - C_A(\nu_r)\nu_r - D_L(\nu_r)\nu_r - D_N L(\nu_r)\nu_r - g(\eta)) \end{aligned} \tag{4.54}$$

4.7.8 Alternative Parametrisation using Euler Angles

The model derived in this chapter is parameterised using quaternions. An alternative parameterisation is to use Euler angles, presented in 4.2. To do so requires modifications to three components of the finalised model (4.49):

- The position and orientation states, η.
- The transformation matrix J_q, that transforms the angular and linear velocities from BODY to NED.
- The generalised restoring force vector, $g(\eta)$.

η is re-defined to $\eta = \begin{bmatrix} p & \Theta \end{bmatrix}^\top = \begin{bmatrix} p & \phi & \theta & \psi \end{bmatrix}^\top$, to include roll, ϕ, pitch, θ and yaw, ψ, instead of the quaternions, $q = \begin{bmatrix} \eta & \epsilon_1 & \epsilon_2 & \epsilon_3 \end{bmatrix}^\top$.

J_q is re-parameterised to J_Θ,

$$J_\Theta = \begin{bmatrix} R(\Theta) & 0_{3\times3} \\ 0_{3\times3} & T(\Theta) \end{bmatrix} \tag{4.55}$$

where $R(\Theta)$ and $T(\Theta)$ are respectively, the linear and angular transformation matrices expressed in terms of Euler angles. $R(\Theta)$ is defined as the Tait-Bryan Z-Y-X parametrization, (4.1). The angular velocity transformation, $T(\Theta)$, is defined in (4.56) [13, p. 29].

$$T(\Theta) = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\ /\ \cos(\theta) & \cos(\phi)\ /\ \cos(\theta) \end{bmatrix} \tag{4.56}$$

The generalised restoring force is calculated by replacing the rotation matrix $R(q)$, with $R(\Theta)$.

$$g(\eta) = -\begin{bmatrix} R^\top(\Theta)\,(f_g + f_n) \\ r_g \times R^\top(\Theta) f_g + r_b \times R^\top(\Theta) f_b \end{bmatrix} \tag{4.57}$$

Where

$$f_g = \begin{bmatrix} 0 & 0 & W \end{bmatrix}^\top,\ f_b = -\begin{bmatrix} 0 & 0 & B \end{bmatrix}^\top$$

Expanding (4.57) and using the same symmetry assumptions as previously described in 4.7.1, yields (4.58)

$$g(\eta) = \begin{bmatrix} (W-B)\sin(\theta) \\ -(W-B)\cos(\theta)\sin(\phi) \\ -(W-B)\cos(\theta)\cos(\phi) \\ z_g W\cos(\theta)\sin(\phi) \\ z_g W\sin(\theta) \\ 0 \end{bmatrix} \tag{4.58}$$

4.8 Discussion

4.8.1 Issues With the Original Model

The initial model that was implemented, was taken from another Master's thesis from Flinders University [14]. However, after encountering problems with stability when implementing the controller in the Gazebo simulation, numerous errors in the model were discovered. Specifically in the rotation matrices that transform the equations of motion from the BODY frame to the NED frame, $R(q)$ and $T(q)$, The Coriolis and Centripetal matrices, $C_{RB}(\nu)$ and $C_A(\nu)$, and the gravitational vector, $g(\eta)$. As a consequence of this, it was deemed necessary to re-work the model.

4.8.2 Parameter Uncertainties

Finding appropriate parameters for the mathematical model of the ROV is a time-consuming process of system identification, and is outside the scope of this project. Therefore, the parameters listed in Table 4.3 were also taken from Wu [14]. These values carry a level of uncertainty since these parameters for the BlueROV2 Heavy were approximated from the parameters of another, similar ROV, the BlueROV, whose parameters were estimated in a Master's thesis by Sandøy [20]. It is therefore reasonable to conclude that the parameters are a considerable element of uncertainty in the mathematical model.

4.8.3 Implementation of Relative Velocity

As previously described, in the Gazebo simulator ocean currents are expressed in the ENU frame. These need to be described in the vehicles' BODY frame. This can be accomplished in two distinct ways. It is possible to describe this transformation in the mathematical model, by introducing additional states. The advantage to this method is that the MPC is able to simulate how a change in orientation will affect the dynamics of the ROV. The disadvantage to this method is that new states have to be introduced, and as a consequence the MPC's calculation time increased, and as such was not deemed feasible. An alternative method is to perform the rotation from the ENU to the BODY frame in the *node*. This has the advantage that one avoids implementing additional states in the model. However, this solution has the distinct disadvantage that the MPC cannot predict how a change in orientation will affect the ROV's dynamics and is instead fed the current velocity vector v_c from the node. The result however did not improve, therefore it was decided not to implement it in the Gazebo simulator when running the standard tests, to avoid unnecessary complexity that can be the source of new errors.

4.9 Conclusion

The mathematical model of the ROV is realised in the form of Fossen‘s robot robot-inspired model for marine craft. The model is an altered version of the one presented by Wu [14], as errors were discovered in this model. The model is parametrized using both unit quaternions and Euler angles. An alternate parametrization of C_{RB} was computed to allow for the equations of motion to be expressed in terms of the relative velocity, which increases the accuracy of the model in environments with irrotational ocean currents (assuming the current is known). An area of significant uncertainty in the model is the parameters used, as the parameters are estimated from the original BlueROV.

Chapter 5

Model Predictive Control

Model Predictive Control, or MPC, is an optimal control technique that uses a model of the system to solve an optimisation problem [21, 22].

Firstly, the theoretical framework of the MPC is presented with the necessary mathematical background for the optimisation problem and constraints. Then the basic structure of the controller will be described, with its key parts, followed by a simplified example of functionality. The theoretical framework is concluded with a description of do-mpc which is a Python package used in the implementation of the MPC. After this, the design of the controller will be presented, describing the cost function and how the FOV problem and collision avoidance problem was solved. The chapter is concluded with a presentation of the results of the design of the MPC, followed by a discussion of results, alternative solutions, and the MPC in general.

5.1 Theoretical Framework

5.1.1 Notation

A table of the symbols used in this chapter:

Table 5.1: Symbol list for MPC chapter

Symbol	Description
J	Cost function
K	Gain constant
x	Dynamic state
u	Controller output
z	Algebraic state in do-mpc
p	Static variable
p_{tv}	Time-varying variable
R	Penalty factor in cost function R-term
l	Lagrange term
m	Mayer term
b	Bounds
ϵ	Slack variable
$\overrightarrow{u}$	Standard vector notation
$\overrightarrow{v}$	Standard vector notation
Δ	Difference between the actual value and desired value
r	Radius
η	Real part in quaternion
$\epsilon_{1,2,3}$	Imaginary part in quaternion
ϕ	Roll
θ	Pitch
ψ	Yaw
α	Angle between directional vector of ROV and vector to other ROV
γ	FOV angle
d	Distance between ROVs

5.1.2 Optimal Control

Optimal control theory is a branch of mathematics developed to find optimal ways to control a dynamic system [23]. MPC utilizes optimal control theory to optimise the control of a constrained system. The objective of an optimal control system is to select appropriate control actions to optimise a cost function, where the cost function symbolises the desired states- and dynamics for the system.

5.1.3 Why MPC?

MPC is a reliable and robust control method. It is an active controller that can predict the behaviour of the system, and take action preventively to stabilise the system. In this system, this can counteract dynamics such as the drift that is present when controlling submerged systems.

The possibility of using constraints in MPC is also a good method of defining undesirable and unfeasible states for the system, as the controller discards or punishes solutions that break the aforementioned constraints, while having no impact on solutions that do not.

MPC is also capable of handling advanced multi-input multi-output (MIMO) non-linear systems such as the case with the ROVs used in this thesis [24].

5.1.4 MPC Structure

The MPC consists of a prediction model and an optimiser. The prediction model is based on the mathematical model of the system. The optimiser contains the cost functions and constraints that describe the desired behaviour for the system [21]. The MPC combines the reference, measured output (current state), and measured disturbances and uses this data to calculate the system input, as shown in Figure 5.1.

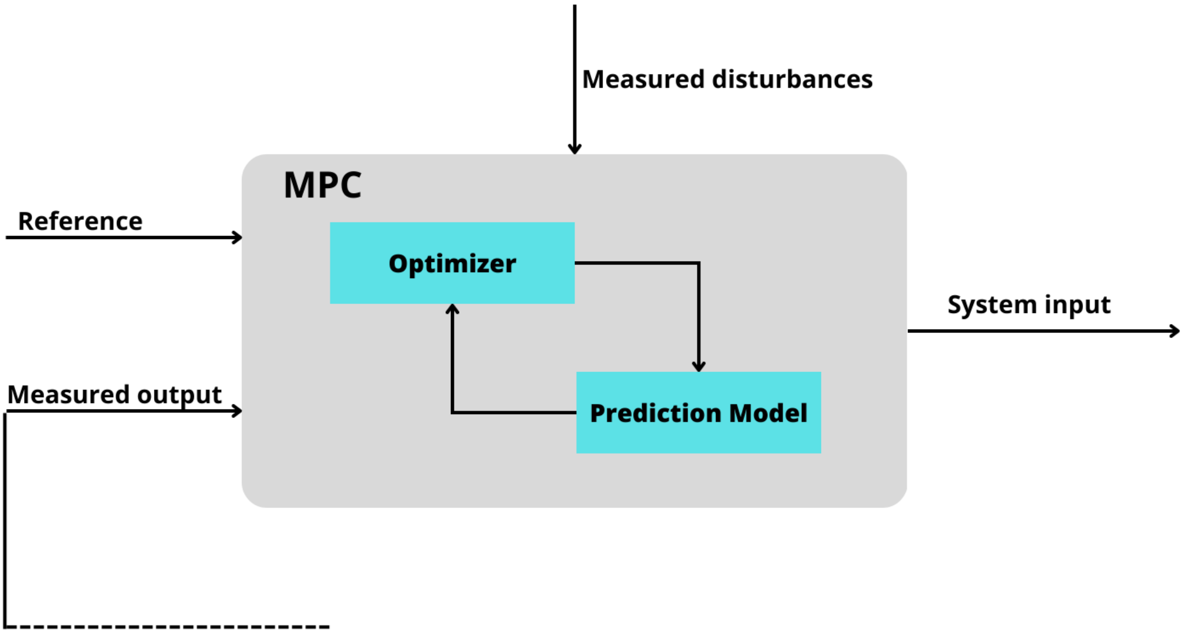


Figure 5.1: MPC Block Diagram

5.1.5 Decentralized Control

A decentralized control system is a system where control decisions are made by multiple independent agents or subsystems rather than a single central controller. In this type of system, each agent or subsystem is responsible for controlling a specific function or process, and communication between agents or subsystems is used to coordinate their actions and achieve the overall goal of the system.

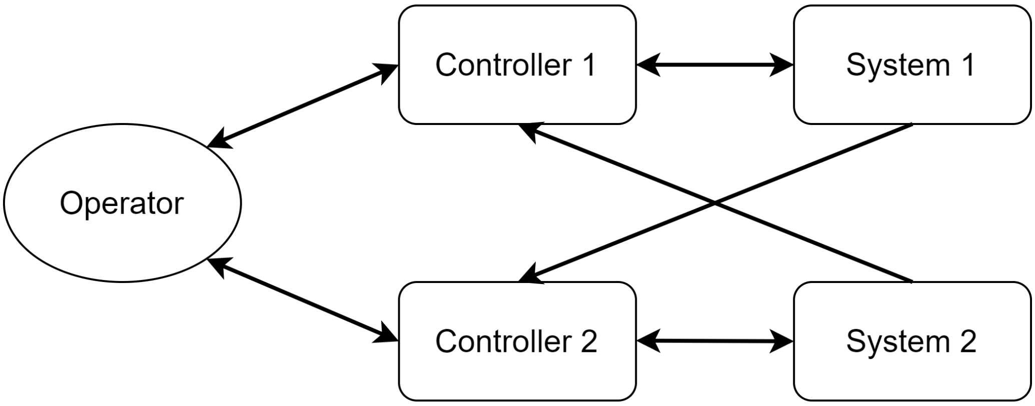


Figure 5.2: Signal flow in decentralized control

Figure 5.2 visualises the signal flow of the implemented system. The operator controls the overall goal of the decentralized system, which is sent to the agents, which in this system are the controllers. The controllers control the associated subsystem and use the states of both subsystems to determine control actions to achieve the overall goal. An example of an overall goal from this thesis is positional control of the ROVs, which is the subsystems while avoiding collisions.

5.1.6 Cost Function

The cost function in the controller is the main definition of the optimisation problem. In model predictive control, the controller's task is to optimise the value of the cost function over time while not violating the constraints.

Generally, when designing the cost function the terms should all be convex. Convex functions have a global minimum point about which they are symmetric. This makes them applicable in cost functions as this characteristic enables the optimisation problem to be solved both reliably and efficiently [25, p. 7-8].

When designing a cost function, the goal is most often to express the difference between the current states and the desired states while penalising unwanted outcomes.

The following equation is an example of what a simple cost function, denoted by J, could look like

$$J(x,u) = K_1(x_d - x)^2 + K_2 \cdot u^2 \tag{5.1}$$

where $(x_d - x)^2$ is a convex expression that creates a cost corresponding to the difference between the desired value of x, referred to as x_d, and the actual value of x. The term u^2 is an example of a convex function that penalises unwanted outcomes as it puts a cost on the controller output diverging from zero. By introducing the strictly positive gain values K_1 and K_2 to this cost function we could modify the impact of the different terms in the cost function.

Exemplifying the Impact of Gains

The following is an illustrative example of a control system that could use the cost function (5.1), and how one could use the gains to change the characteristics of the control system. In the example, x is the amount of liquid in a water tank, and u is the power supplied to the pump that fills the said tank.

1. If $K_1 \gg K_2$, the difference between desired and actual amount of liquid has a big impact on the cost function, and the MPC would use the pump to fulfil $x = x_{desired}$ as fast as possible.
2. If $K_2 \gg K_1$, the power supplied to the pump would cause a big cost, which would likely lead to a slower regression of the difference between x_d and x, but could also reduce the usage of power, and could lead to a more stable regulation.

Depending on the system, both of these scenarios could be the desired control characteristics.

Cost Function in do-mpc

In the do-mpc API reference [26] the cost function found under the function *do_mpc.controller.MPC.set_objective*, and is expressed as

$$J(x,u,z) = \sum_{k=0}^{N} (l(x_k, z_k, u_k, p_k, p_{tv,k}) + \Delta u_k^T R \Delta u_k) + m(x_{N+1}) \tag{5.2}$$

where

- $l(x_k, z_k, u_k, p_k, p_{tv,k})$ is the Lagrangian term and measures the running cost at every time step over the time horizon N.
- $m(x_{N+1})$ is the Mayer term[1] [27], and it represents the desired state at the end of the time horizon N.
- $\Delta u_k^T R \Delta u_k$ is the R-term and it is a factor that penalizes the changes in the output vector of the controller.

5.1.7 Constraints

A constraint on a control system is a way of limiting the dynamics of the system. This usually decreases the efficiency and flexibility of the controller, and should therefore only be used when necessary. One normal usage of constraints is to use a constraint to express the physical limitations of the system. If the output of the controller u controls the power supplied to an electric motor, constraints could be set on u to ensure that the motor is not supplied with power greater or less than it is designed to be supplied with, as this could damage the motor. This type of constraint is referred to as bounds.

Bounds

The following equation is an inequality constraint that exemplifies how bounds could be implemented in the MPC-controller.

$$b_{lb} \leq x, u, z \leq b_{ub} \tag{5.3}$$

where x, u, z is the system variable that is to be bounded, b_{lb} is the lower bound and b_{ub} is the upper bound for the system variable. If (5.3) does not hold for every step in the projected time horizon N, the solution breaks the constraint and is infeasible.

[1] In the do-mpc API reference [26] the m-term is referred to as the *meyer term*. But this appears to be a misprint in the documentation. In this report, it is assumed to be the Mayer term.

Non-linear Constraint

A constraint on the system can also be a more complex and non-linear function, and this is referred to as a non-linear constraint. The non-linear constraint in do-mpc is an inequality constraint that could be configured both as a soft- and hard constraint. Setting the non-linear constraint to be a hard constraint means that any solution from the MPC optimiser that breaks the inequality constraint, is infeasible and will be discarded. Setting the non-linear constraint as a soft constraint introduces slack variables to the cost function, where any violation of the inequality constraint is feasible, but penalised in terms of cost. The cost of violating the soft constraint is determined by multiplying the slack variable with an associated penalty term.

The equation implemented for the hard constraint in do-mpc is:

$$m(x,u,z,p_{tv},p) \leq m_{ub} \tag{5.4}$$

and the equation implemented for the soft constraint is:

$$m(x,u,z,p_{tv},p) - \epsilon \leq m_{ub} \tag{5.5}$$

where $m(x,u,z,p_{tv},p)$ is the non-linear function, m_{ub} is the upper bound and ϵ is the slack variable. These equations could be found in the do-mpc API reference [26] under the function *do_mpc.controller.MPC.set_nl_cons*.

5.1.8 Function

The system input is computed through a series of calculations that predicts future states and the required input to achieve these states. These calculations are done at every time step for a finite time interval called the horizon. In Figure 5.3 one can see the previous states from $t-1$ to t where t is the present time. The dotted line inside the highlighted horizon is predicted states the MPC calculates at time t to achieve the reference. Figure 5.4 shows the required output the MPC calculates for the system to achieve the predicted states within the horizon. After the calculations, the MPC applies the input for the first time-step and disregards the rest [22].

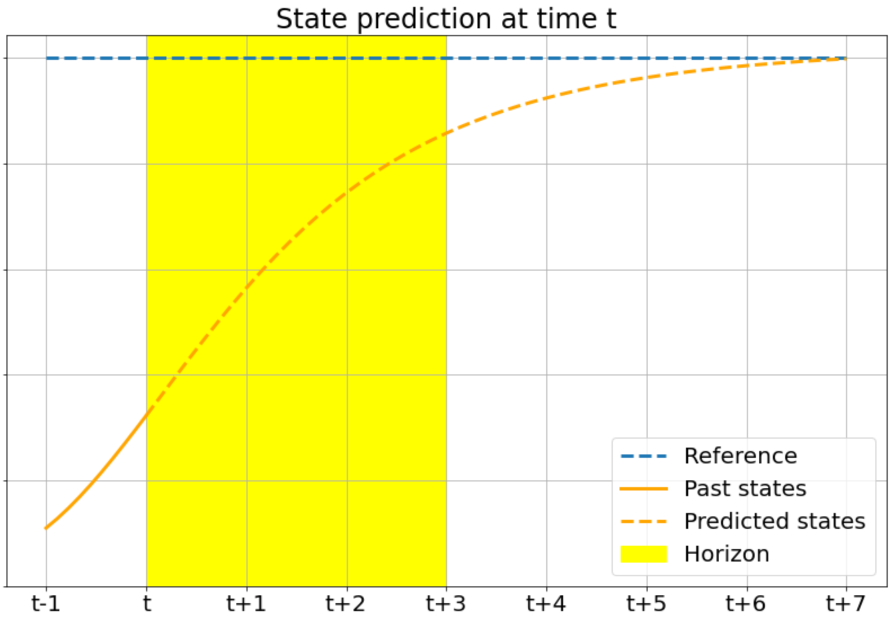


Figure 5.3: MPC State prediction at time t

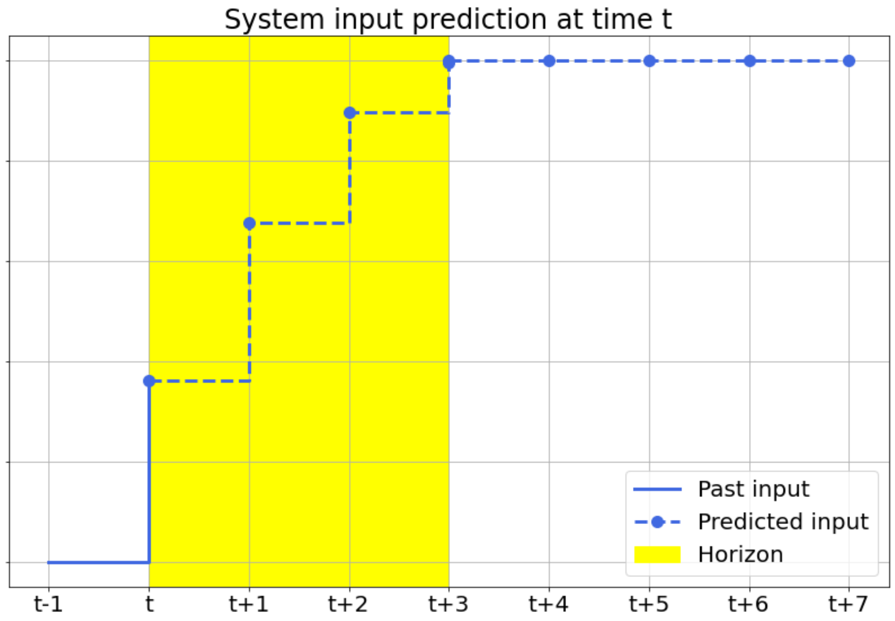


Figure 5.4: MPC system input prediction at time t

The prediction calculation might differ from the actual results. This is because the calculations rely on the accuracy of the prediction model [28, pp. 414–415]. In Figure 5.5 the result at $t+1$ is slightly lower than the prediction. At the time $t+1$ the MPC repeats the calculations for the new state predictions as seen in Figure 5.5 and the new output as seen in Figure 5.6 [22].

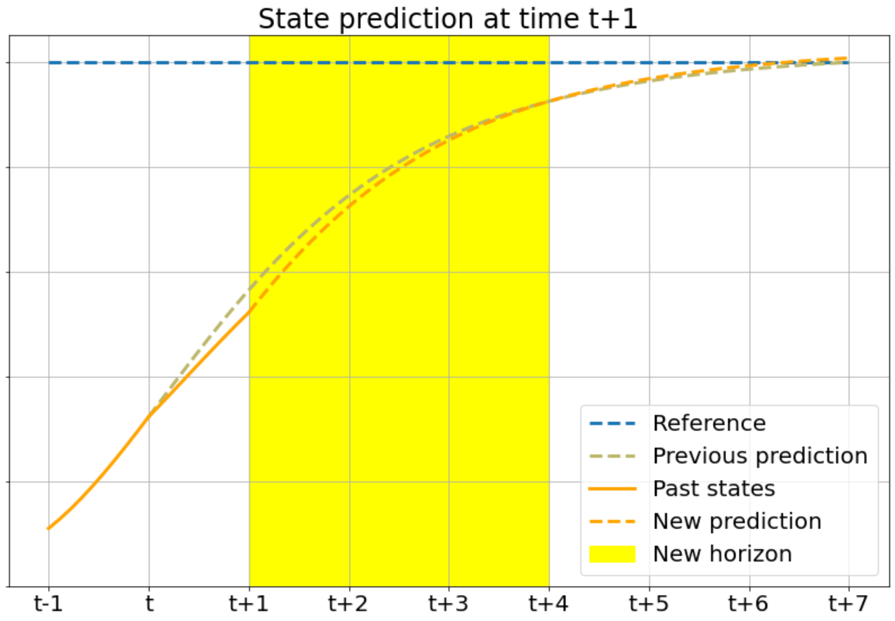


Figure 5.5: MPC State prediction at time t+1

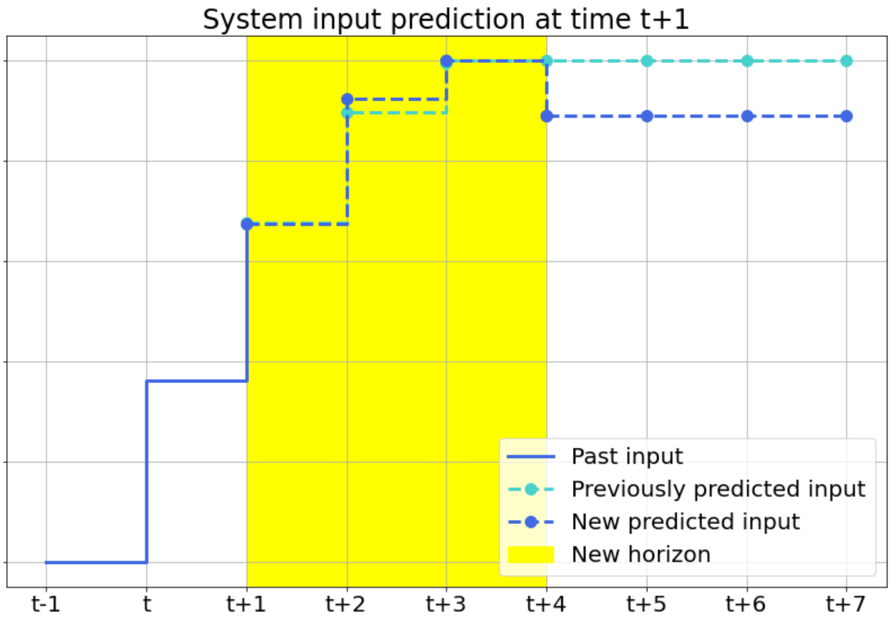


Figure 5.6: MPC system input prediction at time t+1

5.1.9 Modelling Optical Communication

As the ROVs in the research project was planned to communicate through wireless optical communication, an objective for the MPC was that it should ensure line of sight.

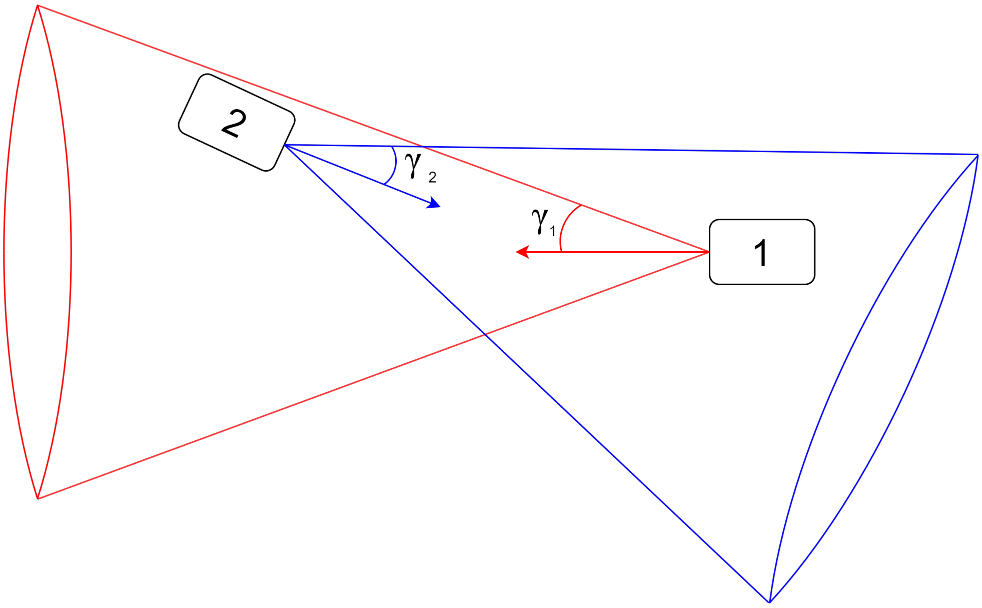


Figure 5.7: Optical communication concept

The line of sight is modelled as a cone in the surge direction in the body frame of the vehicles. The concept is visualised in Figure 5.7, where the two objects represents the ROVs, while γ_1 and γ_2 represents the maximal angle of the cone. If both agents are localised in each other's cones, line of sight is ensured.

5.1.10 Do-mpc

Do-mpc is explained in their webpage as "*..a comprehensive open-source toolbox for robust model predictive control (MPC) and moving horizon estimation (MHE).*" [26, 29]. This free open-source toolbox is a Python library made at the Technical University of Dortmund (TU Dortmund), used to design control systems. It contains a flexible and modular structure for simulation, estimation and control, which in this thesis do-mpc was used in the implementation of the Model Predictive Control (MPC)-controller, and the Python simulator.

By using do-mpc in this thesis there was no need for the design of numerical solvers and optimisers in the thesis, which would have been time-consuming. Instead, more time could be spent implementing the system model and designing the control characteristics and functionality in the controller and the peripheral system.

5.2 Design

For the MPC-controller designed in this thesis, the system model and the controller were split into two modules.

- The model module implemented the system model described in Chapter 4 in a do-mpc format and initialised the system's parameters and variables.
- The controller module contains the cost function, constraints and controller parameters, the controller also implements the system model, which is used as the prediction model for the controller.

This section will focus on the design and implementation of the controller module, as the model is described in Chapter 4, and the parameters and variables can be found in the code on GitHub [3].

5.2.1 Cost Function

The cost function was designed with two main tasks in mind. Positional control, meaning the xyz-coordinates and attitude control, meaning the quaternion-coordinates.

Positioning

In this thesis, two approaches for positional control are proposed, *circular setpoints* and *squared error*.

Circular Setpoint positioning splits the x-, y- and z-positioning into two problems in the cost function. Firstly there is the positioning in the z-direction (the depth), secondly, there is the positioning in the xy-plane. In this solution, when the ROVs are at their desired position they are all at the same depth and a set distance d from the desired point.

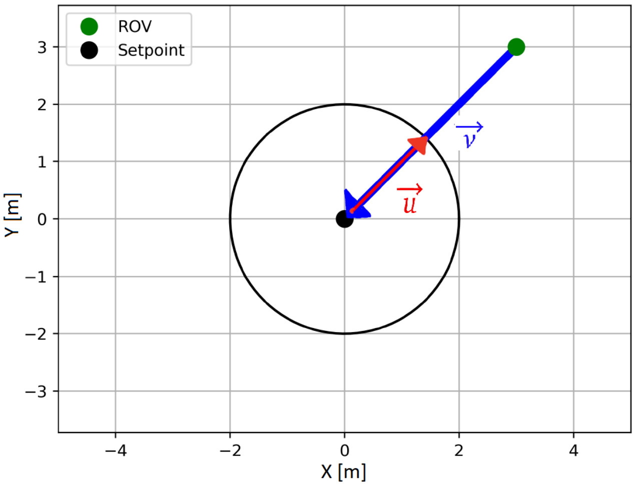


Figure 5.8: Positioning in the xy-plane

If one draws a vector of a set length in every direction from the desired position in the xy-plane the terminal points would make a circle, as seen in 5.8, where $\left\|\overrightarrow{u}\right\|$ is the distance r.
The ROV has reached its desired position when (5.6) is fulfilled.

$$\left\|\overrightarrow{v}\right\| = \left\|\overrightarrow{u}\right\| \wedge z = z_d \tag{5.6}$$

do-mpc would not work when using roots in the expressions of the cost function or constraints. This meant that J_1 in (5.7) could not be implemented, and the alternative form J_2 had to be used.

$$\begin{aligned} J_1 =& (\sqrt{K_x \Delta x^2 + K_y \Delta y^2} - r)^2 + K_z \Delta z^2 \\ J_2 =& ((K_x \Delta x^2 + K_y \Delta y^2) - r^2)^2 + K_z \Delta z^2 \end{aligned} \tag{5.7}$$

Squared Error Squared error positioning is implemented strictly using terms with the layout: $K\Delta x^2$. This is the simplest form of expressing the error between two variables in a function that is convex and quadratic.

In squared error the following Equation was implemented in the cost function:

$$J = K_x(x - x_d)^2 + K_y(y - y_d)^2 + K_z(z - z_d)^2 \tag{5.8}$$

Attitude

Since there were two system models used, one using Euler angles and one using quaternions, it was necessary to make attitude control solutions to match both of these. Which form of attitude control to use was decided by the system model used.

Quaternion Attitude Control Quaternion attitude control is briefly described in section 4.3.5, and is fully derived in appendix B.

The complete formula for attitude control with quaternions is given as:

$$\begin{aligned} J =& K_A(((\eta_d \eta + \epsilon_{1d}\epsilon_1 + \epsilon_{2d}\epsilon_2 + \epsilon_{3d}\epsilon_3)^2 - 1)^2 + (-\epsilon_{1d}\eta + \eta_d\epsilon_1 - \epsilon_{3d}\epsilon_2 + \epsilon_{2d}\epsilon_3)^2 \\ &+ (-\epsilon_{2d}\eta + \epsilon_{3d}\epsilon_1 + \eta_d\epsilon_2 - \epsilon_{1d}\epsilon_3)^2 + (-\epsilon_{3d}\eta - \epsilon_{2d}\epsilon_1 + \epsilon_{1d}\epsilon_2 + \eta_d\epsilon_3)^2)^2 \end{aligned} \tag{5.9}$$

where the variables with lower index d expresses the *desired* quaternion coordinates.

Euler Attitude Control Attitude control is achieved using the same method and layout as described in section 5.2.1.

The formula for attitude control using Euler angles is given in the symbolic form:

$$J = K_{\phi}(\Delta\phi)^2 + K_{\theta}(\Delta\theta)^2 + K_{\psi}(\Delta\psi)^2 \tag{5.10}$$

with ϕ, θ and ψ being the roll, pitch and yaw coordinates.

5.2.2 Constraints

FOV constraint

To ensure that the Field of view (FOV) criteria are fulfilled by the MPC, a non-linear constraint was designed. It was made with the assumption that the ROVs had access to the positional x-, y- and z-coordinates of the other ROVs.

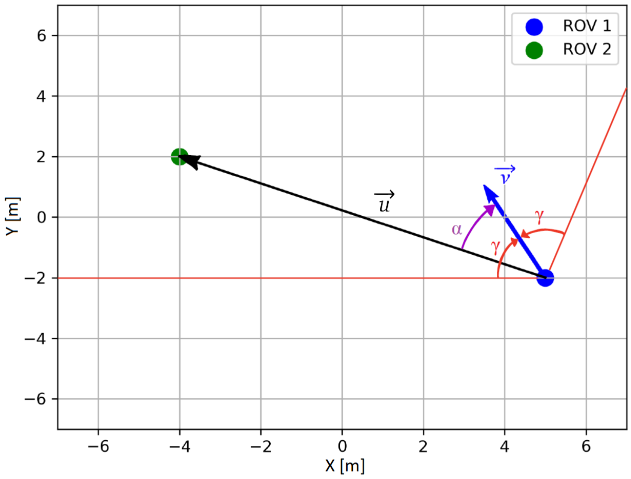


Figure 5.9: Field of view (FOV) concept

The concept of the solution is visualised in two dimensions, as it directly translates into three dimensions. The concept and the symbolic formula are the same in both cases, with the only difference being that in two dimensions the formula calculates the angle between the two-dimensional vectors $\vec{u}$ and $\vec{v}$ in the plane. While in three dimensions calculating the angle between the same vectors with a 3rd-dimensional component, and in space.

In Figure 5.9 the FOV criteria is fulfilled if the inequality constraint in the following Equation holds

$$\alpha \leq \gamma \tag{5.11}$$

The angle α is the angle between $\vec{v}$ and $\vec{u}$, and γ is the maximum FOV angle. In the figure 5.9 the γ angle is 60°.

The vector between the two ROVs and the directional vector of ROV 1 is given by

$$\vec{v} = \begin{bmatrix} 1 - 2(\epsilon_2^2 + \epsilon_3^2) \\ 2(\epsilon_1\epsilon_2 + \epsilon_3\eta) \\ 2(\epsilon_1\epsilon_3 - \epsilon_2\eta) \end{bmatrix}, \vec{u} = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{bmatrix} \tag{5.12}$$

The following formula was used as the basis to calculate the angle between two vectors

$$\cos\alpha = \frac{\overrightarrow{v} \cdot \overrightarrow{u}}{\left\|\overrightarrow{v}\right\| \cdot \left\|\overrightarrow{u}\right\|} \tag{5.13}$$

Rewriting the formula gives

$$\gamma \leq \arccos\left(\frac{\overrightarrow{v} \cdot \overrightarrow{u}}{\left\|\overrightarrow{v}\right\| \cdot \left\|\overrightarrow{u}\right\|}\right), \left\|\overrightarrow{v}\right\| \triangleq 1 \tag{5.14}$$

where Equation (5.14) is the result of rewriting (5.13) and inserting it in to (5.11).
By rewriting (5.14), the final symbolic equation for the non-linear constraint is

$$\cos\gamma \cdot \left\|\overrightarrow{u}\right\| - \overrightarrow{v} \cdot \overrightarrow{u} \leq 0 \tag{5.15}$$

The complete derivation of the equations used in the non-linear constraint used for the FOV constraint can be found in appendix A.

The complete formula of the implemented non-linear expression in the non-linear constraint is given in do-mpc format as

$$\begin{aligned} m(x,u,z,p_{tv},p) = & \cos(\gamma) \cdot d \\ & -((1-2\epsilon_2^2+2\epsilon_3^2)(x_2-x_1) \\ & +(2\epsilon_1\epsilon_2+2\epsilon_3\eta)(y_2-y_1) \\ & +(2\epsilon_1\epsilon_3-2\epsilon_2\eta)(z_2-z_1)) \end{aligned} \tag{5.16}$$

The non-linear constraint was implemented both as a soft constraint and a hard constraint, with the γ-angle set for the soft constraint being smaller than the γ-angle for the hard constraint.

This gave the possibility of introducing steps to the penalty:

Table 5.2: FOV penalty steps

Step	Angle range	Severity	Description	Cost
1	$\alpha < \gamma_S$	Ideal	Optimal range	Low. Only from attitude control 5.2.1.
2	$\gamma_S \leq \alpha < \gamma_H$	Warning	Feasible range	High. Cost from the soft constraint
3	$\gamma_H \leq \alpha$	Infeasible	Infeasible range	Infeasible

Table 5.2 lists the transitions between steps and associated costs. Notably, lower steps' costs persist as the severity increases. This design choice proves to be advantageous in this system as it allows for the creation of transition points that greatly increase the priority of the FOV constraints.
As long as α falls within the ideal range, its specific value holds limited significance, as the optical communication function remains consistent throughout. This is therefore represented by a low penalty.

When reaching the step with the severity *warning*, it is deemed necessary to take action to ensure that α is controlled back to the ideal range. By increasing the cost drastically, the FOV-angle task will have a higher priority and impact on the control.

When reaching the step with severity *infeasible*, then that would mean the hard constraint has been violated and there is no longer communication between the ROVs.

Distancing Between the ROVs

To ensure that the ROVs avoid collisions, a non-linear constraint was designed. As with the FOV constraint, the collision avoidance constraint also was designed with the assumption that any ROV in the multi-agent system had access to the x-, y- and z-coordinates of all the other ROVs.

The constraint that ensured the distance between the ROVs was designed on the basis of

$$\left\| \overrightarrow{v_d} \right\| \geq d \tag{5.17}$$

If the length $\overrightarrow{v_d}$ (the vector between two ROVs) is longer than d (a distance set by the operator), the constraint gave no cost, but if $\left\| \overrightarrow{v_d} \right\| < d$, this gave a high cost.

$$m(x,u,z,p_{tv},p) = d^2 - ((x_2 - x)^2 + (y_2 - y)^2 + (z_2 - z)^2) \tag{5.18}$$

Equation (5.18) is the complete formula of the implemented non-linear expression in the non-linear constraint.

As with the FOV constraint, the distancing constraint also was implemented both as a hard constraint and as a soft constraint. With the soft constraint being d and the hard constraint being $d_{10\%}$ (10% of the value d).

This introduced step for the distancing constraint:

Table 5.3: Distancing penalty steps

Step	Distance range	Severity	Description	Cost
1	$\left\| \overrightarrow{v_d} \right\| \geq d$	Ideal	Optimal range	None.
2	$d > \left\| \overrightarrow{v_d} \right\| \geq d_{10\%}$	Critical	Unsafe range	High. Cost from the soft constraint
2	$d_{10\%} > \left\| \overrightarrow{v_d} \right\| \geq d_{10\%}$	Infeasible	Infeasible range	Infeasible

The states of the distancing constraint are described in Table 5.3. When in the step described with the severity *ideal* the distance between the ROVs is safe, and there is no penalty given.

When reaching the step of severity *critical* there is no collision, but the distance between the ROVs is deemed unsafe, and this introduces a high cost to the controller which makes getting the distance back to the optimal range a priority.

When reaching the severity *infeasible* a collision is probable, which could cause material damage on the ROVs.

System Parameters in the Controller

The do-mpc controller has several parameters that can be adjusted. The complete list of system parameters can be found in the API reference for do-mpc [26] under the function *do_mpc.controller.MPC.set_param*. The parameters used in the controller implemented in this thesis are:

- **n_horizon** - *Prediction horizon of the optimal control problem*
- **t_step** - *Timestep of the mpc*
- **n_robust** - *Robust horizon for robust scenario-tree MPC*
- **nlpsol_opts** - *Dictionary with options for the CasADi solver*
    - **ipopt.max_iter** - *Maximum number of SQP iterations*[2]

5.3 Results

5.3.1 Cost Function

The resulting Lagrange- and Mayer-term of the resulting cost function will be displayed in this section.

The R-term is not displayed, as it only contains a gain R, that increases or decreases the cost of altering the value of the controller output.

Positional Term in the Lagrange-term

$$\begin{aligned}
l_{P1}(x_k, z_k, u_k, p_k, p_{tv,k}) =& ((K_x(x_d - x)^2 + K_y(y_d - y)^2) - r^2)^2 + K_z(z_d - z)^2 \\
l_{P2}(x_k, z_k, u_k, p_k, p_{tv,k}) =& (\sqrt{K_x(x_d - x)^2 + K_y(y_d - y)^2} - r)^2 + K_z(z_d - z)^2 \\
l_{P3}(x_k, z_k, u_k, p_k, p_{tv,k}) =& K_x(x - x_d)^2 + K_y(y - y_d)^2 + K_z(z - z_d)^2
\end{aligned} \tag{5.19}$$

l_{P1}: Using the implemented version of circular setpoint
l_{P2}: Using the optimal version of circular setpoint[3]
l_{P3}: Using squared error

Attitude term in the Lagrange-term

$$\begin{aligned}
l_{A1}(x_k, z_k, u_k, p_k, p_{tv,k}) =& K_A(((\eta_d\eta + \epsilon_{1d}\epsilon_1 + \epsilon_{2d}\epsilon_2 + \epsilon_{3d}\epsilon_3)^2 - 1)^2 \\
& + (-\epsilon_{1d}\eta + \eta_d\epsilon_1 - \epsilon_{3d}\epsilon_2 + \epsilon_{2d}\epsilon_3)^2 \\
& + (-\epsilon_{2d}\eta + \epsilon_{3d}\epsilon_1 + \eta_d\epsilon_2 - \epsilon_{1d}\epsilon_3)^2 \\
& + (-\epsilon_{3d}\eta - \epsilon_{2d}\epsilon_1 + \epsilon_{1d}\epsilon_2 + \eta_d\epsilon_3)^2) \\
l_{A2}(x_k, z_k, u_k, p_k, p_{tv,k}) =& K_\phi(\phi_d - \phi)^2 + K_\theta(\theta_d - \theta)^2 + K_\psi(\psi_d - \psi)^2
\end{aligned} \tag{5.20}$$

l_{A1}: Using quaternions
l_{A2}: Using euler-angles

[2] This is a parameter set for CasADi solver [30], which is used as a solver in the do-mpc toolbox

[3] As described in section 5.2.1, roots was unusable in do-mpc. This solution could therefore never be tested.

Full Lagrage-term

$$l(x_k, z_k, u_k, p_k, p_{tv,k}) = l_P(x_k, z_k, u_k, p_k, p_{tv,k}) + l_A(x_k, z_k, u_k, p_k, p_{tv,k}) \tag{5.21}$$

The full cost function added a positional term and an attitude term, as shown in (5.21).

$$\begin{aligned} m(x_{N+1}) =& l(x_{N+1}, z_{N+1}, u_{N+1}, p_{N+1}, p_{tv,N+1}) \\ &+ K_u(u_{1,N+1}^2 + u_{2,N+1}^2 + u_{3,N+1}^2 + u_{4,N+1}^2 + u_{5,N+1}^2 + u_{6,N+1}^2 + u_{7,N+1}^2 + u_{8,N+1}^2) \end{aligned} \tag{5.22}$$

Equation (5.22) is the Mayer-term of the cost function implemented in the final cost function.

Comparison of Positional Control Approaches

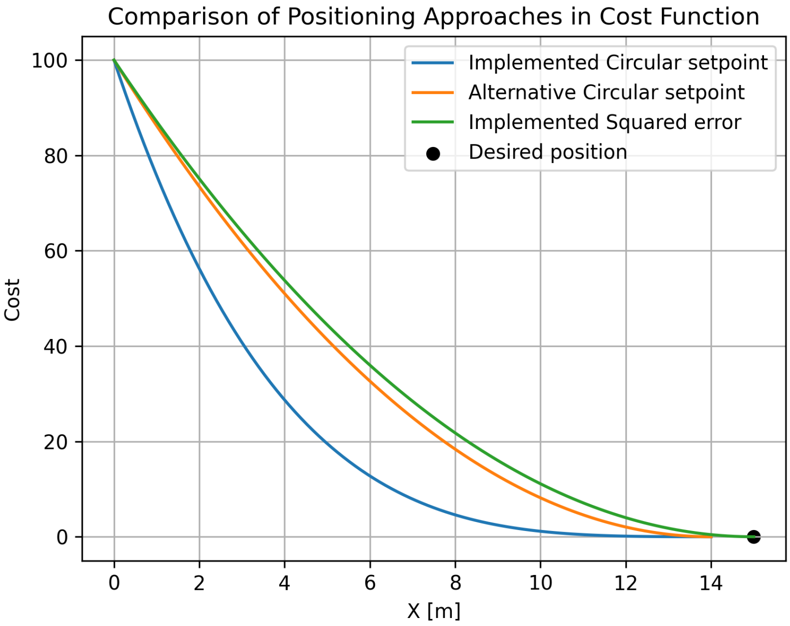


Figure 5.10: Cost. Comparison of circle setpoint and squared error

In Figure 5.10 the cost of the different configurations of the positional control is compared.

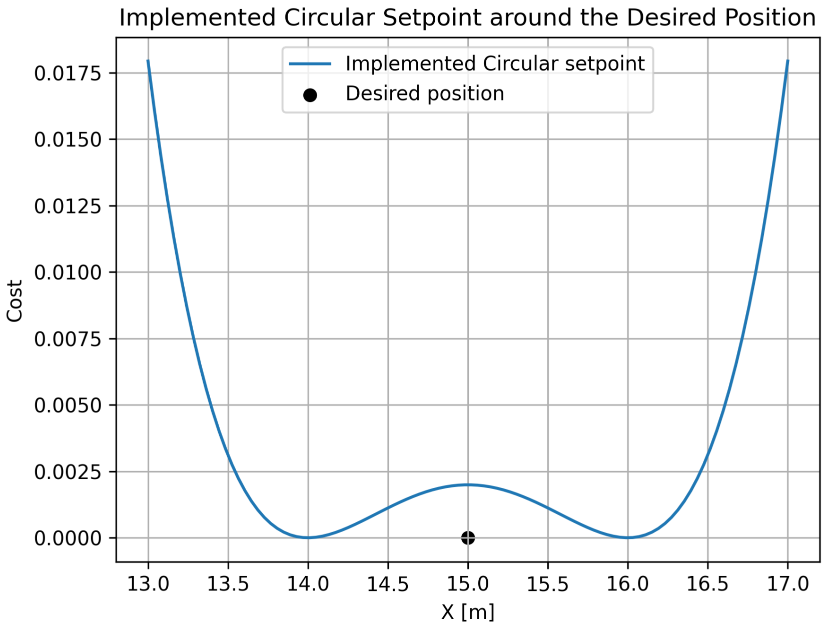


Figure 5.11: Cost. Implemented Circular Setpoint Around the Desired Position

Figure 5.11 gives a closer look at the cost of the implemented circular setpoint solution around the desired position.

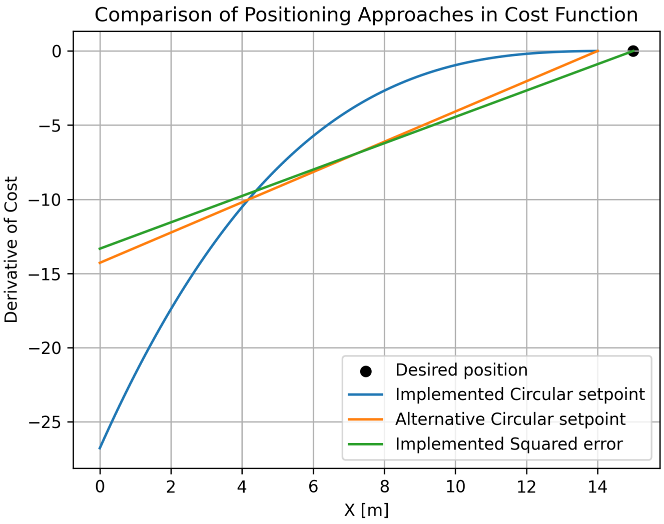


Figure 5.12: Derivative of cost. Comparison of circle setpoint and squared error

In Figure 5.12 the derivative of the cost in the different configurations is compared.

There are some important things to note when analysing these figures:

1. The circular setpoint and the squared error function have the same desired position in Figure 5.10, 5.12 and 5.13, but the final position is different. This is because of the fact that by using the circular setpoint solution, the ROV has a final position at a distance r from the desired position. In this example $r = 1$m.
2. The gain in the three functions for the configurations is adjusted to give them the same starting point in x(0). This is done to make them comparable.

Observe that the derivative of *implemented circular setpoint* is declining exponentially as it approaches the terminal point, while the *alternative circular setpoint* and *implemented squared error* are declining linearly.

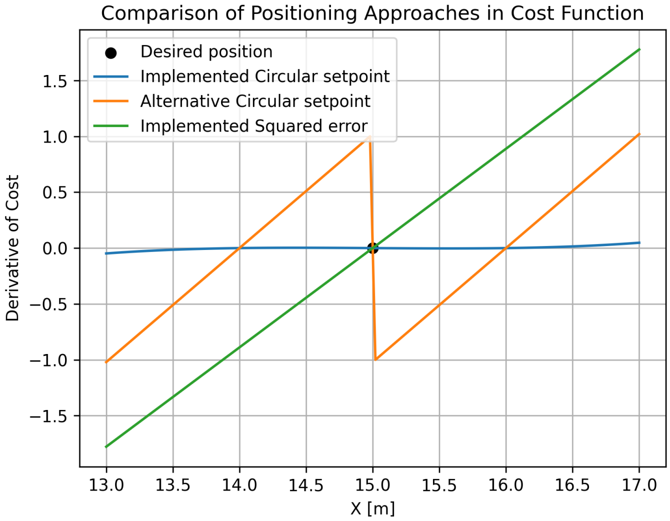


Figure 5.13: Derivative of cost. Discontinuity in comparison of circle setpoint and squared error

However, using *optimal circular setpoint* introduces a discontinuity in the derivative at the desired point, as shown in Figure 5.13.

5.4 Discussion

5.4.1 Cost Function

The circular setpoint solution gave some more predictability in regard to end positions in the multi-agent system compared to the squared error solution, which all had the same desired point as the endpoint, but could not all be placed there because of the distancing necessary between the ROVs. When using the circular setpoint solution with r and d being set adequately, all ROVs are at the same distance from the desired position, which as shown in Figure 5.8 creates a circle around the point. This characteristic can also be used when performing cooperative tasks. By decreasing the value of r and d while all the ROVs are positioned at the circle, they will all approach the desired point from different directions. This could be used if the ROVs had grippers, and were to pick up a fishing net at the ocean floor, and the positioning would be dynamic even when scaling up the system and using several agents.

The circular setpoint solution had some fundamental problems, as the cost function was quartic (fourth order). Namely:

1. A quartic cost function leads to the cost rapidly increasing when the distance between the ROV and the desired point is growing, as seen in Figure 5.10. This could make the system more unstable, as the cost from the positional error would dominate the cost function.
2. The quartic function implemented is not convex, which is a desired characteristic in cost functions. The function has two minimum points, both a distance r from the desired position, as seen in Figure 5.11.
3. The derivative of the function is exponential. This results in a problem where it is harder for the optimiser to find the global minimum, since the derivative is close to zero in the area

around the desired position. This can be seen in Figure 5.12.

This circular setpoint solution was tested in simulations and the result from the testing is described in Chapter 7.

An alternative solution to the implemented circular setpoint solution is also proposed, which uses roots to give the cost a similar shape as the squared error solution, as seen in Figure 5.10. This, for the most part, handled problems 1 and 3 from the implemented circular setpoint solution, which could lead to better control. The alternative solution came with its own set of issues:

1. The issue of non-convexity still persists as problem 2 for the implemented solution. The function was not convex, as it had 2 minimum points, both a distance r from the desired point.
2. The expression introduced a discontinuity in the derivative of the function at the desired point, as shown in Figure 5.13. Discontinuities in cost functions increase the complexity of the optimisation problem.

As do-mpc would not run with the alternative solution for the circular setpoint, it could not be tested in this thesis.

5.4.2 Non-linear Constraints

The FOV-constraint is a dynamic solution that was designed to ensure line of sight between the ROVs. The angle limits for both the soft constraint and hard constraint could be set in the parameters, and the only requirement was the positional states of the other ROV, which could be transferred through optical data transfer as long as the constraint was being held. Using soft constraints to create a range where the FOV angle was prioritized also leads to some robustness, as the ROVs work to keep the angle in a range where the angle is far less than the limit to maintain communication.

A problem with the current implementation of the FOV constraint was that when more than two ROVs were being run, the amount of FOV constraints grew exponentially, since the system needed access to the positional states of all the other ROVs to ensure that the distancing constraints was being held.

This could be solved by introducing functionality for data transfer where the ROVs ensured global access to positional data. Such functionality could have been explored with methods like consensus [31], but as this thesis focused on the two-agent case, it was beyond the scope of the thesis.

The distancing constraint was first implemented in the cost function. This worked, but led to a more complex cost function, which could increase the calculation time for the optimiser, and gave a cost when the distance between the ROVs was larger than what was dangerous. This was unnecessary, as the distance between the ROVs was considered to be satisfactory as long as there was some distance between them. The choice to set the distancing as a constraint meant that there was no cost associated with the distancing as long as it was beyond the set distance d.

A sub-optimal design choice in the distancing constraint was that the hard constraint was set to be 10% of the distance d, which is not a good solution when the value d is small. If d was set to 1m, the hard constraint would be set to 10 cm, and as the physical size of the ROVs is at its longest about 25 cm from the centre, a distance of 10 cm between the ROVs would be impossible, as they would have crashed at about 50 cm. The value for the hard constraint could have been implemented as its own parameter in the controller.

5.4.3 Disadvantages and Alternative Methods of Control

A problem with using MPC is that it is a computationally expensive controller, which requires hardware that is able to handle the multitude of calculations that happens at each time step. This is a key issue, as the BlueROV2 when operating without tethers, use a single board computer as the control unit due to space limitation. This could make the controller unusable for the BlueROV2.

Another potential problem with the use of MPC is that the performance of the controller is dependent on having an accurate prediction model which is dependent on having a good model of the system. If the model of the system is a poor representation, the performance of the controller will be poor.

Alternatively, other methods of control could be explored. One example that was discussed at an early stage in the thesis was to implement a cascade control system where the main control is done by a MPC with a cycle time while the direct control of the thrusters is done with continuous hardware implemented PID-controllers. This solution could handle the problem with hardware limitations as the controller could have a slower cycle time but could lead to a more complex control system, and a less reliable and robust system.

Another alternative to using MPC, is to use LQR, which generally is less computationally expensive. This could have been an advantage with the hardware limitations on the BlueROV2. LQR does not enable the use of constraints on the system, but this could be solved by implementing the distancing constraint and the FOV constraint in the cost function with a high gain value. The problem with using LQR is that the controller is primarily designed to control linear systems, and the model for the BlueROV2 is non-linear, which would affect performance.

5.4.4 Do-mpc

The decision to utilise do-mpc in this thesis gave a head start in the development of the controller as the functionality surrounding the optimiser already was implemented, and ready to be used. There was not any need to design solvers or estimators, which likely would have been time-consuming. The do-mpc package had examples, an API reference, and supported literature that could be used as inspiration in design.

Do-mpc also came with a set of problems. The calculation time of the controller was slow and this reduced the efficiency of the system, which probably will lead to performance issues when implemented on the physical ROV. Another problem working with the do-mpc toolbox was that the inner workings of the solver, and the way it solved complex problems were not always apparent. For example, it was not always clear-cut how it handled broken hard constraints, which lead to a lot of interpretation of results and effects during design and implementation. If the controller were designed without using such a high-level toolbox, the functionality could be exploited more efficiently.

An alternative to do-mpc could have been to explore other options for MPC toolboxes, which could have led to better performance. An example is the MPC toolbox in Matlab, which also has the possibility to convert the code to a C++ format. This likely would have led to a higher performance of the controller nodes.

5.4.5 Model Implementation in do-mpc

Implementing the model in do-mpc was first attempted using matrix operations in numpy, but in the end this was not feasible, since the states and other variables, were implemented as casADI symbols. This caused issues, particularly in the D_{NL} matrix, as it was necessary to use a casADI function inside the numpy matrix, which caused an error. Therefore, for uniformity, the matrices were written out row by row.

The equations of motion were also implemented as differential algebraic equations (DAEs), instead of or ordinary differential equations (ODEs). This was done because in order to express the equations of motion on ODE-form, it is necessary to invert $(M_{RB} + M_A)$. Inverting these matrices results in a "messy" expression, and it is recommended in the do-mpc documentation to utilise DAEs instead [29].

Chapter 6

The Robot Operating System (ROS)

ROS is an open-source software development kit with tools and libraries, designed for use in robot development. ROS is a so-called meta-operating system, which uses concepts typically found in operating systems such as package management and message handling [32].

This chapter describes the implementation of the MPC, presented in Chapter 5, in ROS 2 and how it is configured to interface with the Gazebo simulator. First, an introduction to the theoretical framework is needed to understand the workings of ROS and how it will be used. Then, the mathematical framework for trajectory planning is explained. Afterwards, the implementation's interface with Gazebo is described. The results are then presented, including the package structure and the system topology. Lastly, certain aspects of the implementation are discussed.

6.1 Software

The following table is a list of software that was used in this project.

Table 6.1: Table of Software

Name	Description	Documentation
ROS 2 Humble	Recommend Package for Ubuntu 22.04	ROS2 Documentation[33]
Ubuntu 22.04.2 LTS (Jammy Jellyfish)	Operative System	Ubuntu Image[34]
Gazebo Garden	Application used for Simulation	Gazebo Documentation [35]
Python 10.0 (or newer)	Programming language (Introduces match case)	Python Documents [36]
do-mpc	Python tool	Installation documentation [37]
C++	Programming language	Windows documentation [38]

6.2 Theoretical Framework

6.2.1 A Look Into the ROS 2 Structure

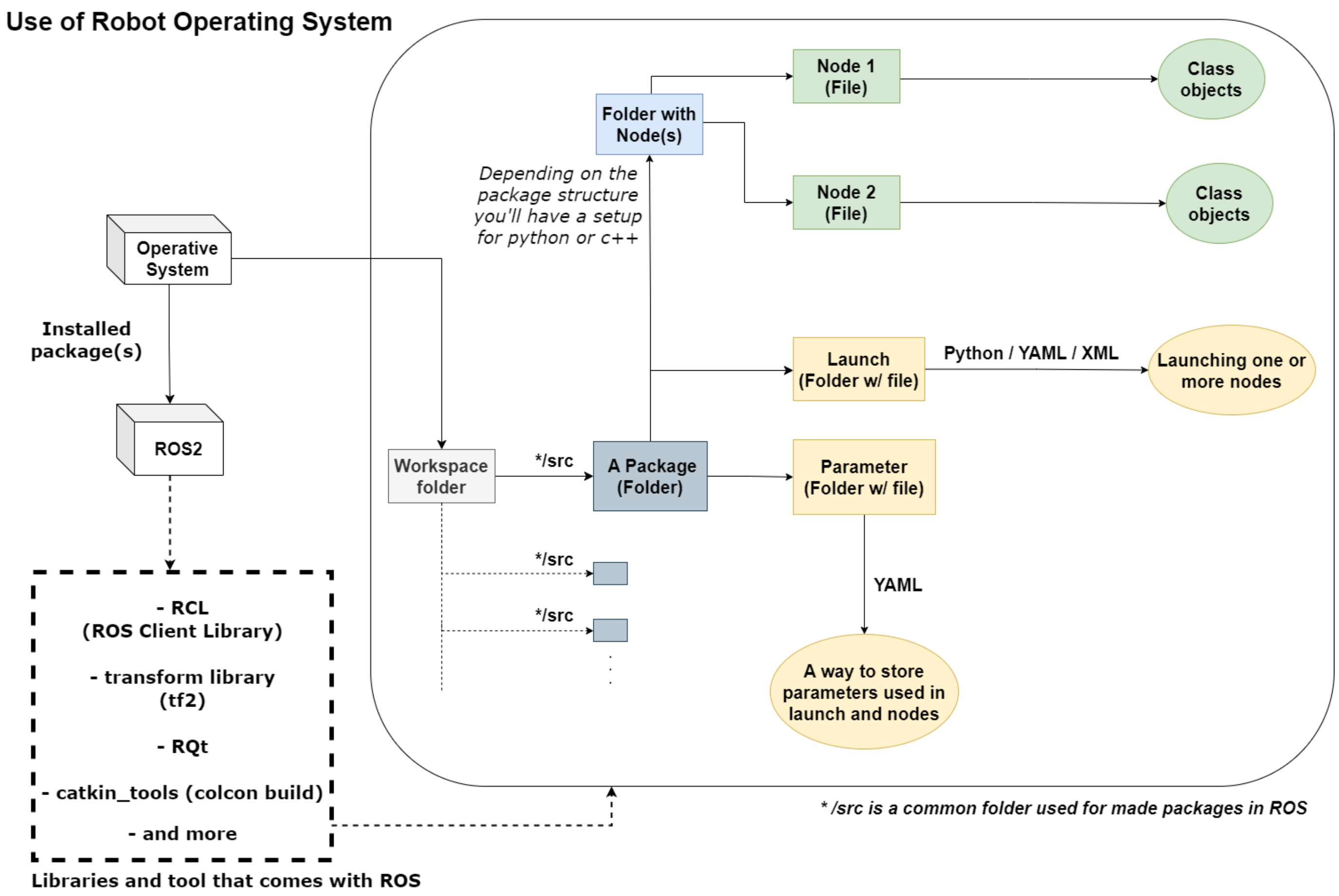

Use of Robot Operating System
Operative System
Installed package(s)
ROS2
- RCL (ROS Client Library)
- transform library (tf2)
- RQt
- catkin_tools (colcon build)
- and more
Libraries and tool that comes with ROS
Workspace folder
*/src
A Package (Folder)
*/src
*/src
Folder with Node(s)
Depending on the package structure you'll have a setup for python or c++
Node 1 (File)
Class objects
Node 2 (File)
Class objects
Launch (Folder w/ file)
Python / YAML / XML
Launching one or more nodes
Parameter (Folder w/ file)
YAML
A way to store parameters used in launch and nodes
* /src is a common folder used for made packages in ROS

Figure 6.1: A simple representation of how ROS filesystem is set up.

Workspace

The recommended way to organise all ROS 2 related work is to use a folder as a repository to work from, and this folder is what would be called the workspace [33]. In Linux, one usually has to build their workspace with *colcon build* and source it before use. Usually what one would see inside a workspace, is a source folder (/src) where *packages* are stored, and the folders *build*, *install* and *log* that gets generated by running *colcon build*.

Colcon, an acronym for "collective construction", is an important tool in ROS 2, which sets up the environment for the packages to be built, tested and used in, amongst other things. More information can be found in the documentation [39, 40].

Package

A package contains the source code and the build around it to implement its functionality. Packages allow users to pack their source code neatly and can be easily distributed amongst others on different platforms for use and further development [41].

When creating a new package, ROS 2 offers a command to build a new package using the ament tools, which can be used to build packages with files required when using Python or C++.

For Python:

- *setup.py* - Necessary for the user to tell ROS 2 how it should install the package.
- *setup.cfg* - Tells ROS 2 where it can find the executable that exists in the package
- *source folder* - has the same name as the package name and is for storing nodes.
- *package.xml* - Contains important information about the package and what libraries and tools the package is dependent on.

For C++:

- *CMakeLists.txt* - Necessary for the user to tell ROS 2 how the code within the package should be built
- *package.xml* - Contains important information about the package and which libraries and tools the package is dependent on.
- *source folder* - Is named *src/*

ROS 2 also has a tutorial [41] for how packages are created and used, and documentation on the content of setup and CMakeLists [42].

Nodes

A node in the ROS 2 environment is an executable that runs a programmed task. Nodes can be a part of a larger system and use *topics*, Section 6.2.2, *actions* and *services*, Section 6.2.2 to communicate with other existing nodes within the same network [43, 44].

Nodes are structured as classes and with the use of the ROS 2 client library [45], which exist for Python [46] (rclpy) and C++ [47] (rclcpp). The client library implements ROS 2 specific language which enables the possibility for nodes written in different programming languages to communicate.

More documentation about classes exists for Python [48] and C++ [49]. How nodes are used to set up topics can be found in chapter 6.2.2 with reference to a more in-depth tutorial that includes how nodes are structured.

Launch & Parameters

Launch files are a helpful tool for projects that have multiple nodes running simultaneously. With this tool, one only needs to run a single command to launch all of the nodes at once instead of having to launch each node individually. With the way ROS 2 is designed, it is possible to launch nodes from different packages if necessary, as ROS 2 is capable of handling that as long the package is within the same workspace. Launch can be written in Python, XML and YAML, and can be read more about in ROS 2 documentation [50]

Nodes also can come with parameters, which act like the configuration of the node. These are written in launch files like Python or YAML, and make it possible for the user to change the value of these parameters during runtime [51].

Tutorials on how to create a launch file in ROS 2 [52] and how parameters are used in nodes [53], can be found in the ROS 2 documentation.

6.2.2 Communication in ROS

Topics

Topics [32, p.3] are based on subscribe/publish messaging with an example shown in Figure 6.2. Whenever a node publishes a new message to a topic, then all nodes that are subscribing to this topic will get sent the new message and run a callback function. This function is an assigned task that runs when a subscriber node gets a new message from the topic it subscribed to.

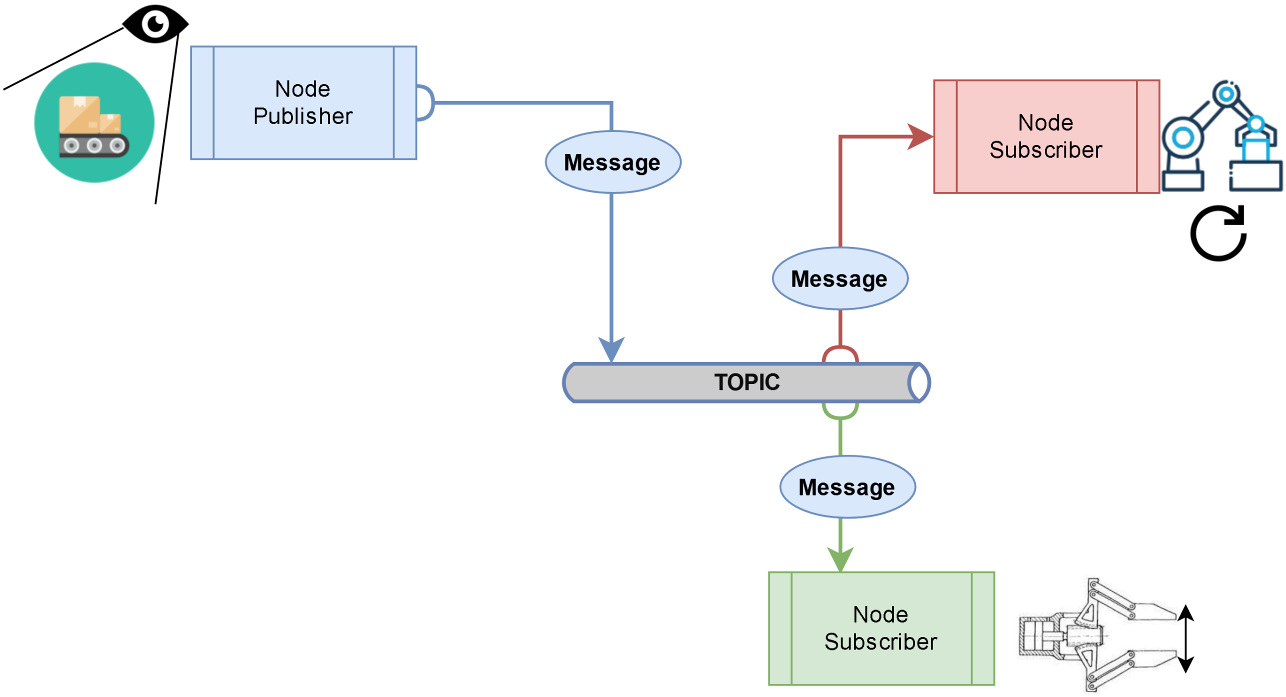


Figure 6.2: Example of how a topic is used in a ROS environment.

Implementation of Topics in Python

When implementing a subscriber or publisher in Python, one has to define which topic name it is going to communicate with and what type of message this topic uses. The documentation on standard messages used in ROS 2 can be found in their index API [54].

Unlike publishers, a callback function also needs to be defined for subscribers to run whenever a topic gets updated with a new message.

A more in-depth tutorial on how topics are implemented and used can be found in the ROS 2 documentation [55].

Action & Service

Services is an alternative way of communicating, Services use the call-and-response method. As shown in Figure 6.3, the client will send a request to the server (step 1) and the server will respond (step 2).

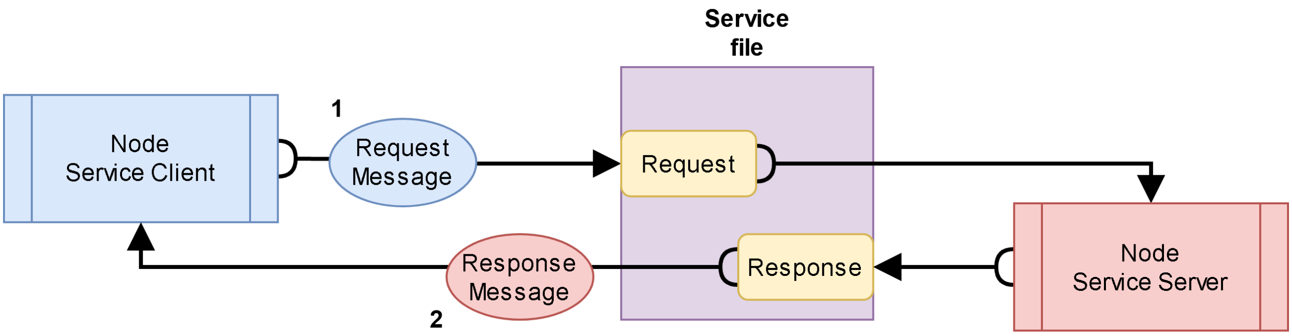


Figure 6.3: Example of how services are used in a ROS environment.

When using services, only one client node can use the call-and-response method at a time with the server node, and the client node has to wait for a response before continuing its task [56][32, p. 4].

Actions are a combination of topics and services. After the nodes have exchanged messages regarding the goal through services' call-and-response (steps 1 and 2), the client will then send a call, or request, for the result (step 3). While the server is running its process, the client can continue doing other tasks while getting feedback from the server (step 4). After completion, the server will then respond to the client with the result (step 5).

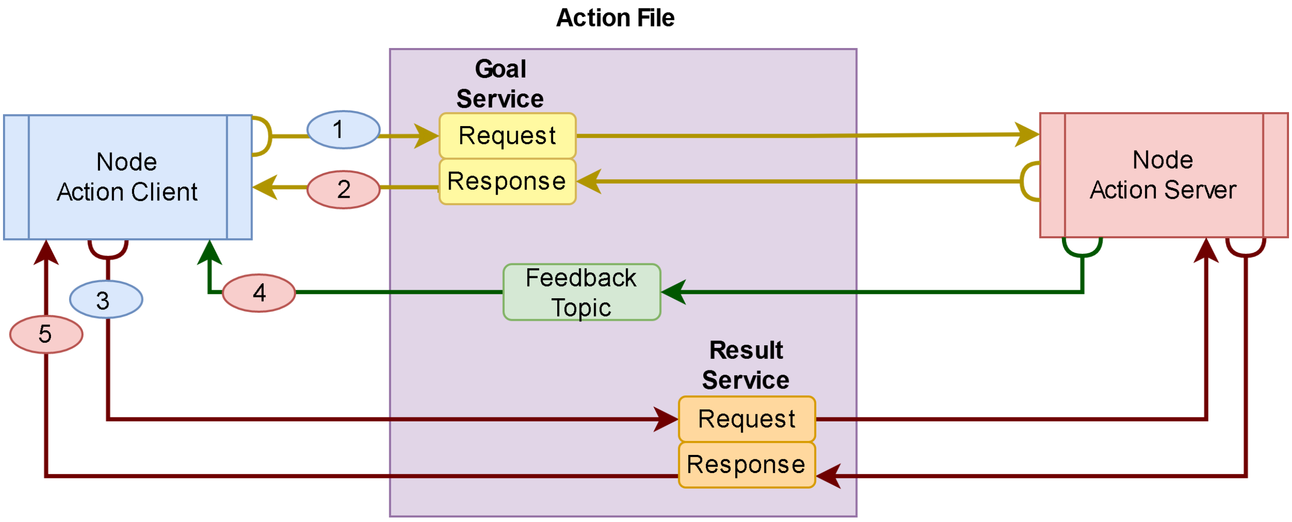


Figure 6.4: Example of how actions are used in a ROS environment.

Actions are better suited for long-running tasks as it allows the client to run other tasks while waiting for the results, and actions can also be cancelled [57][32, p. 4].

6.2.3 Gazebo Garden & BlueROV2 Garden

Gazebo Garden is a simulation program for simulating physical objects in different realistic environments. It has multiple open-source libraries for simulation, logging, sensors and plugin-based interface to physics engines [58]. Some of the important features include Linux compatibility, ROS2-integration, 3D graphics, the possibility of loading custom systems into the simulation and a precise physics engine [59].

BlueROV2 Garden is a workspace, which is still under development, that allows the user to spawn multiple BlueROV2 Heavy models in a Gazebo simulated underwater environment. This was created by Mikael Medina [4] and this workspace contains a physics engine simulating buoyancy and hydrodynamics, and enables the possibility of simulating ocean currents, thruster dynamics and collision amongst other things.

BlueROV2 Garden creates topics associated with the ROVs spawned, which is used in control. The topics used in this thesis are:

- Odometry with information about position, quaternions, linear- and angular velocity
- Thrusters 1 to 8

6.3 Mathematical Framework

This section introduces the necessary mathematical prerequisites to understand the functionality of the trajectory node.

6.3.1 Polynomial Trajectories

A common method to parametrize a trajectory is to use a polynomial, $q(t)$, where $q(t)$ is the setpoint at time t. These polynomial trajectories can be divided into two main categories: a single continuous polynomial, and piecewise polynomials, also known as *splines*. It is possible to use a polynomial of a high order to represent a single continuous trajectory. This strategy has the advantage that the derivatives are continuous throughout the entire trajectory. However unnecessary oscillations in the polynomial, known as Runge's phenomenon, are a distinct disadvantage to employing this strategy [60].

Splines

An alternative strategy is to use *splines*, or piecewise polynomials of a certain order that satisfies the given constraint requirements. Through this, it is possible to create trajectories using polynomials of a lower order that include an arbitrary amount of waypoints. The disadvantage of employing this strategy is that the derivatives are susceptible to being discontinuous in the *via-points* (points that connect two splines). This phenomenon is shown for cubic polynomials in Figure 6.5.

Cubic Polynomial Trajectories

Consider the cubic polynomial in Equation (6.1).

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{6.1}$$

Differentiating (6.1) with respect to t yields (6.2).

$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2. \tag{6.2}$$

Let t_0 denote the start time, and t_1 denote the end time of the spline. Set the constraints $q(t_0) = q_0$ and $\dot{q}(t_0) = v_0$, $q(t_1) = q_1$ and $\dot{q}(t_1) = v_1$. Where q_0, q_1 and v_1, v_1 are, respectively, the start and end time position and velocity. This results in four different equations, and as such, becomes a set of equations that requires six constraints (t_0, t_1, q_0, q_1, v_0 and v_1) to have a unique solution. Equation (6.3) expresses this set of equations in matrix form. Solving this will yield numerical values for a_0, a_1, a_2 and a_3. Inserting these into (6.1) results in a cubic polynomial that satisfies the constraints [15, p. 253-254].

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_1 & t_1^2 & t_1^3 \\ 0 & 1 & 2t_1 & 3t_1^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ q_1 \\ v_1 \end{bmatrix} \tag{6.3}$$

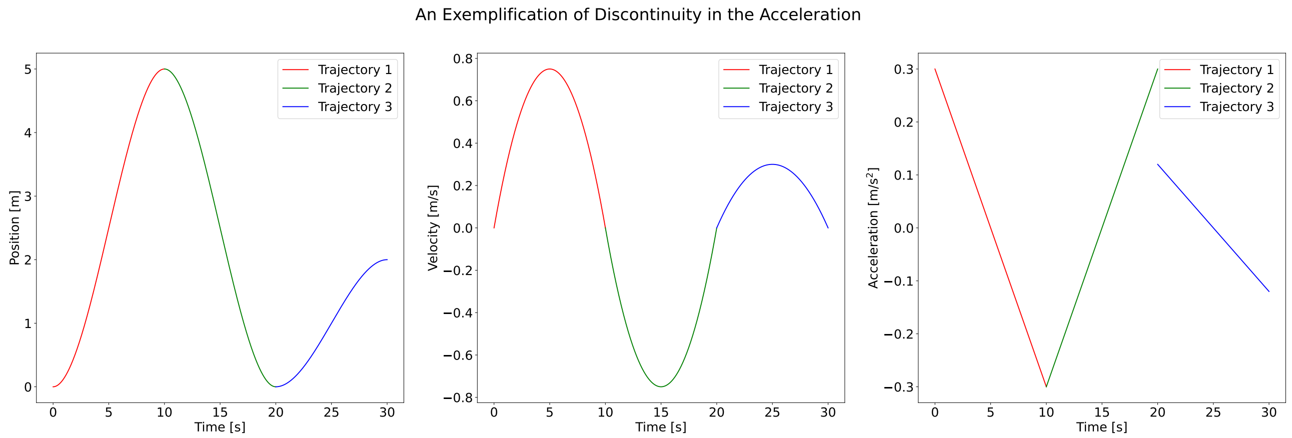


Figure 6.5: Three cubic polynomial splines showing discontinuity in the acceleration in the via-points

If it is necessary to add constraints on acceleration, quintic polynomial trajectories are an option to accomplish this. To put constraints on the jolt, a septic polynomial would be required.

6.4 Implementation

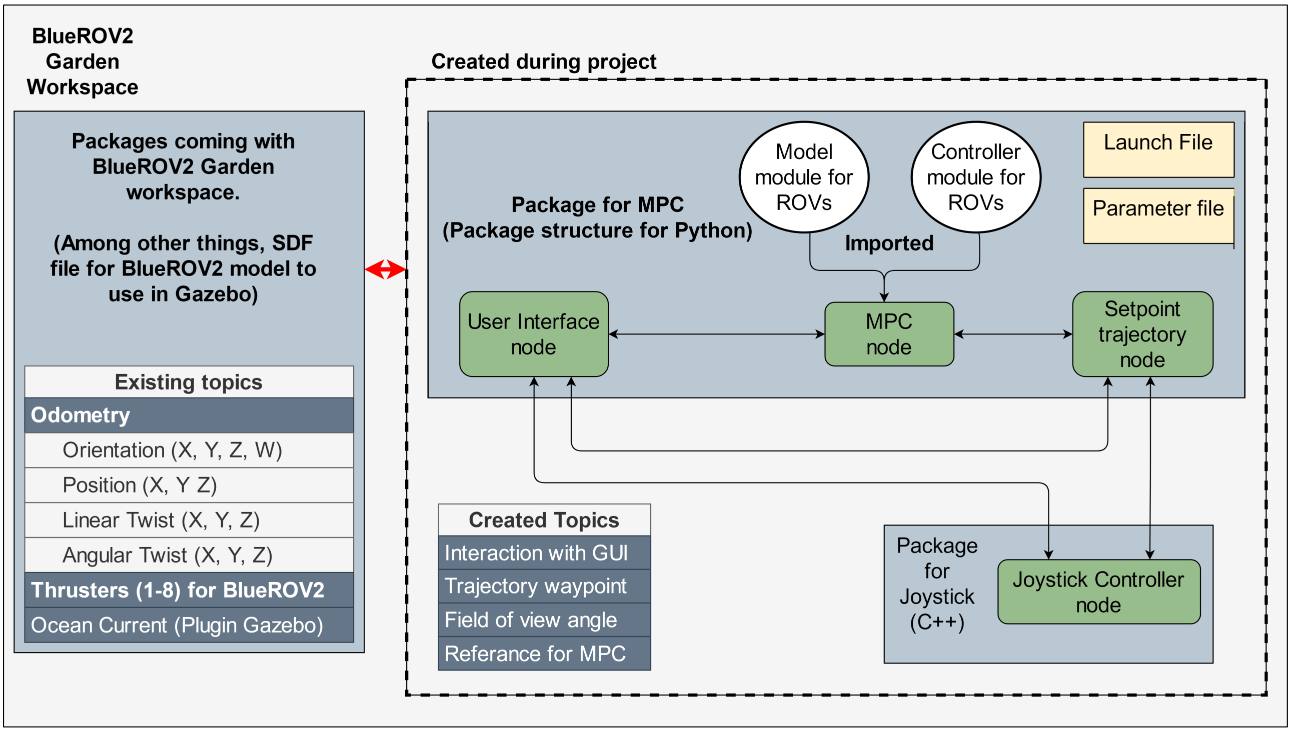


Figure 6.6: Overview of ROS 2 filesystem and connections

Figure 6.6 shows how packages created during this project (inside the dotted line) are structured inside the BlueROV2 Garden workspace. Inside the dotted lines are two packages, one for MPC which was created in Python and stores the majority of the nodes, including also do-mpc modules, a launch file and a parameter file, while the other is a package for the possibility of using a joystick for manual steering, written in C++.

6.4.1 Package for MPC

When creating a package to store the source code, ROS 2 offers a command to build the necessary structure of a package for either CMake or Python. Considering this package would store the majority of the nodes that would be created, including the actual MPC, Python were used, as the toolbox used for the MPC was a Python library.

Package Configuration

All ROS 2 Python packages need to be configured with two configuration files. The file *setup.py* describes how nodes are executed. This file contains *entry_points* that create executables.

The following entry points needed are:

- *"GUI = mpc_controller.GUI:main"*
- *"bluerov_mpc = mpc_controller.mpc_controller:main"*
- *"setpoint = mpc_controller.setpoint_publisher:main"*

The other configuration file is *package.xml* and this is where the ROS 2 package dependencies are defined. For this package, the following dependencies are required:

- rclpy - ROS client library for Python
- std_msgs - Library of standard messages used in ROS
- nav_msgs - Library used for navigation messages, specifically Odometry for this project
- geometry_msgs - Library used to get common geometric primitives, specifically Vector for this project
- ros2launch - Specify *ros2launch* package to make sure ROS 2 can recognise all launch file formats and that ROS 2 can launch the launch file with ROS 2 command.

6.4.2 Launch and Parameters

As mentioned in Subsection 6.2.1, launch files have the option to add parameters when launching different nodes. A YAML file was used for storing these parameters as it gives an overview of existing parameters for the user without having to access the source code of the launch file.

Table 6.2 lists the parameters found inside the YAML file.

Table 6.2: Table of YAML Parameters

Parameter variable	Description	Option
n_multi_agent	Size of the fleet	[1, 2 or 3] agents
debug_rov	Tells what should be printed to the terminal screen	[0 or 1] prints the coordinates of trajectory setpoint. [2 to 4] prints the mpc calculation for BlueROV2, BlueROV3 or BlueROV4
FOV_constraint	If the MPC should have any Field of View constraint	[True or False]
radius_setp	Radius on the circle around setpoint	[Float] in meters
distance_rovs	The distance between ROVs	[Float] in meters
FOV_range_deg	Hard constraint on field of view angle	[Float] in degrees
FOV_range_soft_deg	Soft constraint on field of view angle	[Float] in degrees
cycle_time_publish	How often it will do the calculation and publish it	[Float] in seconds

Launching Multiple Agents

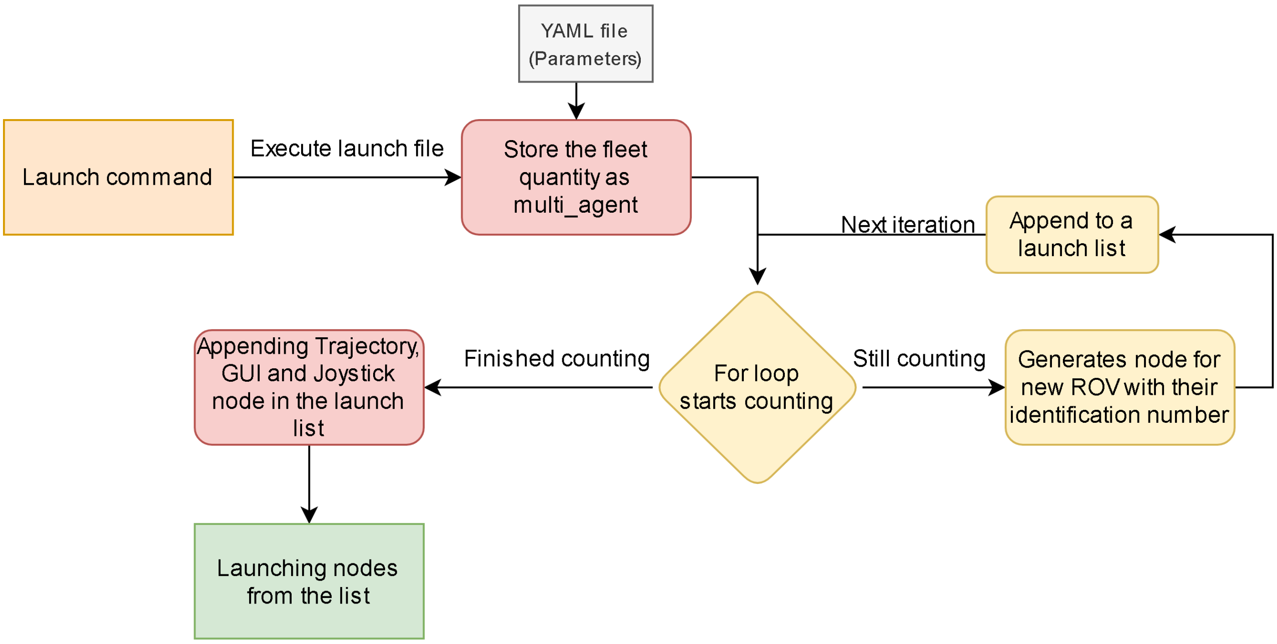

YAML file (Parameters)
Launch command
Execute launch file
Store the fleet quantity as multi_agent
Next iteration
Append to a launch list
Appending Trajectory, GUI and Joystick node in the launch list
Finished counting
For loop starts counting
Still counting
Generates node for new ROV with their identification number
Launching nodes from the list

Figure 6.7: Simple flowchart showing how launch-file handles multi-agent

The YAML file contains a parameter for the number of ROVs spawned called *multi_agent*, and this parameter gets extracted from YAML file into the launch file to generate a MPC node for each ROV in the fleet.

The way BlueROV2 Garden handles multiple agents is by giving each model a unique name. In this case, it was done such that the first ROV starts with “bluerov2”, and then continues with “bluerov3” and onwards. The number behind “bluerov” is how the models are identified and are used to make it able to give each node a unique name and also it is how MPC nodes know which ROV to control.

6.4.3 MPC Node

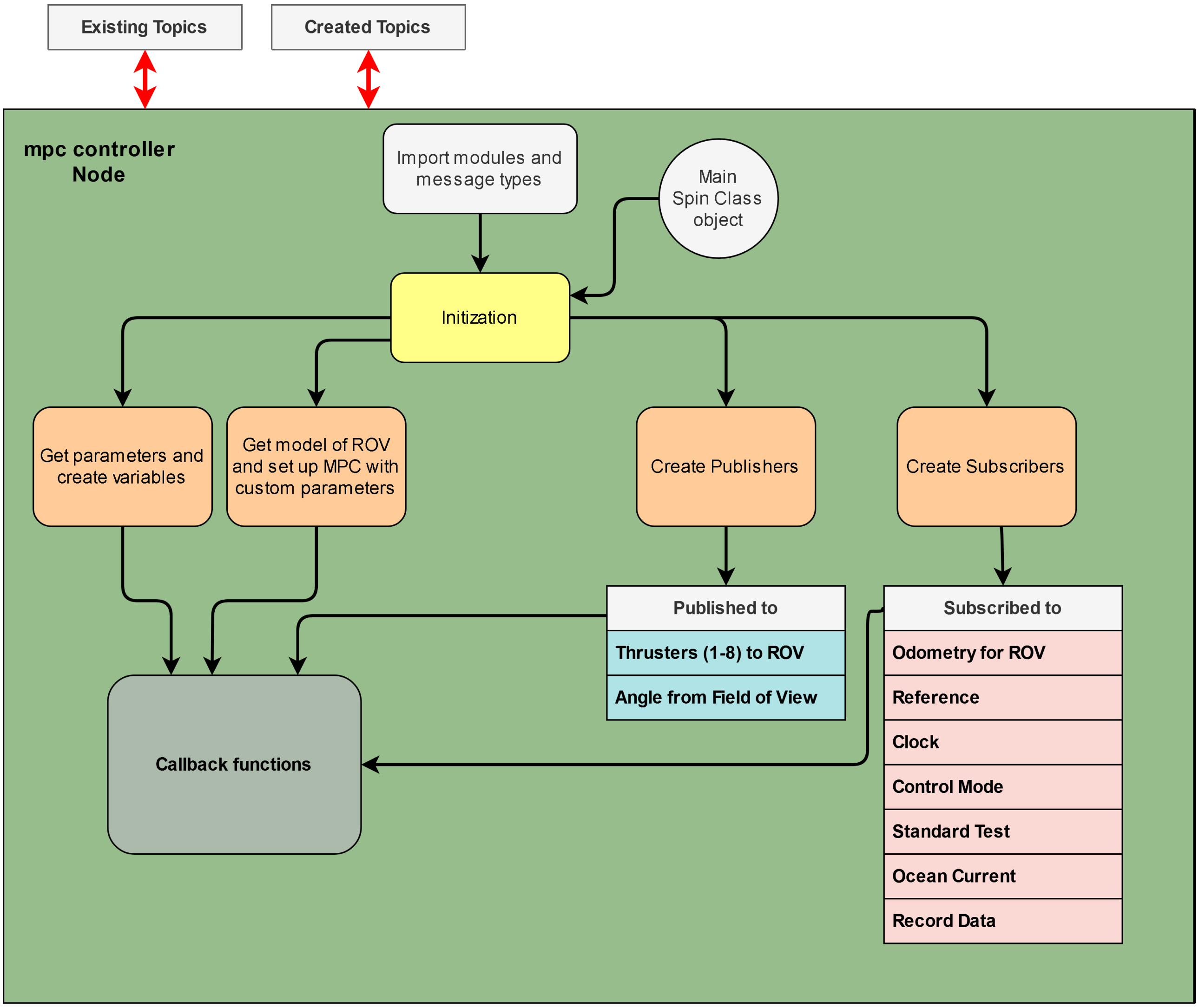


Figure 6.8: Overview of the node structure for mpc_controller.py

Figure 6.8 gives an overview of the structure of the MPC node with the initialisation phase in focus. In its initialisation phase, the node performs tasks like declaring parameters, configuring the MPC, and creating subscribers and publishers. During its runtime, subscribers will run callback functions when new messages arrive, while publishers run at a set rate, and publish to the thruster topics after the MPC has calculated new values.

Configuring Node for Multiple Agents

Launching multiple agents required modifications to the node. The MPC node takes the ROV's unique identification number into consideration and assigns each ROV with their own MPC. Since the only difference in topic names are the numbering (bluerov2, bluerov3 etc), it was sufficient to use that number as their *main_id* and extract that from the list of parameters that launched with the node.

Together with their identification number, the number of agents, known as *n_multi_agent*, is sent as a parameter to the node which is used for collecting the odometry of other ROVs in the fleet.

Implementing do-mpc

To avoid unnecessarily convoluting the source code, the mathematical model and the controller created with do-mpc were imported as modules to the node.

In its initiation phase when a node starts running, the necessary MPC parameters are extracted and used to initiate the controller created with do-mpc. From Table 6.2, the parameters that are used to configure the MPC controller are as follows:

- n_multi_agent
- FOV_constraint
- radius_setp
- distance_rovs
- FOV_range_deg
- FOV_range_soft_deg

At the end of the initiation, a timer will be set up to run in a cycle of the user's specification. For every cycle, a function will run, which is used for the MPC to calculate new values before publishing it to all eight thrusters. After some trial and error, 0.05 seconds seemed to work well enough for the purpose of this project.

Alternative Models and Controllers

The model and controller modules, seen in Figure 6.8, are built as classes and imported to the node to be used as regular class objects. Assuming the user is familiar with do-mpc [26, 29], the model can be readjusted for other ROV models and the controller, which amongst other things takes the model class in its function, can also be adjusted.

Topics

Subscribers and publishers are also created during the initiation phase of the node. Table 6.3 shows the topics that are used in the MPC node and if the topic is used for a subscriber or publisher.

Table 6.3: Table of Topics used in MPC node

Topics that are used from BlueROV2 Garden

Topic name	Description	Pub/Sub
/bluerov2_pid/bluerov{} /observer/nlo/odom_ned	Odometry to the bluerov model, {} is replaced with a number	Subscribed
/model/bluerov{}/joint /thruster1_joint/cmd_thrust	Used to to send values to thrusters. Each thruster has its own topic and goes from thruster1_joint to thruster8_joint	Published

Topics created from this Project

Topic name	Description	Pub/Sub
/ref	Used to exchange the coordinates of setpoint	Subscribed
/clock	Used to exchange the *real-time*	Subscribed
/control_mode	Communication from GUI about which modes is being used	Subscribed
/std_test	Communication from GUI about which standard tests to do	Subscribed
/record_data	Communication from GUI about if data should be recorded	Subscribed
/filename_data	Custom name on the file for recorded data	Subscribed
angle/from_{}_to_{}	Used to exchange the field of view angle between ROVs. An example could be from 2 to 3.	Published

6.4.4 GUI

A graphical user interface node, or *GUI.py*, was created to simplify the user interaction with the simulator and the BlueROV2 Garden workspace.

To publish, *pub*, a message to a topic from the terminal, it can be done by using the command [61]

```
$ ros2 topic pub <topic_name> <msg_type> '<args>'
```

and to see what is being published on a topic, one would have to *echo* the topic name. With a GUI however, it would allow the user to set new coordinates for the setpoint and create a waypoint quicker. It also would access the information that was thought to be useful for the user, like the positioning of the ROVs and if it is violating any constraint that is mentioned in Subsection 5.2.2. More details about the use of GUI can be found in Section 7.1.2, but it is still structured as a regular node.

6.4.5 Trajectory Node

The mathematical framework behind trajectory planning is explained in Section 6.3.1. This section describes the ROS 2 implementation of the trajectory planning feature.

The trajectory node, or *setpoint_publisher.py*, is a node that is used to send the path of the trajectory to the MPC nodes. It is also implemented with a set of standard tests, explained in Section 7.1.3. The trajectory node is controlled from the GUI node through topics and publishes a reference at a set cycle to the topic */ref*.

Figure 6.9 is a flowchart of the functionality implemented in the node. A trajectory is computed by entering the goal position and velocity. The stop time is not needed, as it is computed by setting an average velocity, and then computing the end time, by calculating the distance from the start point to the endpoint, and dividing by the average velocity.

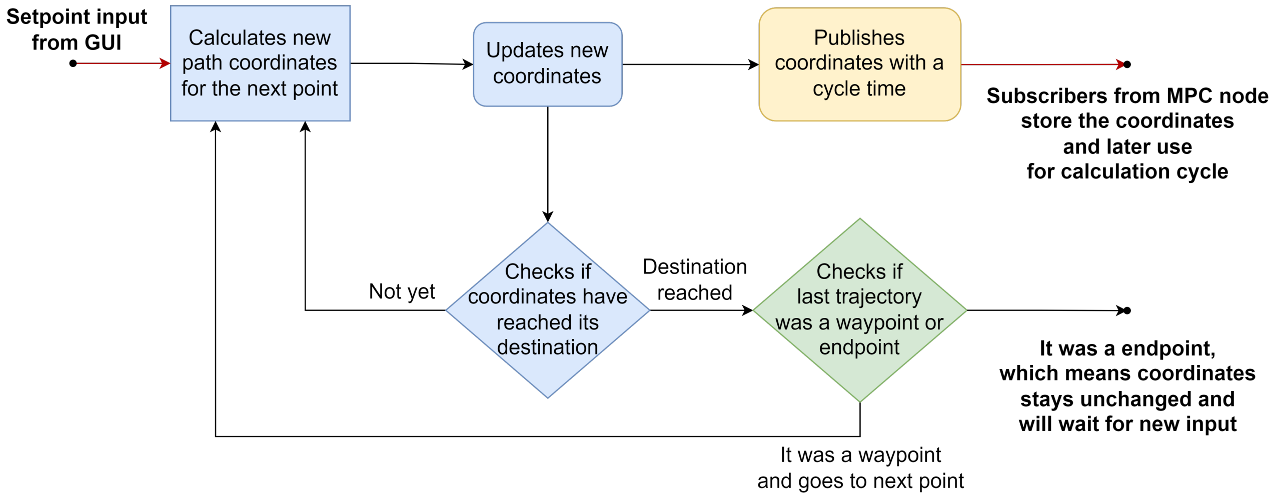

Setpoint input from GUI
Calculates new path coordinates for the next point
Updates new coordinates
Publishes coordinates with a cycle time
Subscribers from MPC node store the coordinates and later use for calculation cycle
Not yet
Checks if coordinates have reached its destination
Destination reached
Checks if last trajectory was a waypoint or endpoint
It was a endpoint, which means coordinates stays unchanged and will wait for new input
It was a waypoint and goes to next point

Figure 6.9: Flowchart - Trajectory node when getting setpoint from user

6.4.6 Package for Joystick Controller

The functionality of using a joystick to steer the setpoint manually was added as an option. This package was implemented in C++.

The *joy_con* package with its node steers the setpoint and makes it possible to manually control the fleet. The node is dependent on another node, *joy*. The *joy* node is a part of the standard ROS 2 distribution, the node detects inputs from the joystick and publishes these as a *Joy* message to the */joy* topic. The *joy_con* node subscribes to the */joy* topic, and converts inputs in this topic, to the setpoint topic, */ref*. Both of these nodes were added to the launch file in the *mpc_controller* package.

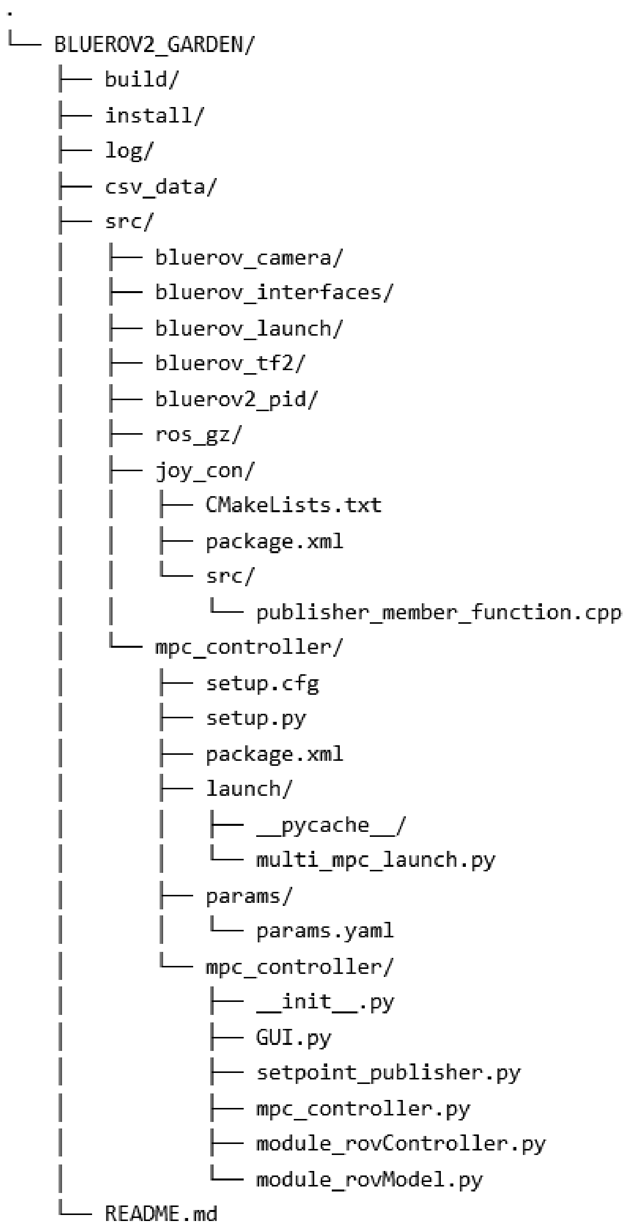
When the user changes the control mode to “manual” in the GUI, a message is published to the topic */controller_mode*, that both the trajectory and joy_con node subscribes to, in order to avoid conflicts when publishing to */ref*.

Moving the setpoint in the xy-plane is done by moving the left joystick, and in the z-axis is accomplished with the triggers.

6.5 Results and Findings

6.5.1 Filesystem

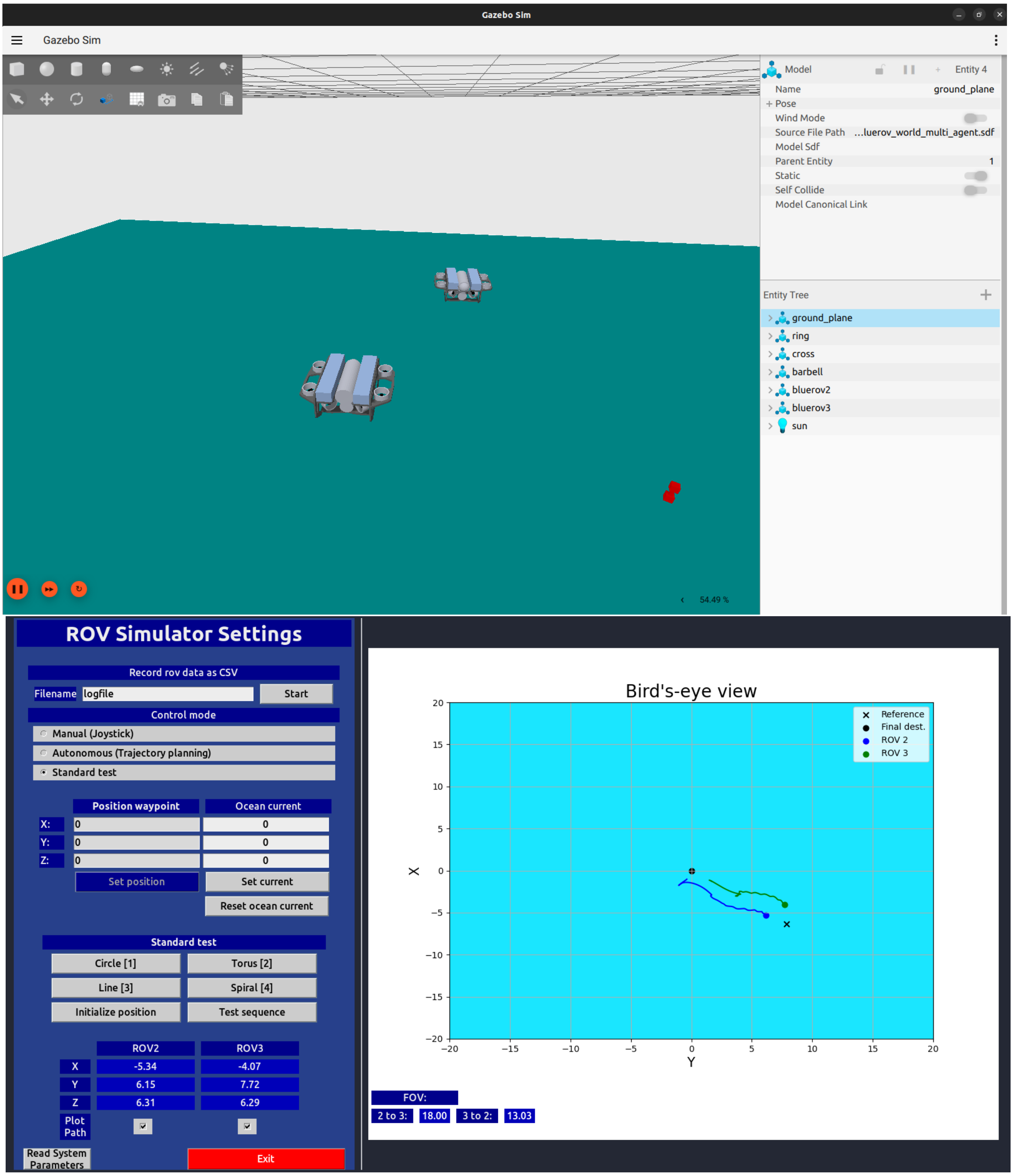
Figure 6.10 shows the filesystem of BlueROV2 Garden workspace, with details of contents of the packages, joy_con and mpc_controller, that were specifically created for this project.



```
.
└── BLUEROV2_GARDEN/
    ├── build/
    ├── install/
    ├── log/
    ├── csv_data/
    ├── src/
    │   ├── bluerov_camera/
    │   ├── bluerov_interfaces/
    │   ├── bluerov_launch/
    │   ├── bluerov_tf2/
    │   ├── bluerov2_pid/
    │   ├── ros_gz/
    │   ├── joy_con/
    │   │   ├── CMakeLists.txt
    │   │   ├── package.xml
    │   │   └── src/
    │   │       └── publisher_member_function.cpp
    │   └── mpc_controller/
    │       ├── setup.cfg
    │       ├── setup.py
    │       ├── package.xml
    │       ├── launch/
    │       │   ├── __pycache__/
    │       │   └── multi_mpc_launch.py
    │       ├── params/
    │       │   └── params.yaml
    │       └── mpc_controller/
    │           ├── __init__.py
    │           ├── GUI.py
    │           ├── setpoint_publisher.py
    │           ├── mpc_controller.py
    │           ├── module_rovController.py
    │           └── module_rovModel.py
    └── README.md
```

Figure 6.10: Folder structure of BlueROV2 Garden with the filesystem of Joystick and MPC packages

6.5.2 Launch of Packages w/ BlueROV2 Garden - Desktop Interface



Name	ground_plane
+ Pose	
Wind Mode	
Source File Path	...luerov_world_multi_agent.sdf
Model Sdf	
Parent Entity	1
Static	
Self Collide	
Model Canonical Link	

- ground_plane
- ring
- cross
- barbell
- bluerov2
- bluerov3
- sun

Filename: logfile

○ Manual (Joystick)
○ Autonomous (Trajectory planning)
● Standard test

	Position waypoint	Ocean current
X:	0	0
Y:	0	0
Z:	0	0

	ROV2	ROV3
X	-5.34	-4.07
Y	6.15	7.72
Z	6.31	6.29
Plot Path	☑	☑

FOV: 2 to 3: 18.00 3 to 2: 13.03

Figure 6.11: Launching BlueROV2 Garden (upper) and launch file from mpc_controller (below) together

6.5.3 System Topology

The graph was made by using the ROS 2 tool RQt-graph [62].

RQt-graph is a plugin for ROS 2 which is a GUI used to visualise the system topology. In Appendix C, the system topology for one and two agents can be found. These are graphical visualisations of the ROS 2 network.

Some comments regarding the RQt-graphs:

- *bluerov2_pid* is a package name that has the odometry in it, and so even though there is a PID-controller in the BlueROV2 Garden workspace, for it to be used, one has to publish to the topic name */bluerov2_pid/bluerov2/references*, which would then take over and render the MPC controller useless. This is because the PID controller has no limitation on what values it can publish to thrusters.
- Circles represent the node that is running, while squares with only text within the square, represent topics.

6.6 Discussion and Analysis

6.6.1 Communication

This project has solely used the subscriber/publishers model as a way to communicate with other nodes. However, the background for this thesis mentioned ROVs cooperating in performing complex tasks like picking up loose fishing nets. This is where Action servers and clients would be a beneficial asset to use for communication between ROVs.

Consider the example previously stated, Actions would be used to make sure that while one of the ROVs is still getting into position, the ROV that is in position and is waiting for the ready signal from the action server, is still able to perform control actions. Services are unsuitable for tasks like these because it relies on the call-and-respond method. This way, one does not only achieve a higher degree of autonomy in the ROVs, but it also enables more intelligent cooperation between them.

Another possible use case for Actions is in the trajectory node. If the reference is an Action server instead of a topic, it would enable the possibility to only update the reference if certain criteria are met. Examples of criteria include; all ROVs being within a certain distance of the current reference, or the FOV constraint being upheld. This functionality can be useful if, in a real-world scenario, an ROV malfunctions, the line of sight is broken and the ROVs are no longer able to communicate, the ROVs will then have an estimate of each other's position.

6.6.2 Parameters

The usage of a YAML file in this project is intended to give the user an overview of what parameters exist and can be changed before launch. Generally, YAML files can be modified during runtime, but the current implementation does not support this feature. The initialisation of the MPC only happens in the initiation phase of the MPC node, which means if the user changes MPC parameters during runtime, then this will have no effect on the controller unless the node is re-launched. However, to re-initialise the MPC controller with new parameters, a new topic can be created for this purpose. This can be particularly useful when tuning parameters on the physical system, without having to restart the entire system.

6.6.3 Reuse of Code

ROS 2 Philosophy

The ROS 2 philosophy is rooted in *modularity*. This means that the communication between the nodes in the network should be as standardised as possible, using the standard message types, with each node performing one specific task or a subset of tasks. This enables increased reusability of code. The MPC node utilises only default ROS 2 message types, including the *odometry* message type. In addition, since the node imports the controller and the model separately, one can readjust the model with no alterations to the source code. However, depending on the changes in the controller, the source code of the MPC node might need to be adjusted to account for more substantial modifications. The user would only need to be wary of the topic names for odometry and thrusters, and the number of publishers created for thrusters. It can, therefore, be argued that this package follows the ROS 2 philosophy.

Transfer to Real-Life Application

For this project's packages to be applicable to physical ROVs, it is uncertain to which extent modifications need to be made. However, something that is believed would need to be changed is the launch of multiple MPC nodes. Considering that each ROV is going to have its own single-board computer, changes and adjustments to the source code would be needed for it to only launch its own MPC node and still take other ROVs in the fleet into consideration like its odometry for the field of view constraint.

Chapter 7

Simulation

After designing the controller and implementing a system ready for use, the next step is to simulate and perform tests to analyse the system's robustness. This is a crucial part of research to ensure a smoother transition to real-life application.

During the simulation phase, testing was done with both Python and ROS 2 with Gazebo. With Python, simulations were done through do-mpc's simulation which enabled testing with an ideal system model. With ROS 2, Gazebo enables interaction with a non-ideal system in real time.

A set of standardised tests were created for analysing the controller. Each test ran numerous times for data collection and was later analysed to describe the controllers' performance.

7.1 Design and Implementation

7.1.1 Python Simulator

The do-mpc toolbox has the possibility to run a controller, simulator and state estimator [26]. Configuring do-mpc with the model of the system and then running the controller, simulator and estimator sequentially, enabled testing the MPC with an ideal system model. Meaning that the system model used in the MPC would not deviate from the actual dynamics of the system, as it does in real-world applications.

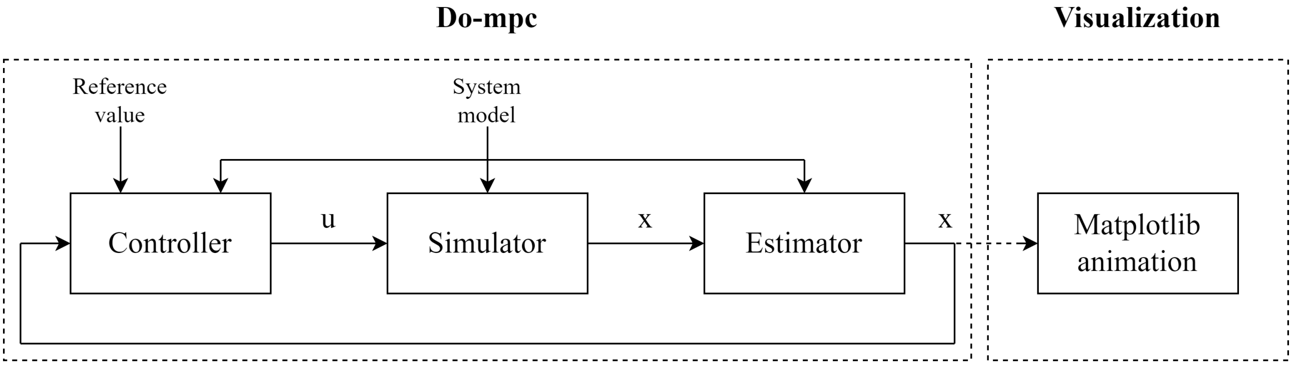


Figure 7.1: Block diagram of do-mpc simulator

Figure 7.1 is a block diagram of the signal flow in the do-mpc simulator. The signal notation **u** and **x** is in state space representation, where **u** is the output vector, which in this system is the signal to the thrusters and **x** is the states of the system, which in this system is the position and attitude of the ROVs.

Some important distinctions between the Python simulator and the Gazebo simulator:

- **The Python simulator is not implemented in ROS 2:** The Python simulator does not run in real-time, and there is no need to transfer any data to any other nodes, as the controller, simulator and estimator are being run within the same script. Therefore ROS is not needed.
- **The Python simulator used the system model using Euler angles:** There were problems with getting a stable version of the Python simulator using the quaternion model. Therefore the model using Euler angles was used, as the physical properties of the system should remain the same.

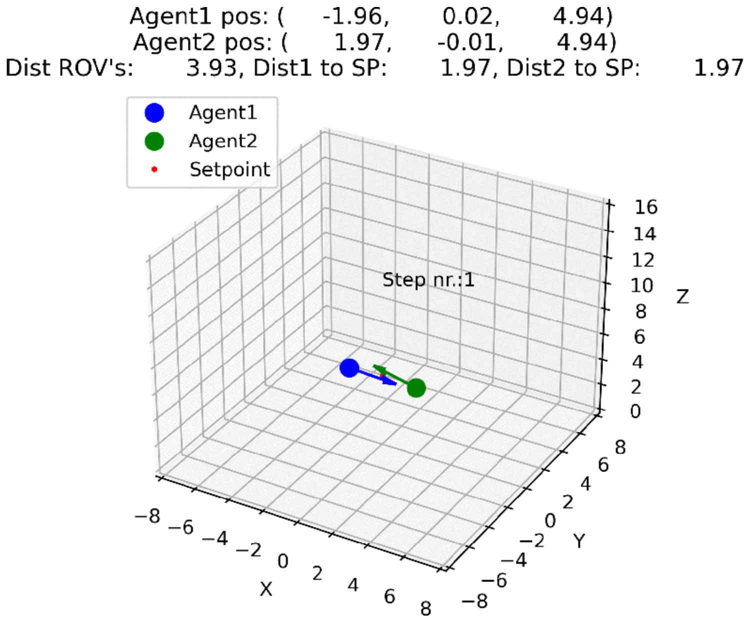


Figure 7.2: Screenshot from the do-mpc simulator animation

To visualise the result from the do-mpc simulator, a Python script, which used the library Matplotlib [63] to create animations of the states of the system, was created. This script plotted the positions of the simulated ROVs as scatter points, and the directional vectors as quivers. While running the simulator the states were written to CSV files at every iteration, the CSV files are then read by the visualisation script for visual interpretation of the results. A screenshot from the do-mpc simulator animation can be seen in Figure 7.2.

7.1.2 Gazebo Simulator

The Gazebo Simulator is an appropriate intermediary step before implementing the system on the physical model, as it can interface with ROS 2 and simulate asynchronously to the MPC. This creates a real-time simulation that is closer to the conditions of physical testing.

The implementation of the control system in ROS 2 is described in Chapter 6, with a description of how the control system interfaces with the BlueROV2 garden package [4], used for simulating the dynamics of the BlueROV2 Heavy.

GUI

To interact with the Gazebo simulator, a Graphical User Interface (GUI) was made, as mentioned in Section 6.4.4. The GUI also was used as the control unit for the standard tests. The GUI was made using the Python package PySimpleGUI [64] and was implemented as a node in ROS 2.

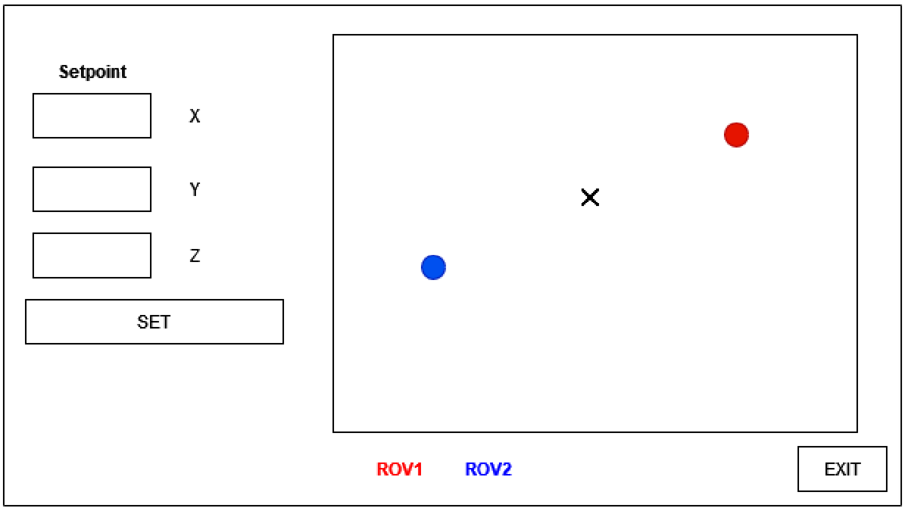
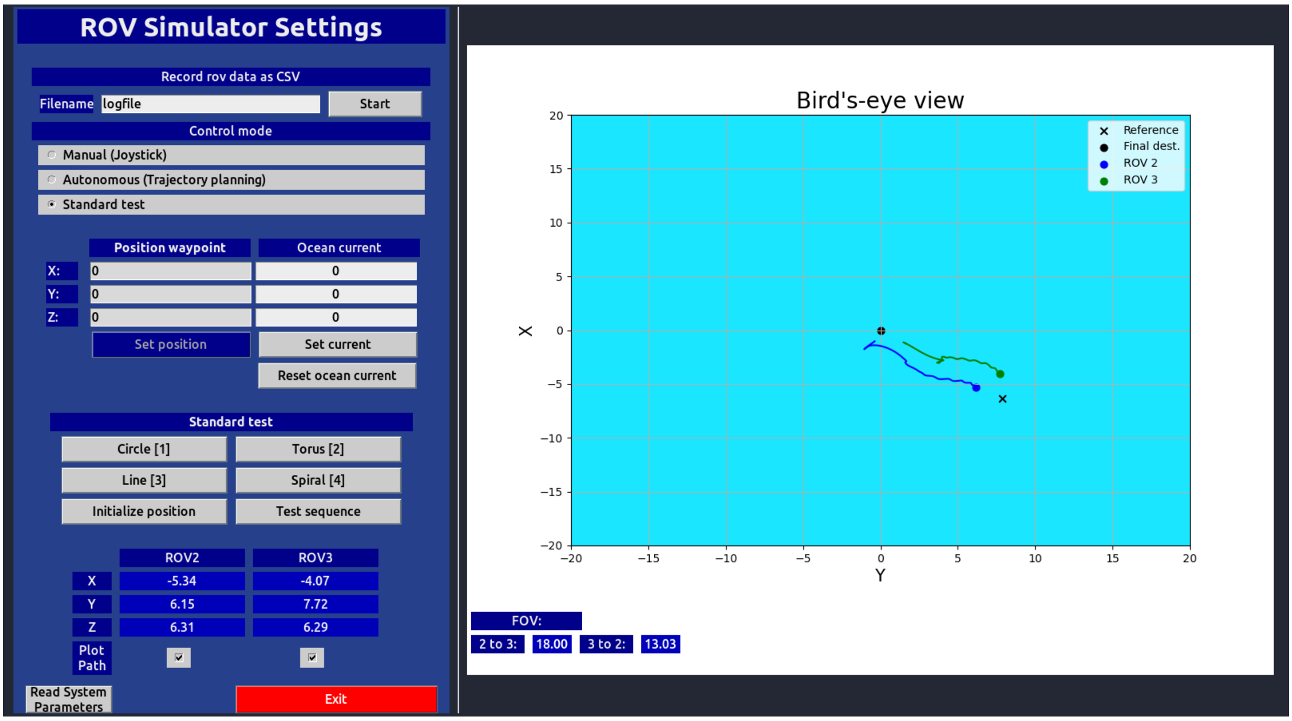


Figure 7.3: Initial sketch of layout for the GUI

Figure 7.3 is the initial sketch of the layout for the GUI, it contained functionality to set positional setpoints and a graphical representation of the positional states of the two ROVs. The ROVs were plotted onto a two-dimensional Matplotlib canvas that was imported to the GUI.

When the functionality planned in the initial sketch had been implemented, it was decided to expand the functionality. In the final version of the GUI, the following was implemented:

- Setting the control mode:
    - Joystick control - manual control with joystick.
    - Autonomous control - setting waypoints for the trajectory planning.
    - Standard tests - running standard tests individually or sequentially.
- Enabling and disabling logging of the data from the simulator into CSV files.
- Plotting the path of the ROVs.
- Setting ocean current for the Gazebo simulator.
- Displaying the position of the ROVs.
- Observing the FOV angle between the ROVs. With indication of broken constraint.



Filename: logfile

○ Manual (Joystick)
○ Autonomous (Trajectory planning)
● Standard test

	Position waypoint	Ocean current
X:	0	0
Y:	0	0
Z:	0	0

	ROV2	ROV3
X	-5.34	-4.07
Y	6.15	7.72
Z	6.31	6.29
Plot Path	☑	☑

FOV:

2 to 3:	18.00	3 to 2:	13.03

Figure 7.4: Final version of GUI

The final version of the GUI is displayed in Figure 7.4.

The GUI had the functionality to run the standard tests (described in Section 7.1.3), sequentially. Sequential testing made it possible to run all standard tests numerous times sequentially without requiring an operator to oversee the test. This also ensured consistency in the testing.

To ensure that the initial conditions in all the tests were similar, the ROVs had to hold certain conditions over a period of time before the next test could be run. The conditions were:

$$\alpha < 15^\circ \wedge d_d < 2m \tag{7.1}$$

where α is the FOV-angle between the ROVs and d_d is the distance between the ROV and the starting point for the tests. For the next standard test in the sequence to start both these conditions had to be fulfilled for a total of 5 seconds for both ROVs.

7.1.3 Standard Tests

To perform a statistical analysis of the performance and robustness of the control system, a set of standardised tests was designed. Using similar initial conditions for all tests, allowed for the same scenario to be simulated multiple times. Four distinct trajectories were designed. Namely, the circle-, torus-, line- and spiral test. Figure 7.5 shows 3D visualisation of the test trajectories.

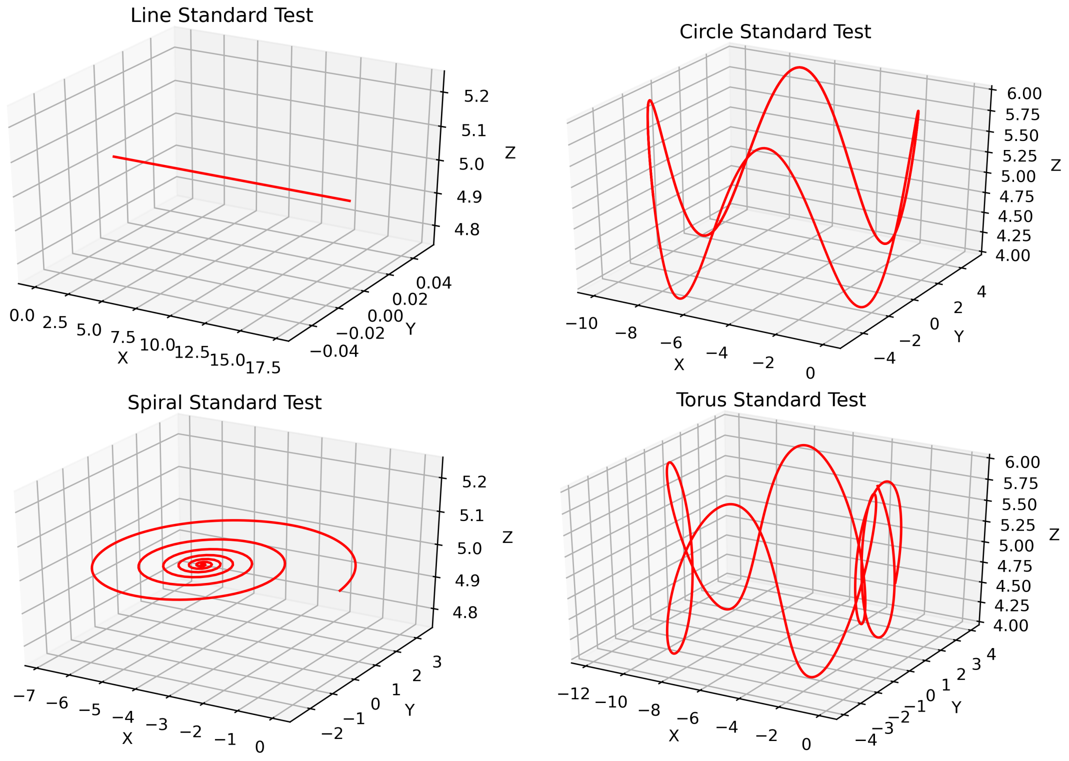


Figure 7.5: 3D-plots of the standard tests

The standard tests were designed with an increasing level of complexity, with the line test being the least complex, then the circle test, the spiral test, and the torus test being the most complex.

Line Test

The test is implemented with a cubic polynomial along the x-axis, with constraints $q_0, v_0, v_1 = 0.2$, $q_1 = 15$, and an average speed of 0.2 m/s. Furthermore $x(t)$ is limited to $x_{max}(t) = 15$ These constraints yield (7.2). This test runs for 130 seconds.

$$x(t) = \begin{cases} 5.33 \times 10^{-3} t^2 - 3.55 \times 10^{-5} t^3 & \text{for} \quad t \leq 75 \\ 15 & \text{for} \quad t > 75 \end{cases} \tag{7.2}$$

$$y(t) = 0 \tag{7.3}$$

$$z(t) = 5 \tag{7.4}$$

Circle Test

This test is defined by the following Equations (7.5), (7.6), and (7.7). In the xy-plane, the setpoint traces a circle with a radius of five meters and a period of 200 seconds. Along the z-axis, the trajectory moves with a sine wave, with a period of 50 seconds. This test runs for one period along the xy-plane, or 200 seconds.

$$x(t) = 5\cos\left(\frac{\pi t}{100}\right) - 5 \tag{7.5}$$

$$y(t) = 5\sin\left(\frac{\pi t}{100}\right) \tag{7.6}$$

$$z(t) = \sin\left(\frac{\pi t}{25}\right) + 5 \tag{7.7}$$

Spiral Test

The spiral test uses a path that traces a circle in the xy-plane, but with decreasing radius and increasing frequency as the time, t, increases. This is defined by Equations (7.8), (7.9) and (7.10). This test runs for 250 seconds.

$$x(t) = (4 - 0.015t)\cos\left(\frac{\pi t}{100 - 0.3t}\right) - 4 \tag{7.8}$$

$$y(t) = (4 - 0.015t)\sin\left(\frac{\pi t}{100 - 0.3t}\right) \tag{7.9}$$

$$z(t) = 5 \tag{7.10}$$

Torus Test

The Torus test traces a trajectory of a helix that is wrapped around a torus. A torus is a surface that can be created by revolving a circle one full rotation about an axis that is co-planar to the circle. One possible parametrization for this trajectory is shown with Equations (7.11), (7.12), and (7.13). Where r_1 is the major radius, the outer edge of the torus, and r_2 is the minor radius. The ratio $\frac{\phi}{\theta}$ gives the number of winds around the torus. The following parameters were used: $\phi = 200$, $\theta = 35$, $r_1 = 7$, $r_2 = 2$. This test runs for 400 seconds, or one period in the xy-plane [65].

$$x(t) = \left(r_1 + r_2 \cos\left(\frac{\pi t}{\theta}\right)\right)\cos\left(\frac{\pi t}{\phi}\right) - (r_1 + r_2) \tag{7.11}$$

$$y(t) = \left(r_1 + r_2 \cos\left(\frac{\pi t}{\theta}\right)\right)\sin\left(\frac{\pi t}{\phi}\right) \tag{7.12}$$

$$z(t) = r_2 \sin\left(\frac{\pi t}{\theta}\right) + 5 \tag{7.13}$$

7.1.4 Controller Parameters

The parameters for the controller were found by using an experimental method. These parameters were initially set by prioritising the objectives set for the controller and setting the gains and penalty terms associated with the objectives with the highest priority, with a bigger gain value.

The main objectives for the controller in prioritised order were:

1. Communication between ROVs.
2. Collision avoidance.
3. Path following.

For the FOV constraints the directly associated gains are either K_A and K_{FOV} or K_ϕ, K_θ, K_ψ and K_{FOV}, depending on the model used. For the collision avoidance the directly associated gain is K_D and for the path following the directly associated gains are K_x, K_y and K_z.

The gains are intertwined, and thus adjusting one gain will have some impact on the performance of the system as a whole. Therefore, adjusting the gains was an iterative process where the tweaking had the following procedure:

1. Conducting a simulation test that involves the ROVs tracing a designated path while observing their performance with respect to the aforementioned primary objectives.
2. Analysing the result, and the impact of the parameter set.

3. Deciding on what parameter to tweak next, by using the results from the previous test.

This procedure was repeated until the performance of the controller was satisfactory.
After testing the following parameters were used for the controller:

Table 7.1: Controller parameters

Symbol	Description	Value	Simulator
K_x	Cost function positioning gain constant	35	Python and Gazebo
K_y	Cost function positioning gain constant	35	Python and Gazebo
K_z	Cost function positioning gain constant	60	Python and Gazebo
K_A	Cost function attitude gain constant	12	Gazebo
K_x	Cost function positioning gain constant	25	Python and Gazebo*
K_y	Cost function positioning gain constant	25	Python and Gazebo*
K_z	Cost function positioning gain constant	12	Python and Gazebo*
K_A	Cost function attitude gain constant	30	Gazebo*
K_ϕ	Cost function attitude gain constant	20	Python
K_θ	Cost function attitude gain constant	20	Python
K_ψ	Cost function attitude gain constant	20	Python
K_D	Penalty term distance constraint	70	Python And Gazebo
d	Distance constraint distance	2	Python and Gazebo
K_{FOV}	Penalty term FOV constraint	70	Python And Gazebo
γ_{soft}	Soft FOV constraint angle	30	Python and Gazebo
γ_{hard}	Hard FOV constraint angle	60	Python and Gazebo
R	R-term gain	0.1	Python And Gazebo
n_horizon	Controller parameter	20	Python and Gazebo
t_step	Controller parameter	0.1	Python
t_step	Controller parameter	0.05	Gazebo
n_robust	Controller parameter	2	Python and Gazebo
ipopt.max_iter	Controller parameter	3000	Python
ipopt.max_iter	Controller parameter	25	Gazebo

The explanation of the symbols in Table 7.1 can be found in Chapter 5.

One noteworthy parameter for the controller is ipopt.max_iter, which sets the maximum number of iterations for the solver that tries to solve the optimisation problem. This was set to 25 in the Gazebo simulator and 3000 in the Python simulator. The justification for this is that when the solver tried to find the solution to the optimisation problem, all other functionality in the node was blocked. Running thousands of iterations of the solver could last close to a minute.

7.1.5 Test Scenarios

Prior to deciding on the test scenarios to focus on, preliminary tests were conducted to get a sense of which scenarios would yield the most interesting results. The full list of tests conducted can be found in Appendix E, with notes of the results from each test.

The specific scenarios decided to test are physical disturbances, modifications in the model parameters and communication errors. The following list contains the specific conditions tested:

- *Default test*: Benchmark test to get a reference point to which the rest of the tests can be compared too. This test has no disturbances, model alterations or communication errors.

- *Model mass*: Model error test to see how sensitive the system is to model errors. The mass was changed to double of correct mass.
- *Package loss*: Communication error test to see performance with 70% package loss.
- *Currents with waves*: A test with physical disturbances, to see the robustness of the controller against ocean currents. The current speed in the x-direction is set to 0.25 m/s and 0.5 m/s. The current speed in the z-direction is the same as the x but with a change between positive and negative direction every 2 seconds, which is a frequency of 0.5Hz. This is a very simplistic representation of what would be called *waves* and does not represent the true nature of waves.
- *Added mass*: Model modification. Test how deviations in the added mass parameters affect the MPCs' performance. Tests were run with halved and doubled added mass.
- *Damping*: Model modification. Test how deviations in the damping parameters affect the MPCs' performance. Tests were run with halved and doubled damping.
- *Circular setpoint*: Test how the proposed circular setpoint solution, derived in Chapter 5, affect the performance of the ROVs

The four standard tests are used to test these parameters. Only one disturbance parameter is tested at a time to try and isolate the system's weaknesses. With tests being time-consuming, three computers with differing hardware were used. Ideally, all tests should be done on one computer or computers with equal hardware. However, this was taken into account by running default tests on all three computers for comparison. The list of the relevant hardware of the computers used in testing can be found in Appendix D.

Table 7.2: Overview of test run on each computer

Test Type	PC
Default	All
Halved added mass	1
Doubled added mass	1
Halved damping	2
Doubled damping	2
Doubled mass	3
Packet loss 70%	2
0.25 m/s Current with waves	3
0.5 m/s Current with waves	3
Circular setpoint	2

7.1.6 Statistical Analysis

The robustness of the controller is measured by analysing how well it performs the three main objectives set for the controller. The criteria for which each objective is measured are as follows:

Communication between ROVs: The communication is measured by checking if the ROVs break the FOV constraint at any point when running standard tests, the maximum angle set for the FOV constraint is 60°. This angle was chosen by considering the FOV of the camera, as mentioned in Section 3.3.2, which is 110° ± 90° tilt. Therefore it is considered reasonable with a FOV of 120° for the optical sensor, giving a FOV angle of 60°.

Collision avoidance between ROVs: Collision avoidance is measured by analysing the standard tests, and checking if the distance between the ROVs at any point is less than 75 cm. This value is determined by considering the physical size of the ROV, seen in Figure 3.2, which is a maximum of about 25 cm from the geometrical centre of the ROV. Setting the minimum distance to 75 cm gives a safety margin to ensure that no collision could have occurred.

Trajectory planning and path following: Path following is measured in the time shift of the tests in the circle standard test.

To test the system thoroughly, it was decided to run each test about 100 times with two agents per test condition, totalling 200 individual tests for each condition. Each test having 200 CSV files will give an adequate amount of data to analyse the system. Since all tests are conducted multiple times, it is impractical to show each individual test, therefore it was decided to show these data points:

- *Median value*: Was chosen in favour of the average value since it is not affected by outliers.
- *Maximum and minimum value*: To show the worst outcome and potential outliers.
- *90th and 10th percentiles*: Indication of the spread of the data points.
- *Failure rate*: Percentage of unique tests that failed to uphold the constraints.
- *Time shift*: Time shift from peak to peak, from the reference to the median. Indication of how closely the reference is followed.

Plotting

Each MPC node creates and writes to a CSV file that contains the odometry and other relevant information for plotting, along with a timer that counts the real-time since the simulation began. The plotter reads all the CSV files, takes the real-time of the first file read and finds the closest real-time in all the other files, then calculates the median value and median time. From the data collected, it was decided on five different values to plot: the x, y and z position, the angle between the ROVs, and the distance between the ROVs. The angles between the ROVs are of interest because the ROVs exchange data using optical sensors, and are therefore reliant on being in each other's field of view in order to exchange data. The distance between the ROVs is included because vehicles must remain at a certain distance from each other to avoid a collision. All plots are made using the Python packages *Pandas* [66] and *Matplotlib* [67].

7.2 Results and Findings

The following table, 7.3, conveys the percentage of failed tests. A failed test is defined as a test that violates the FOV and distance constraints at any point in the test. All tests were run with the *squared error* cost function, with the exception of the test *Circular setpoint*, which was run using the *circular setpoint* cost function.

Table 7.3: Table of the percentage of tests where constraints failed

	FOV Constraints Failed [%]				Distance Constraints Failed [%]			
Test name	Circle	Torus	Line	Spiral	Circle	Torus	Line	Spiral
Default PC 1	7.74	5.29	0.59	11.76	0.0	0.0	0.0	0.0
Default PC 2	5.83	5.83	0.97	5.34	0.0	0.0	0.0	0.0
Default PC 3	5.22	3.91	8.26	16.09	0.0	0.0	0.0	0.0
Halved added mass	6.25	7.00	23.80	9.09	0.0	0.0	0.0	0.0
Doubled added mass	4.04	6.80	1.46	17.48	0.0	0.0	0.97	0.0
Halved damping	4.41	6.31	0.48	3.40	0.0	0.0	0.0	0.0
Doubled damping	1.39	3.24	0.0	1.39	0.0	0.0	0.0	0.0
Doubled mass	0.41	0.0	0.0	0.0	0.0	0.0	0.0	0.0
70% Packet loss	3.88	4.85	0.49	1.94	0.0	0.0	0.0	0.0
0.25 m/s Current with waves	9.72	5.09	9.26	11.11	0.0	0.0	0.46	0.0
0.5 m/s Current with waves	16.96	19.13	11.4	16.67	0.0	0.0	0.44	0.0
Circular setpoint	90.95	100	14.76	92.86	15.71	22.86	7.14	15.24
Default Python simulator	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 7.4 presents the time shift from the circle test standard test, for all testing conditions.

Table 7.4: Table of time shift from the reference to the median linear positions

Time Shift in Circle Test

Test name	Time Shift [s]			
Positions	x	y	z	PC
Default PC 1	10.13	11.86	3.99	1
Default PC 2	21.69	23.86	7.66	2
Default PC 3	17.92	17.39	9.16	3
Halved added mass	12.87	14.17	3.93	1
Doubled added mass	15.76	18.28	5.54	1
Halved damping	21.78	22.10	7.05	2
Doubled damping	26.87	24.64	9.05	2
Doubled mass	33.81	27.30	10.67	3
0.25 m/s Current with waves	20.57	25.86	9.52	3
0.5 m/s Current with waves	20.40	20.50	8.75	3
70% Packet loss	20.40	20.50	8.75	2
Circular setpoint	20.58	11.16	2.30	2
Default Python simulator	19.90	1.80	1.15	N/A

Figures

This section presents selected figures, that are used to give an overview of the results.
A complete collection of all the figures from the Gazebo test results is found in Appendix G.
A complete collection of all the figures from the Python test results is found in Appendix F.

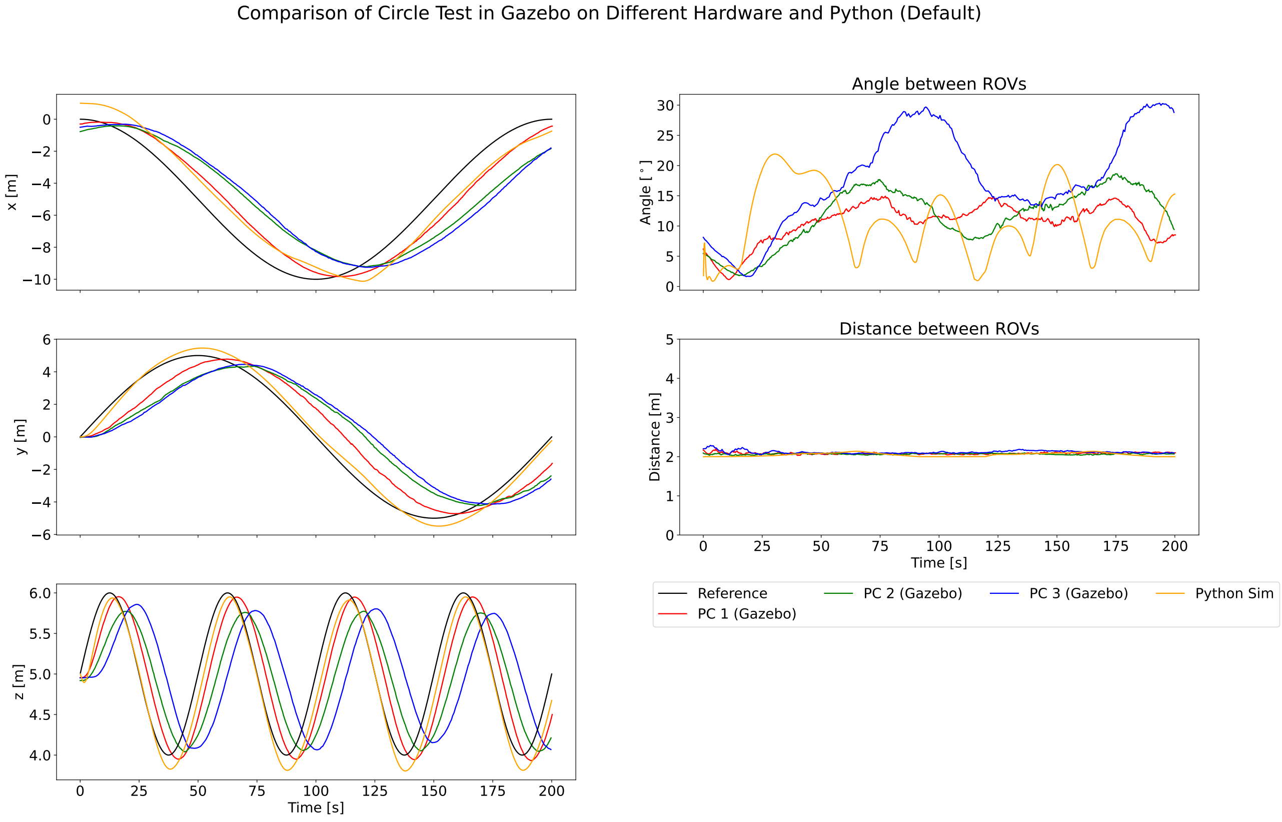


Figure 7.6: Comparison between Gazebo and Python (median)

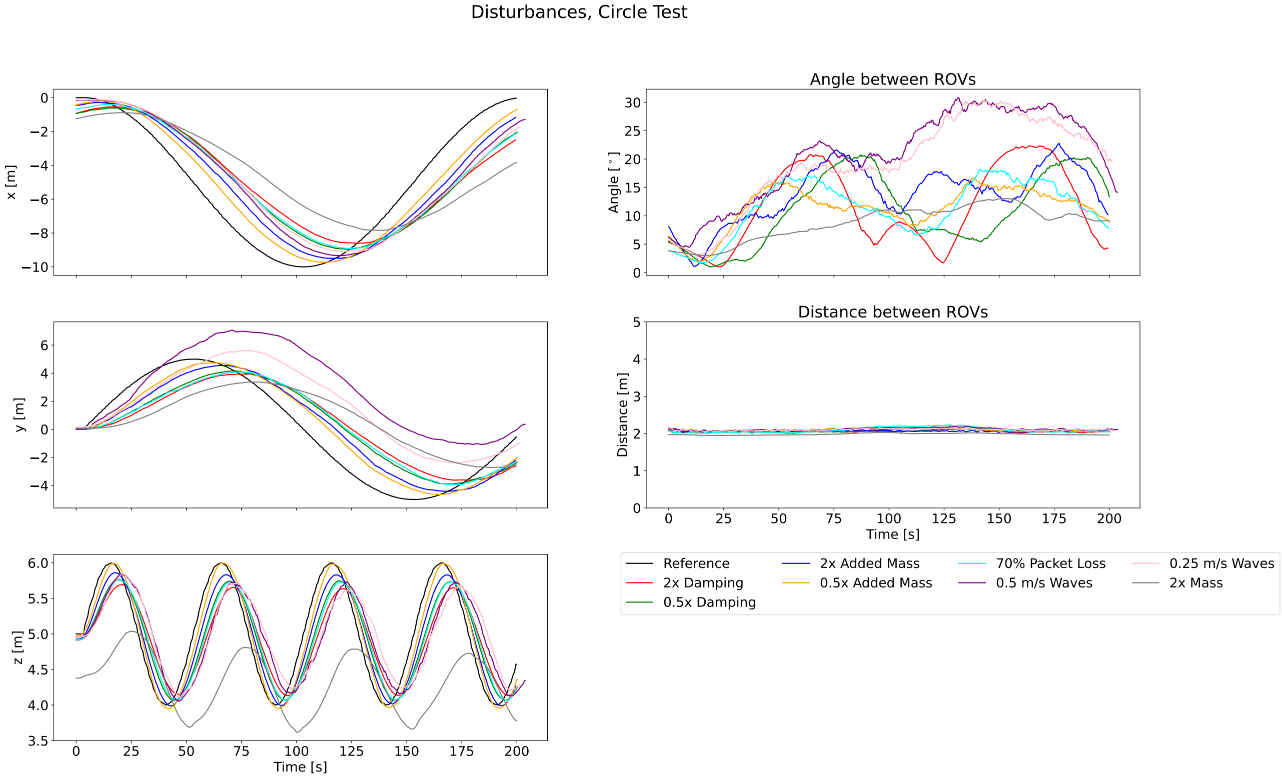


Figure 7.7: Showcase of disturbances, circle test

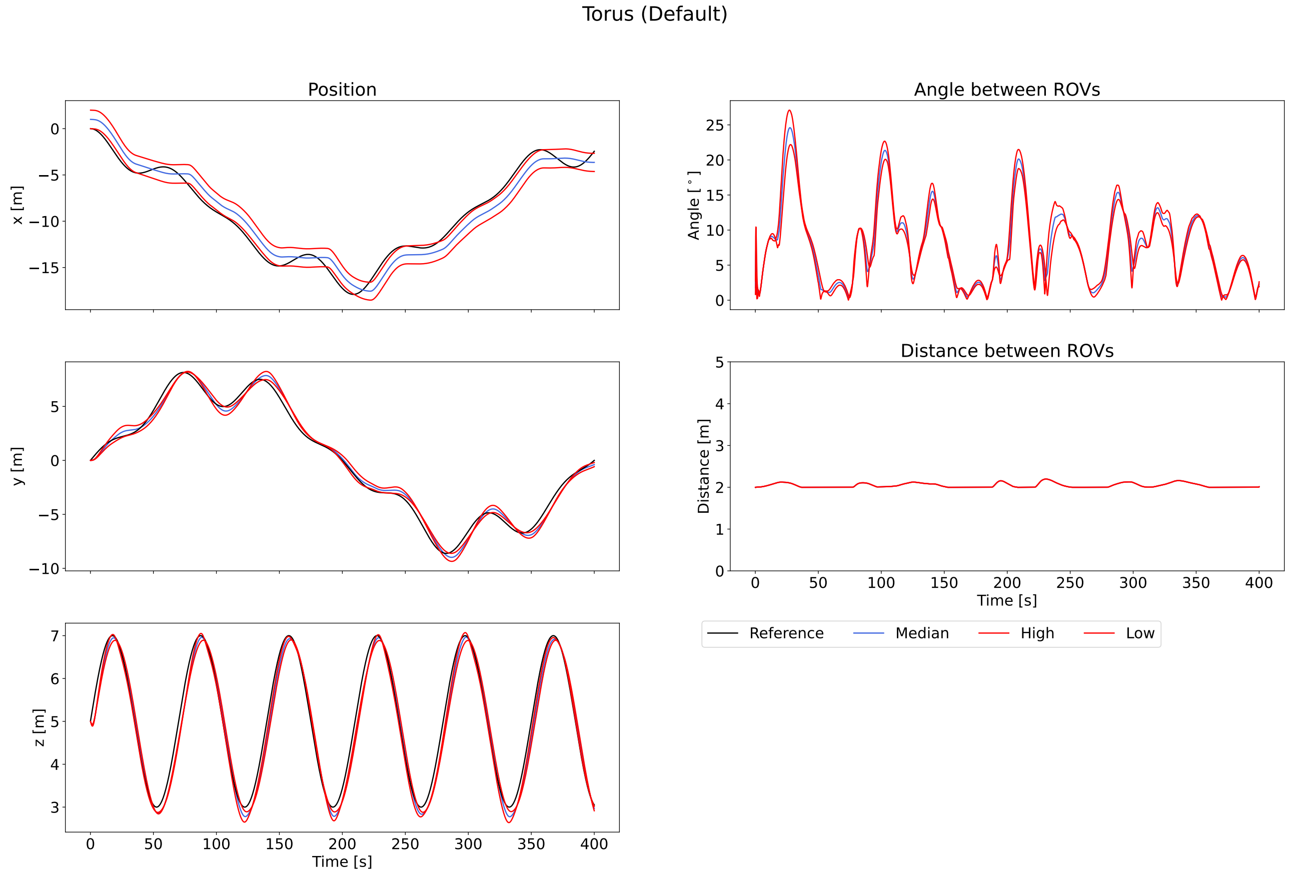


Figure 7.8: Torus test Python simulator

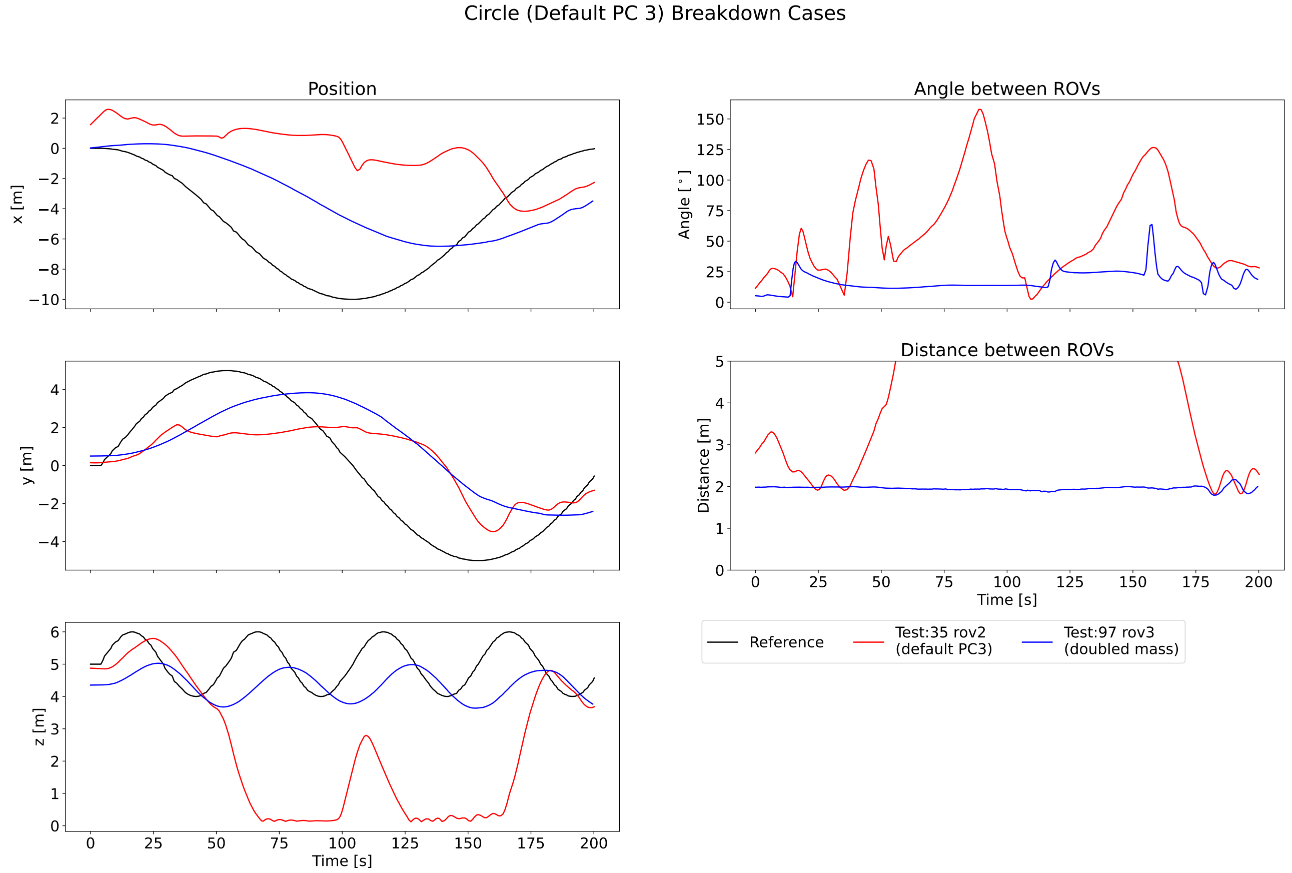


Figure 7.9: Breakdown case and failed test

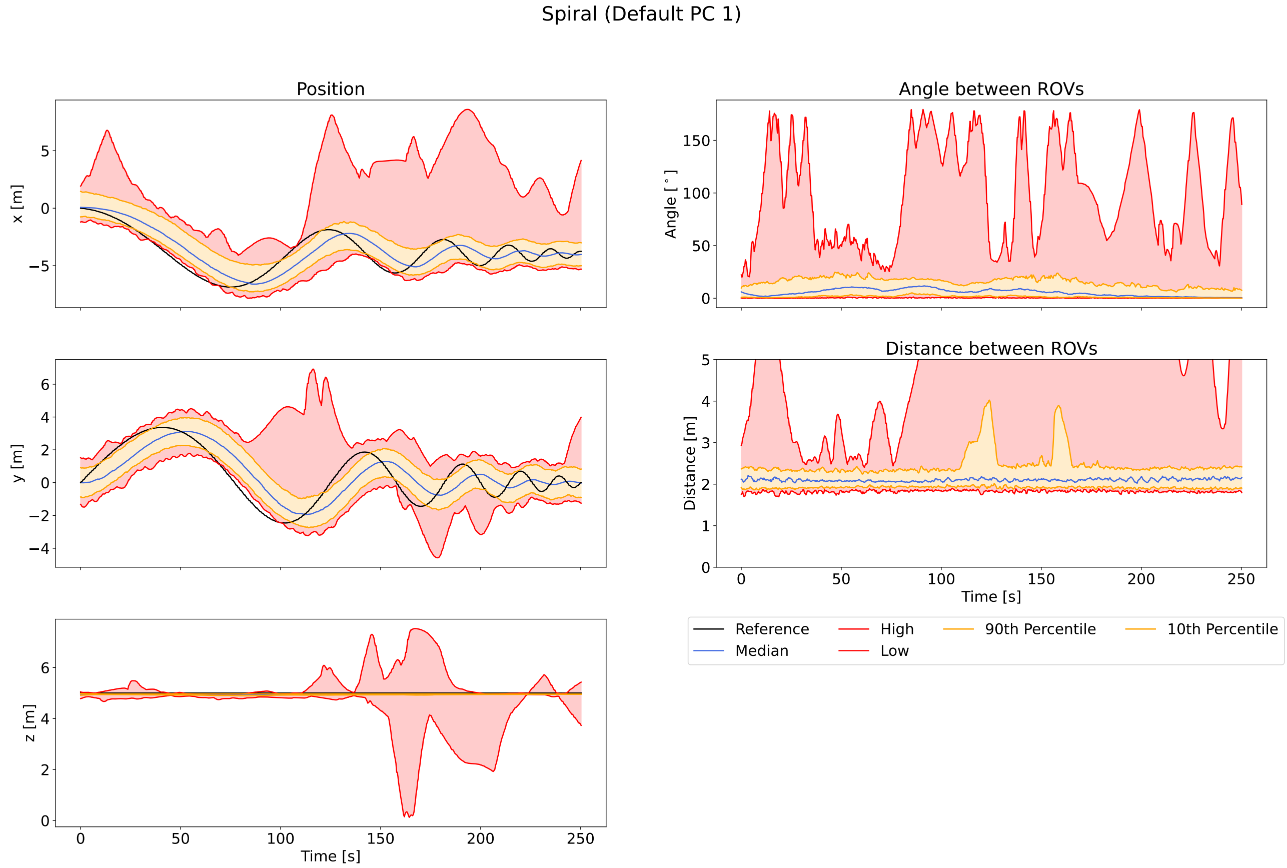


Figure 7.10: Spiral test under default conditions

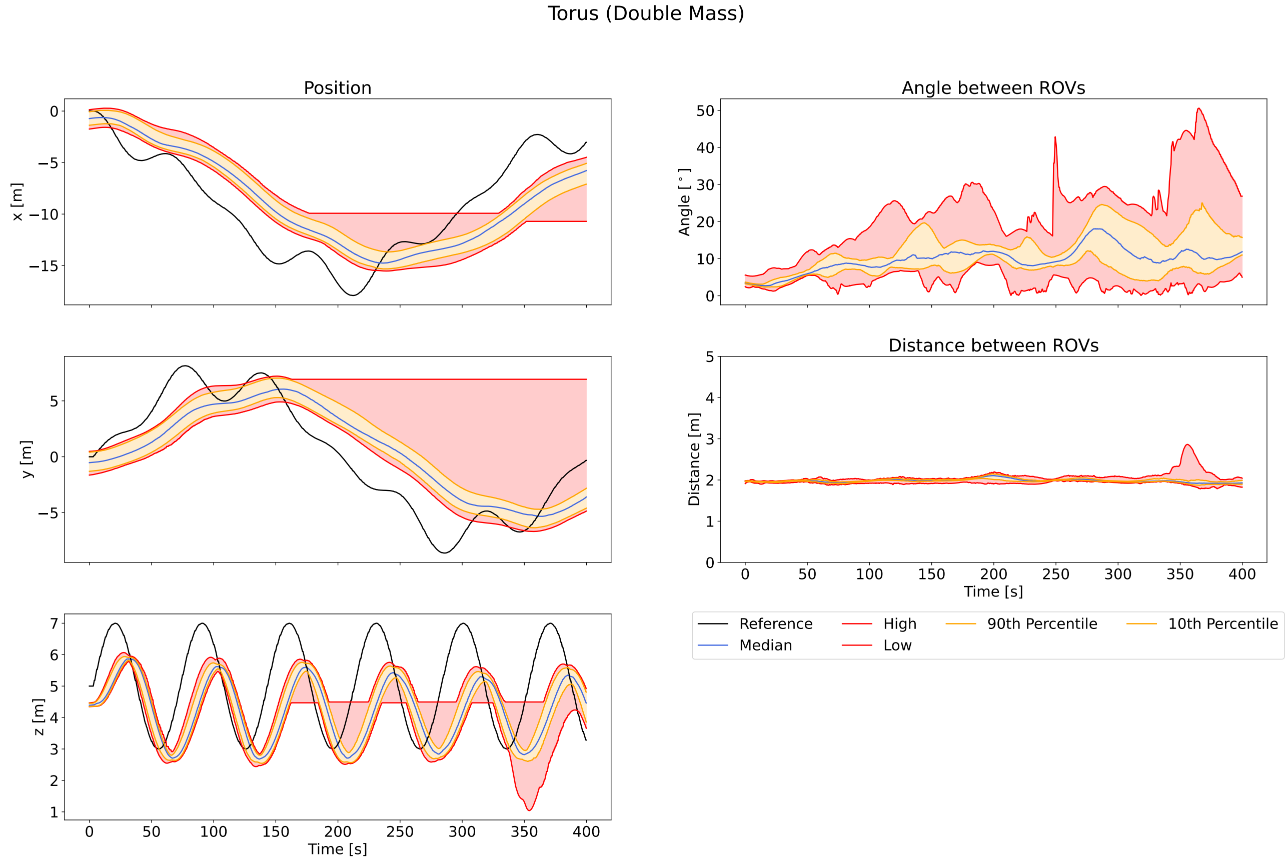


Figure 7.11: Torus test with doubled mass

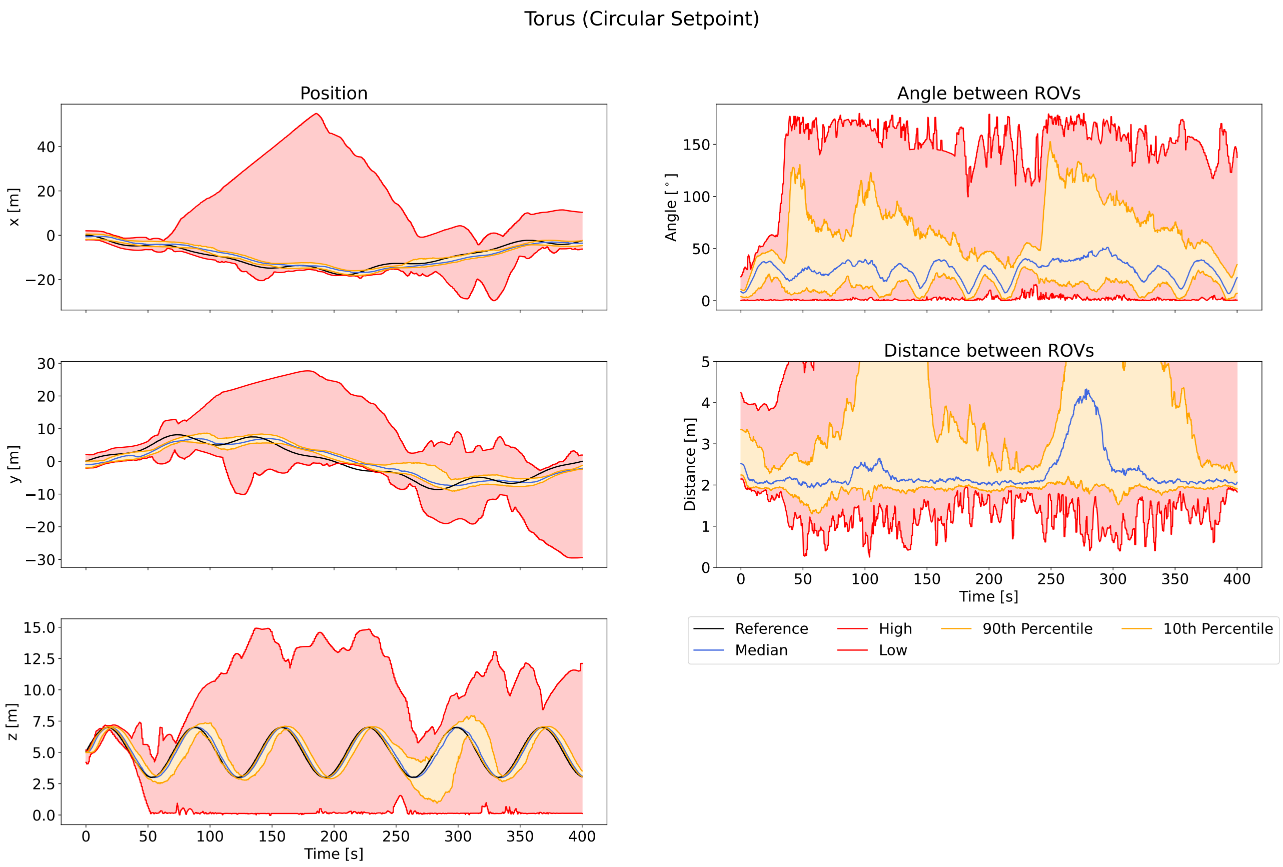


Figure 7.12: Torus test with circular setpoint

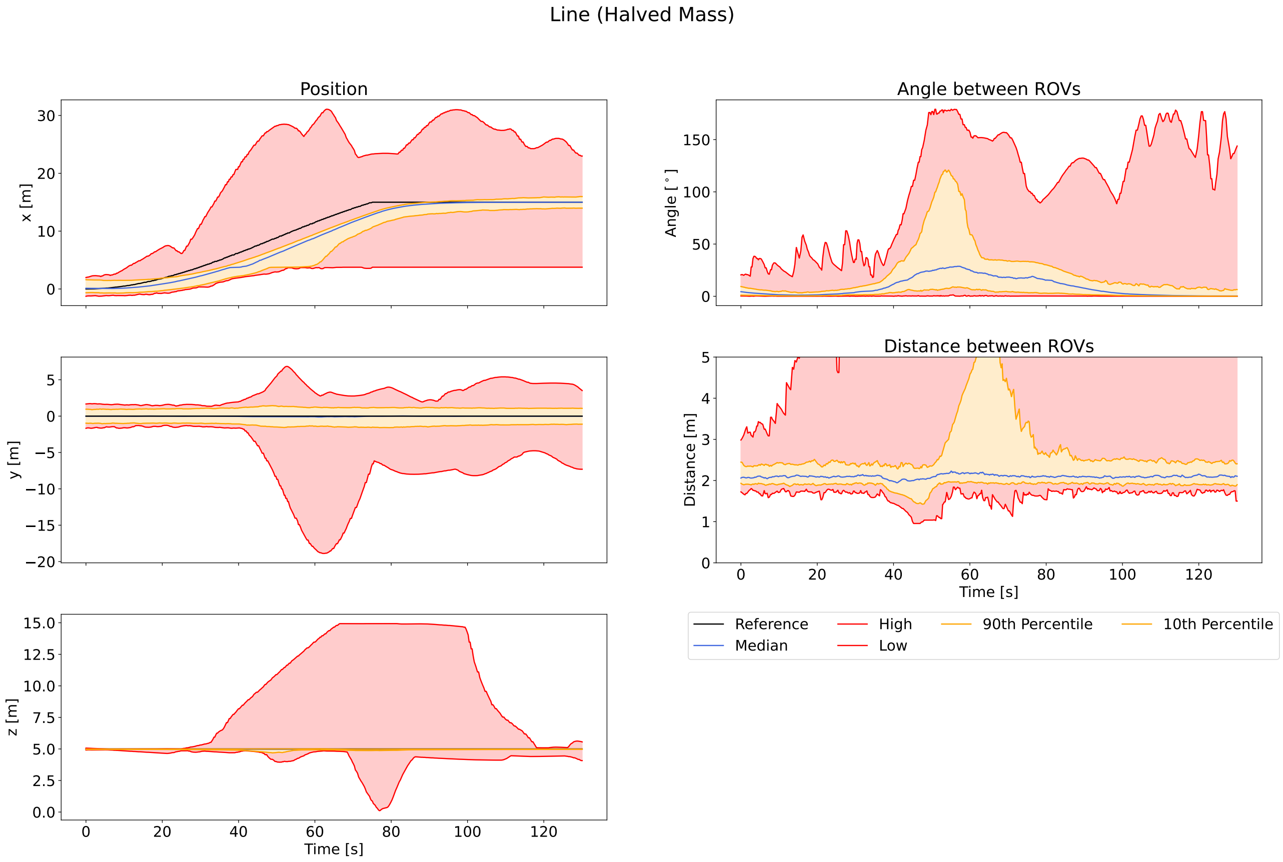


Figure 7.13: Line test with halved added mass

7.3 Discussion

7.3.1 Python Simulator

When creating the initial version of the controller, the Python simulator was an important tool as the ideal model removed discrepancies between the model implemented in the controller and the actual system model, which otherwise could have been a source of error. The simulator also enabled an indication of system performance by observing the calculation time of the controller at each cycle. If modifying the controller severely prolonged the calculation time, it likely worsened the performance of the system.

Since the model in the Python simulator is parameterised using Euler angles as opposed to quaternions, which is implemented in the Gazebo simulator, there can be a slight difference in the performance. However, the dynamics in the model should remain the same.

7.3.2 Gazebo Simulator

The control system designed in this thesis was designed and implemented for both single-agent and multi-agent systems. The controller, the GUI and Gazebo can be launched with anywhere from one to three agents by modifying the parameter for the number of agents in the ROS 2 files. A specific cost function was also assigned depending on the number of agents launched, which meant that no modifications had to be done in the code.

The system can be scaled up to N agents with few modifications in the code.

It was decided to focus on the two-agent case in the thesis, as opposed to the three-agent case, as hardware limitations led to slower simulation speed, which worsened by adding more agents.

A parameter in the ROS 2 files which disabled the FOV constraint was also implemented. If the FOV constraint was disabled the ROVs would not have any constraints in the attitude but would have a desired attitude from the cost function, where they were to be directed towards the desired point. This removed constraints on the system, which can enhance performance in collision avoidance and path following.

Using Gazebo for Simulations

Using Gazebo for the simulations offered the opportunity to run tests of the control system with realistic physics and robot dynamics. As field testing with the ROVs was not possible in this thesis, Gazebo gave the opportunity to run simulations on a more realistic system than the Python simulator.

An important aspect to note regarding BlueROV2 Garden, which is the workspace used with Gazebo, is that the model for the BlueROV2 Heavy might deviate from the dynamics of the actual ROV. This means that the performance of the controller could differ if used for field testing. As system identification was not a part of this thesis, no tests were done to analyse the accuracy of the model, and the model was assumed to be an accurate representation.

Computer Hardware's Impact on Results

The Gazebo simulator runs the simulations at or below real-time speed, so simulating many iterations was time-consuming. In order to accelerate this process, simulations were run in parallel across several computers. This introduces a new source of error because the speed at which Gazebo simulates is dependent on the hardware of the computers. The trajectory publisher does not account for differing simulation speeds across the computers but instead publishes at a set rate. The effect of this is that the trajectory will move faster relative to the speed of the simulation, and will change the testing conditions in the different computers used. The effect of this can be confirmed in Table 7.4, where it can be observed that the time shift value differs between the different computers, with PC 1 having the lowest time shift value. When comparing the average time shift value in the x-, y-, and z-coordinates, PC 2 had a 102.43% increase, and PC 3 had an 84.37% increase, compared to PC 1.

One way to mitigate this would be to have the publishing frequency, or the step size of the trajectory node modified by the real-time factor $\left(\frac{t_{sim}}{t_{real}}\right)$. The real-time factor oscillates at a very high frequency, so a viable solution would likely be to take the average real-time factor over a period and multiply the publishing frequency (or the trajectory step size) with this factor.

If all the test was being run on the same computer the impact of hardware in the tests would have been removed. This would have made the tests more directly comparable.

7.3.3 Robustness of Controller

The failure rates from the tests are displayed in Table 7.3, where it can be observed that the distancing constraint had a less than 1% failure rate in all tests using the squared error, which proves some robustness in the collision avoidance objective.

Using the default parameters in the system model, the deviation in position was small, as exemplified in Figure 7.10. This test has a Failure rate of over 11%, seen in Table 7.3, but still follows the trajectory of the path closely. This can also be observed in the other tests, seen in Appendix G.

The objective of communication between the ROVs did however have a considerable failure rate. A frequent occurrence was controller breakdowns, where the constraints and deviation between desired and actual position, were violated by a large margin. As can be seen in Appendix G, with considerable peaks in the range outside the 10th to the 90th percentile.

The tests with the circular setpoint solution had a failure rate close to 100% in the standard tests and thus did not achieve robustness.

Controller Parameters

The parameters for the controller were decided through an experimental method. As there were a lot of parameters in the controller which could be adjusted, the parameters selected for testing might not be the parameters that ensure the best-performing system. This is a source of error, as continued experiments would have led to changes in performance, which could have led to improvements in the performance of the controller.

Quantifying Path Following

Using time shift as a measurement for path following can lead to misleading results if there is another external force acting upon the ROV, such as ocean currents. This is because the time shift does not take into account the distance from the reference, but is rather an estimate of how quickly the ROV reacts to changes in the reference. External forces like ocean currents can lead to a static error.

Python and Gazebo Comparison

The Python simulator managed to fulfil the main objectives set, as exemplified in 7.8. When running the four standard tests, it did not violate the FOV constraint or the distancing constraint set at any point, as seen in Table 7.3. The time shift was lower compared to the tests performed in the Gazebo simulator, as seen in Table 7.4, where the average time shift value of the x-, y-, and z-coordinates are 19.85% lower compared to PC 1, which is the PC with the lowest time shift in the test.

The median values of the default tests run in Gazebo had similar performance as the default test run in Python, as can be observed in Figure 7.6, where the average FOV angle and distance between ROVs are all within the values of the soft constraint. The main difference is the time shift, where the Python simulator performs better than Gazebo. This could also be observed for the other standard tests, which can be seen in Appendix I

The standard test was only run once with the Python simulator, which has an impact on the result. To fully test the robustness of the Python simulator, the test could have been run a multitude of times, with varying initial conditions.

Effect of Altering Test Conditions

The median values from the tests with the altered test conditions are plotted in Figure 7.7, where it can be observed that the median FOV angle for all tests is lower than the maximum angle set for the FOV constraint, and the median distance between the ROVs is higher than the minimum distance. This is also the case for all of the other standard tests, which can be found in Appendix H.

Added Mass Modifying the added mass parameters had a significant impact on the performance of the controller. From Table 7.3, an increase of line tests that fail to uphold the constraints can be computed to 3893% when the added mass is halved. Inspecting Figure 7.13 confirms that halving this parameter drastically affected the robustness of the controller, with the 90th percentile in the angle between the ROVs peaking above 100°. It can also be observed, from Figures 7.7 and 7.6 that the peaks are lower when the added mass is doubled. This can imply that the MPC is “stopping short” by overestimating the added mass that is applied to the ROVs from inertia in the surrounding water, compared to the internal model of the Gazebo simulator.

Damping The impact of modifying the damping parameters in the model is limited. From Table 7.3 an improvement in all tests can be observed when doubling the damping. Similarly, all but one standard test see an improvement when halving the damping, but these values can be considered to be negligible. From Table 7.4 an increase in the time shift can be seen, with an average increase across the x-, y- and z-positions of 15.1% when doubling the damping.

Doubled Mass Doubling the mass in the controller improved the performance in terms of the communication and collision avoidance between the ROVs, having a close to 0% failure rate in the tests. However, the test was performed on PC 3, and from Table 7.4 the average time shift for the x-, y- and z-coordinate is 54.1% higher for the test with doubled mass compared to the test under default conditions.

In Figure 7.11, it can also be observed that the oscillations along the z-axis are centred around $z \approx 4$ m, 1 m lower than the reference signal. This is due to the fictitious mass in the controller model, which causes large deviations in the controller's $g(\eta)$ compared to the simulator's. The controller is wrongly predicting how much force is required to keep its current altitude. This in turn also causes the ROVs to fall further from the reference trajectory, and the nuances in the trajectory are not as closely traced by the ROVs.

Packet Loss The packet loss tests improved the results from the tests when compared to the default test on the computer that was used (PC 2). The percentage of failed tests in terms of communication, and collision avoidance was lower in all the standard tests, and from Table 7.3, the time shift was an average of 1.93% smaller for the x-, y- and z-axis.

The improvement in performance was unexpected, as less frequent data exchange between the ROVs was thought to lead to worsened results. A theory as to why these results did occur is that the less frequent changes in the time-varying parameters, x_2, y_2 and z_2, led to better initial guesses for the optimiser, which led to better results. The MPC could not predict future changes in the aforementioned time-varying parameters, which could have made them a problem for the optimiser.

The result however did prove that packet loss had little impact on the robustness of the system.

Ocean Current with Waves Introducing ocean currents and waves as disturbances worsened the performance in terms of communication and collision avoidance. When running tests with 0.25 m/s current, the ROVs violated the constraints more often for the circle test, the torus test and the line test, and when running tests with 0.5 m/s current, the ROVs violated the constraints more often for all tests, as seen in Table 7.3. The time shift also worsened in both tests compared to the default test ran on the same computer (PC 3). Using the result from Table 7.4, the average increase of the time shift over the x-, y-, and z-axis is calculated to 22.5% when running 0.25 m/s, and 9.1% when running 0.5 m/s.

When comparing the weaker current (0.25 m/s) with the stronger current (0.5 m/s), considering the communication between the ROVs, the test with the stronger current performed worse in all tests, violating the constraints twice as often. When considering collision avoidance the difference in the performance in the tests is negligible. The time shift was bigger in the tests with smaller waves, which could be attributed to an increased cost connected to the positioning of the ROVs. This is caused by the ocean current and waves creating a bigger difference Δy and Δz, as can be observed in Figure 7.7, where the median trajectories are displayed.

Circular Setpoint The circular setpoint solution failed to provide robust control in terms of communication and collision avoidance, as can be observed in Table 7.3. With close to 100% failure rate for the FOV constraint, and also a high failure rate for the distancing constraint. The time shift was lower when using the circular setpoint, with a decrease of 42.8% when compared to the default test on the same computer that ran the circular setpoint test (PC 2).

Using the circular setpoint solution in the cost function made the positional term in the xy-plane into a quartic function, as described in Section 5.2.1. This caused a more rapid increase in cost, compared to the squared error as the position in the xy-plane deviates from the desired position. This rapidly increasing cost makes the positional task in the controller prioritised, which leads to the time shift being considerably lower when using the circular setpoint solution.

The rapidly increasing cost caused by the positional error seemed to undermine the FOV constraint and the distancing constraint, as the cost of these soft constraints could become comparatively low, this probably was the main cause of the lack of robustness in the communication. The effect of this could be observed in Figure 7.12, where the 90th percentile of the values of the positioning only slightly deviated from the desired position, while the 90th percentile of values for the FOV angle broke the hard constraint set. This is likely the reason for the high failure rate in collision avoidance as well. As the FOV hard constraint is broken, the controller breaks down and is unable to find converging solutions, which leads to potentially random solutions, as will be described in the following Section *Failed Tests and Breakdowns*.

The aforementioned problems with the circular setpoint solution could be attempted fixed by turning the attitude control, FOV- and distancing soft constraints into quartic functions. This could make the cost more comparable to the positional cost. Doing this could however lead to a less stable system.

Failed Tests and Breakdowns

The graphs from the tests in appendix G show the median value, high, low, 90th, and 10th percentiles from the different tests. A common recurrence in the results is controller breakdowns, which can be identified by the large gap between the 90th and 10th percentiles, and the high and low values. An example of a controller breakdown is visible in 7.12, where the highs in the positioning are remarkably larger than the limits for the 90th percentile.

Cases of a breakdown of the controller differ from other failed tests by them being unable to rapidly converge back to desired values for positioning, distancing and the FOV angle. This difference is displayed in Figure 7.9 where *Test:35 rov2 (default PC3)* is a controller breakdown case, and *Test:97 rov3 (doubled mass)* is a failed test. From the figure, it can be observed that the FOV angle falls lower than the limit of the hard constant (60°), but still is not able to converge. This is why it is considered a controller breakdown, as it clearly is within an ideal range for the FOV- and distancing constraint between time 100-125 s, but can not control the ROV to remain within these limits.

There could be several underlying problems that can cause these problems with controller breakdowns, for example:

- **The MPCs' inability to find an optimal input.**
  In the MPC, the maximum number of iterations was set to 25 to prevent unreasonably long calculation times. The disadvantage of this approach is that the optimiser might not be able to find the optimal system input and as such, can result in system behaviour that is suboptimal, particularly if the controller is unable to find an optimal solution several times in a row. As the optimiser in do-mpc relies on initial guesses for the solver for fast optimising, having bad previous optimising solutions can lead to poor initial guesses, which can lead to slow optimising. The combination of having poor initial guesses, and a maximum iteration of 25 for the solver could probably lead to controller breakdowns.

A potential countermeasure could be to remove the hard constraint for the FOV constraint. This constraint is a non-linear function, which is harder to optimise, and the optimiser can struggle to find converging solutions back to the feasible range.

- **Problem with the attitude control on the model.**
  As no tests or analysis was undertaken to test the accuracy of the system model used for the MPC, and that system modelling was not the main concern in BlueROV2 Garden, which is the workspace used with Gazebo. The model used could have some problems with the attitude dynamics. The fact that the Python simulator wouldn't work when using the quaternion model can be symptomatic of underlying problems. This should be explored if this model were to be used in further work.

Chapter 8

Conclusion

This thesis set out to design and implement a decentralized MPC in a set of ROVs. The original goal was to implement the solution the physical BlueROV2 Heavys. However, due to their involvement in the broader research project, the ROVs were not finalised, and thus unavailable. The results were, as a consequence, based on simulations. The results from repeated testing under various conditions showed that the system can handle disturbances, as well as modifications to the system parameters. The distancing constraint designed for collision avoidance worked as intended, with a failure rate of less than 1% when using the squared error cost function. In terms of the FOV constraint issues regarding robustness did arise, with a failure rate of up to a maximum of 23.8% when running tests with disturbances.

The proposed circular setpoint solution proved good path-following capabilities but struggled with consistently upholding the FOV constraint. Despite the challenge with robustness, this implementation has the potential to perform more complex positioning tasks and thus has more potential than the squared error solution.

However, system breakdowns, where one or more of the constraints were violated, did occur and thus complete robustness was not achieved. The likely source of this error is that the optimiser, at times, struggled to find a converging solution within the maximum number of iterations allotted. This results in the MPC applying a non-optimal input to the thrusters.

One weakness in the test results is that the results were sensitive to the hardware of the computer. To account for this, tests were run with default conditions on the three available computers and comparisons between tests were only made with tests run on the same computer.

In conclusion, despite challenges with consistently upholding the FOV constraint, the specifications set in the problem statement are met. Thus, the contributions from this thesis provide a basis for future implementation into the physical system.

8.1 Further Work

Physical Implementation

The natural next step is to implement the system to the BlueROV2 Heavy in preparation to perform physical testing. Physical testing will allow the test of the potential of the current implementation by seeing how it other handles real-world dynamics and disturbances. Achieving this also opens for performing system identification, such that accurate system parameters can be estimated.

The controller struggled with upholding the FOV constraint, therefore, initial testing should be done with wired communication, as a fail-safe.

Replacing do-mpc

Moving away from do-mpc can be advantageous as it struggled with performance, and could become a problem in time-critical applications. An alternative is to use Matlab's MPC-toolbox, this would allow for rapid prototyping of the MPC in Matlab, and then use the C++ code generation feature to implement it to ROS 2.

Tuning the Circular Setpoint

The circular setpoint showed that despite issues with the robustness, proved to have potential, and is likely the path forward as it has the potential to support more complex manoeuvres, like reducing the radius of the circle to simulate picking up an object with the grippers and transporting it. Tuning the parameters, and implementing attitude control as a quartic function as described in Section 5.4.1 is something that should be attempted in the future.

Troubleshooting Controller Breakdowns

To attempt to solve the problems with controller breakdowns, reproducing the scenarios when breakdowns have occurred should be attempted. If the breakdowns are reproducible, analysis of the system at the moment of breakdown is a meaningful way of troubleshooting. The initial conditions in tests when a breakdown has occurred can be found in the CSV files [3] of tests with breakdowns in control, as with the test *Test:35 rov2 (default PC3)*, displayed in Figure 7.12.

Bibliography

[1] NOAA, *How much of the ocean have we explored?* 2023. [Online]. Available: https://oceanservice.noaa.gov/facts/exploration.html, (Accessed: 09/05/2023).

[2] R. D. Christ and R. L. Wernli Sr., *The ROV Manual, A User Guide for Remotely Operated Vihicles*, 2nd ed. Oxford, UK: Butterworth-Heinemann, 2014.

[3] L. R. Fosso, P. K. Kjærem, T. H. Staurnes and K. A. Johannesen, *Decentralized model predictive control for increased autonomy in fleets of rovs*, 2023. [Online]. Available: https://github.com/lrfosso/TowardsUnderwaterAutonomousFleets.

[4] M. A. Medina, 'Template based underwater object detection in a simulation environment,' Tech. Rep., Dec. 2022.

[5] E. Rowe, S. Hustad, P. Ø. Juliebø and E. O. Almenningen, 'Implementation of a quaternion-based PD controller in ROS2 for a generic underwater vehicle with six degrees of freedom,' B.S. thesis, NTNU, 2022. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3002446.

[6] A. D. Bowen, M. V. Jakuba, N. E. Farr, J. Ware, C. Taylor, D. Gomez-Ibanez, C. R. Machado and C. Pontbriand, 'An un-tethered rov for routine access and intervention in the deep sea,' in *Proc. 2013 OCEANS - San Diego*, San Diego, CA, USA: IEEE, Sep. 2013, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6741383.

[7] G. Cossu, A. Sturniolo, A. Messa, S. Grechi, D. Costa, A. Bartolini, D. Scaradozzi, A. Caiti and E. Ciaramella, 'Sea-trial of optical ethernet modems for underwater wireless communications,' *Journal of Lightwave Technology*, vol. 36, no. 23, pp. 5371–5380, 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8468031.

[8] T. Inoue, T. Shiosawa and K. Takagi, 'Dynamic analysis of motion of crawler-type remotely operated vehicles,' *IEEE Journal of Oceanic Engineering*, no. 2, pp. 375–382, 2013. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6425431.

[9] BlueRobotics, *BlueROV2*, 2023. [Online]. Available: https://bluerobotics.com/store/rov/bluerov2/, (Accessed: 11/04/2023).

[10] R. Jehangir, *Bluerobotics bluerov2 heavy*, [Screen grab of 3D model], 2018. [Online]. Available: https://grabcad.com/library/bluerobotics-bluerov2-heavy-1, (Accessed: 12/04/2023).

[11] BlueRobotics, *BlueROV2 Heavy Configuration Retrofit Kit*, 2023. [Online]. Available: https://bluerobotics.com/store/rov/bluerov2-upgrade-kits/brov2-heavy-retrofit/, (Accessed: 11/04/2023).

[12] BlueRobotics, *BlueROV2 datasheet*. 2023, Revision 03/22. [Online]. Available: https://www.dropbox.com/s/0j0puotbfs7dj9s/br_bluerov2_datasheet_rev2022-R4ROV.pdf?dl=0, (Accessed: 14/04/2023).

[13] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*, 2nd ed. Chichester, UK: John Wiley & Sons Inc, 2021, ISBN: 9781119575054.

[14] C.-J. Wu, '6-dof modelling and control of a remotely operated vehicle,' M.S. thesis, Flinders University, 2018. [Online]. Available: `https://theses.flinders.edu.au/view/27aa0064-9de2-441c-8a17-655405d5fc2e/1`.

[15] M. W. Spong, S. Hutchinson and M. Vidyasagar, *Robot Modeling and Control*, 2nd ed. Chichester, UK: John Wiley & Sons Inc, 2020, ISBN: 9781119523994.

[16] R. Mukundan, *Quaternions: From classical mechanics to computer graphics, and beyond*. New Zealand: University of Canterbury, 2002. [Online]. Available: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=64e152a06f51f2d729e803c3e8c88f497a51a4aa`, (Accessed: 09/05/2023).

[17] J. Chou, 'Quaternion kinematic and dynamic differential equations,' *IEEE Transactions on Robotics and Automation*, vol. 8, no. 1, pp. 53–64, 1992. DOI: `10.1109/70.127239`.

[18] O.-E. Fjellstad and T. I. Fossen, 'Quaternion feedback regulation of underwater vehicles,' in *Proc. 1994 Proceedings of IEEE International Conference on Control and Applications*, vol. 2, Glascow, UK: IEEE, Aug. 1994, pp. 857–862. DOI: `10.1109/CCA.1994.381209`.

[19] O. Calvo, A. Rozenfeld, A. Souza, F. Valenciaga, P. F. Puleston and G. G. Acosta, 'Experimental results on smooth path tracking with application to pipe surveying on inexpensive auv,' in *Proc. 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, [Photo], Nice, France: IEEE, Sep. 2008, pp. 3647–3653. DOI: `10.1109/IROS.2008.4650966`.

[20] S. S. Sandøy, 'System identification and state estimation for rov udrone,' M.S. thesis, NTNU, 2016. [Online]. Available: `https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2409503`.

[21] The MathWorks, Inc., *What is model predictive control?* 2023. [Online]. Available: `https://se.mathworks.com/help/mpc/gs/what-is-mpc.html`, (Accessed: 03/04/2023).

[22] K. Hauser, *Section iv. dynamics and control, chapter 17. optimal control*, 2020. [Online]. Available: `https://motion.cs.illinois.edu/RoboticSystems/OptimalControl.html`, (Accessed: 03/04/2023).

[23] S. P. Sethi, *What Is Optimal Control Theory?* Springer International Publishing, 2021, ISBN: 978-3-030-91745-6.

[24] E. F. Camacho, *Model Predictive Control*, 2nd ed. 2007. London: Springer London : Imprint: Springer, 2007, ISBN: 978-0-85729-398-5.

[25] S. Boyd and L. Vandenberghe, *Convex optimization*. UK: Cambridge University Press, 2004.

[26] S. Lucia and F. Fiedler, *Model predictive control python toolbox*, 2023. [Online]. Available: `https://www.do-mpc.com/en/latest`, (Accessed: 03/04/2023).

[27] A. Bürger, C. Zeile, A. Altmann-Dieses, S. Sager and M. Diehl, 'Design, implementation and simulation of an mpc algorithm for switched nonlinear systems under combinatorial constraints,' *Journal of Process Control*, vol. 81, pp. 15–30, 2019. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0959152419303592`.

[28] D. E. Seborg, T. F. Edgar, D. A. Mellichamp and F. J. Doyle III, *Process Dynamics and Control*, 3rd ed. UK: John Wiley & Sons Inc, 2011.

[29] S. Lucia, A. Tătulea-Codrean, C. Schoppmeyer and S. Engell, 'Rapid development of modular and sustainable nonlinear model predictive control solutions,' *Control Engineering Practice*, vol. 60, pp. 51–62, 2017. DOI: `10.1016/j.conengprac.2016.12.009`.

[30] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings and M. Diehl, 'CasADi – A software framework for nonlinear optimization and optimal control,' *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019. DOI: `10.1007/s12532-018-0139-4`.

[31] R. Abdulghafor, S. Abdullah, S. Turaev and M. Othman, 'An overview of the consensus problem in the control of multi-agent systems,' vol. 59, Apr. 2018. DOI: `10.1080/00051144.2018.1492688`.

[32] S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall, 'Robot Operating System 2: Design, architecture, and uses in the wild,' *Science Robotics*, vol. 7, no. 66, May 2022. DOI: `10.1126/scirobotics.abm6074`, (Accessed: 09/05/2023).

[33] Open Robotics, *Ros2 documentation*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/index.html`, (Accessed: 10/04/2023).

[34] Canonical Ltd, *Ubuntu 22.04.2 lts (jammy jellyfish)*, 2023. [Online]. Available: `https://releases.ubuntu.com/jammy/`, (Accessed: 10/04/2023).

[35] Open Robotics, *Getting started with gazebo?* 2023. [Online]. Available: `https://gazebosim.org/docs/garden`, (Accessed: 28/04/2023).

[36] Python Software Foundation, *Python 3.10.0 documentation*, 2023. [Online]. Available: `https://docs.python.org/release/3.10.0/`, (Accessed: 28/04/2023).

[37] S. Lucia and F. Fiedler, *Installation*, 2023. [Online]. Available: `https://www.do-mpc.com/en/latest/installation.html`, (Accessed: 07/05/2023).

[38] Microsoft, *Microsoft c++, c, and assembler documentation*, 2023. [Online]. Available: `https://learn.microsoft.com/en-us/cpp/?view=msvc-170`, (Accessed: 13/05/2023).

[39] D. Thomas, *Colcon - collective construction*, 2023. [Online]. Available: `https://colcon.readthedocs.io/en/released/index.html`, (Accessed: 11/04/2023).

[40] Fraunhofer IPA, *Ros 2 file system*, 2023. [Online]. Available: `https://ros2-industrial-workshop.readthedocs.io/en/latest/_source/basics/ROS2-Filesystem.html`, (Accessed: 12/04/2023).

[41] Open Robotics, *Creating a package*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html`, (Accessed: 11/04/2023).

[42] Open Robotics., *Developing a ros 2 package*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/How-To-Guides/Developing-a-ROS-2-Package.html`, (Accessed: 15/05/2023).

[43] Open Robotics, *Nodes*, 2018. [Online]. Available: `https://wiki.ros.org/Nodes`, (Last edited: 12/04/2018 20:54:54).

[44] Open Robotics, *Understanding nodes*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html`, (Accessed: 12/04/2023).

[45] Open Robotics, *About ros 2 client libraries*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Concepts/About-ROS-2-Client-Libraries.html`, (Accessed: 28/04/2023).

[46] Open Robotics, *Rclpy*, 2023. [Online]. Available: `https://docs.ros2.org/foxy/api/rclpy/index.html`, (Accessed: 28/04/2023).

[47] Open Robotics, *Welcome to the documentation for rclcpp*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/p/rclcpp/`, (Accessed: 28/04/2023).

[48] Python Software Foundation, *9. classes*, 2023. [Online]. Available: `https://docs.python.org/3/tutorial/classes.html`, (Accessed: 28/04/2023).

[49] cplusplus.com, *Classes (I)*, 2023. [Online]. Available: `https://cplusplus.com/doc/tutorial/classes/`, (Accessed: 28/04/2023).

[50] Open Robotics, *Using python, xml, and yaml for ros 2 launch files*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/How-To-Guides/Launch-file-different-formats.html`, (Accessed: 12/05/2023).

[51] Open Robotics., *Understanding parameters*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html#ros2-param-set`, (Accessed: 12/05/2023).

[52] Open Robotics, *Creating a launch file*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Creating-Launch-Files.html`, (Accessed: 05/05/2023).

[53] Open Robotics, *Using parameters in a class (python)*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Using-Parameters-In-A-Class-Python.html`, (Accessed: 05/05/2023).

[54] Open Robotics, *std_msgs Message Documentation*, 2021. [Online]. Available: `https://docs.ros2.org/galactic/api/std_msgs/index-msg.html`, (Accessed: 05/05/2023, Autogenerated on May 19 2021 01:52:13).

[55] Open Robotics, *Writing a simple publisher and subscriber (python)*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html`, (Accessed: 28/04/2023).

[56] Open Robotics., *Understanding services*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html`, (Accessed: 12/04/2023).

[57] Open Robotics., *Understanding actions*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html`, (Accessed: 12/04/2023).

[58] Open Robotics, *How it works*, 2023. [Online]. Available: `https://gazebosim.org/home`, (Accessed: 04/05/2023).

[59] Open Robotics, *Features and benefits*, 2023. [Online]. Available: `https://gazebosim.org/feature`, (Accessed: 04/05/2023).

[60] Y. Chen, 'High-order polynomial interpolation based on the interpolation center's neighborhood the amendment to the runge phenomenon,' in *Proc. 2009 WRI World Congress on Software Engineering*, vol. 2, Xiamen, China: IEEE, May 2009, pp. 345–348. DOI: `10.1109/WCSE.2009.295`.

[61] Open Robotics, *Understanding topics*, 2023. [Online]. Available: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html#ros2-topic-pub`, (Accessed: 08/05/2023).

[62] D. Thomas, *Rqt*, 2016. [Online]. Available: `http://wiki.ros.org/rqt`, (Last edited: 30/08/2016 18:41:47).

[63] J. D. Hunter, 'Matplotlib: A 2d graphics environment,' *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: `10.1109/MCSE.2007.55`.

[64] PySimpleGUI, *Pysimplegui - github*, 2023. [Online]. Available: `https://github.com/PySimpleGUI/PySimpleGUI`, (Accessed: 18/05/2023).

[65] Z. Chonoles, *Do these equations create a helix wrapped into a torus?* 2013. [Online]. Available: `https://math.stackexchange.com/q/324553`, (Version: 2013-03-08).

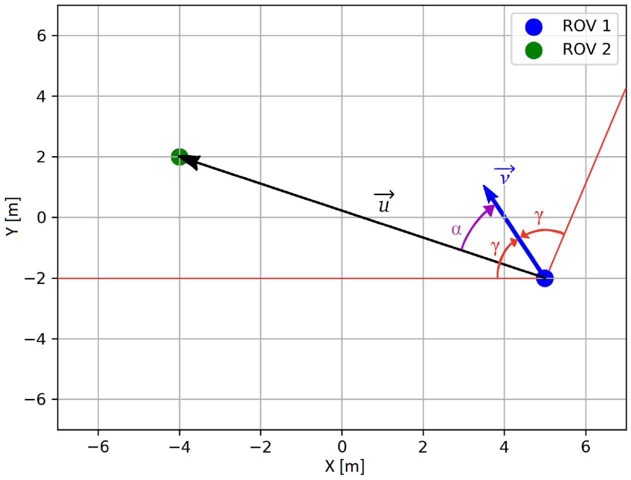
[66] Pandas, *Api reference*, 2023. [Online]. Available: `https://pandas.pydata.org/docs/reference/index.html`, (Accessed: 10/05/2023).

[67] Matplotlib, *Api reference*, 2023. [Online]. Available: `https://matplotlib.org/stable/api/index.html`, (Accessed: 10/05/2023).

Appendix

Appendix A

Derivation of FOV Formula



$$FOV\ criteria : \alpha \leq \gamma \tag{A.1}$$

If the inequality constraint A.1 is true, then the FOV constraint is fullfilled.
The next step is determining the α angle:

$$\alpha = \arccos \frac{\overrightarrow{v} \cdot \overrightarrow{u}}{||\overrightarrow{v}|| + ||\overrightarrow{u}||}, ||\overrightarrow{v}|| \triangleq 1 \tag{A.2}$$

The formula A.2 is used to determine the angle between the two vectors. $\overrightarrow{v}$ is a unit vector, and its length is by definition 1.

Inserting the formula for α, A.2, in to the FOV criteria, A.1:

$$\arccos \frac{\overrightarrow{v} \cdot \overrightarrow{u}}{||\overrightarrow{u}||} \leq \gamma \tag{A.3}$$

Multiplying A.3 with cos. This makes the equation less non-linear, which makes it easier to use when used for control.

$$\frac{\overrightarrow{v} \cdot \overrightarrow{u}}{||\overrightarrow{u}||} \geq \cos \gamma \tag{A.4}$$

Multiplying with $\left\|\overrightarrow{u}\right\|$. This is also a trick to make the equation less non-linear, and also to avoid the possibility of division by zero faults in the program.

$$nl\_con : \cos\gamma \cdot ||\overrightarrow{u}|| - \overrightarrow{v} \cdot \overrightarrow{u} \leq 0 \tag{A.5}$$

Equation A.5 is the resulting non-linear constraint in symbolic form.

$\overrightarrow{v}$: Attitude vector of the vehichle

$$\overrightarrow{v} = R \cdot \hat{i} \tag{A.6}$$

Equation A.6 is used to calculate $\overrightarrow{v}$. This is the attitude vector of the vehichle in the local surge-direction.

$$R = \begin{bmatrix} 1-2(\epsilon_2^2+\epsilon_3^2) & 2(\epsilon_1\epsilon_2-\epsilon_3\eta) & 2(\epsilon_1\epsilon_3+\epsilon_2\eta) \\ 2(\epsilon_1\epsilon_2+\epsilon_3\eta) & 1-2(\epsilon_1^2+\epsilon_3^2) & 2(\epsilon_2\epsilon_3-\epsilon_1\eta) \\ 2(\epsilon_1\epsilon_3-\epsilon_2\eta) & 2(\epsilon_2\epsilon_3+\epsilon_1\eta) & 1-2(\epsilon_1^2+\epsilon_2^2) \end{bmatrix}, \hat{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{A.7}$$

The rotation matrix R is from page 32 in T. I. Fossen's book *Handbook of Marine Craft Hydrodynamics and Motion Control*[1]. The î vector is by definition.

$$\overrightarrow{v} = \begin{bmatrix} 1-2(\epsilon_2^2+\epsilon_3^2) \\ 2(\epsilon_1\epsilon_2+\epsilon_3\eta) \\ 2(\epsilon_1\epsilon_3-\epsilon_2\eta) \end{bmatrix} \tag{A.8}$$

Multiplying the matrix with the vector in A.7, we get the resulting $\overrightarrow{v}$ in A.8.

$\overrightarrow{u}$: Vector from ROV1 to ROV2

$$\overrightarrow{u} = \begin{bmatrix} x2-x1 \\ y2-y1 \\ z2-z1 \end{bmatrix} \tag{A.9}$$

The vector in A.9 is the standard formula for a vector between two points.

$$\begin{aligned} nl\_con : \quad & \cos\gamma \cdot ||\overrightarrow{u}|| - \overrightarrow{v} \cdot \overrightarrow{u} \leq 0 \\ nl\_con : \quad & cos(\gamma) \cdot \sqrt{(x2-x1)^2+(y2-y)^2+(z2-z)^2} \\ & -((1-2\epsilon_2^2+2\epsilon_3^2)(x2-x1)+(2\epsilon_1\epsilon_2+2\epsilon_3 e\eta)(y2-y1)+(2\epsilon_1\epsilon_3-\epsilon_2\eta)(z2-z1)) \leq 0 \end{aligned} \tag{A.10}$$

In A.10 the complete equation for the non-linear constraint is written out. First in symbolic form, then in full form.

[1] T. I. Fossen, Handbook of Marine Craft Hydrodynamics and Motion Control, 2nd ed. Wiley, 2021. DOI: 10.1002/9781119994138.

Appendix B

Derivation of Formula for Attitude Control Using Quaternions

The formulas in B.1, B.2 and B.3 is based on O.-E. Fjellstad and T. I. Fossen's report *Quaternion feedback regulation of underwater vehicles*[1]

$$\tilde{q} = q_d \cdot q \tag{B.1}$$

$\tilde{q}$: difference between current attitude and desired attitude
q_d : desired attitude
q : current attitude

Quaternion product from Fossen's report:

$$\tilde{q} = q_d \cdot q = \begin{bmatrix} \eta_d & \epsilon_{1d} & \epsilon_{2d} & \epsilon_{3d} \\ -\epsilon_{1d} & \eta_d & -\epsilon_{3d} & \epsilon_{2d} \\ -\epsilon_{2d} & \epsilon_{3d} & \eta_d & -\epsilon_{1d} \\ -\epsilon_{3d} & -\epsilon_{2d} & \epsilon_{1d} & \eta_d \end{bmatrix} \begin{bmatrix} \eta \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \end{bmatrix} = \begin{bmatrix} \eta_d\eta + \epsilon_{1d}\epsilon_1 + \epsilon_{2d}\epsilon_2 + \epsilon_{3d}\epsilon_3 \\ -\epsilon_{1d}\eta + \eta_d\epsilon_1 - \epsilon_{3d}\epsilon_2 + \epsilon_{2d}\epsilon_3 \\ -\epsilon_{2d}\eta + \epsilon_{3d}\epsilon_1 + \eta_d\epsilon_2 - \epsilon_{1d}\epsilon_3 \\ -\epsilon_{3d}\eta - \epsilon_{2d}\epsilon_1 + \epsilon_{1d}\epsilon_2 + \eta_d\epsilon_3 \end{bmatrix} \tag{B.2}$$

$$If \ q_d \ = \ q \ \rightarrow \ \tilde{q} \ = \begin{bmatrix} \pm 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{B.3}$$

Implementation in cost function :

$$\begin{aligned} J_q = \quad & ((\eta_d\eta + \epsilon_{1d}\epsilon_1 + \epsilon_{2d}\epsilon_2 + \epsilon_{3d}\epsilon_3)^2 - 1)^2 \\ & +(-\epsilon_{1d}\eta + \eta_d\epsilon_1 - \epsilon_{3d}\epsilon_2 + \epsilon_{2d}\epsilon_3)^2 \\ & +(-\epsilon_{2d}\eta + \epsilon_{3d}\epsilon_1 + \eta_d\epsilon_2 - \epsilon_{1d}\epsilon_3)^2 \\ & +(-\epsilon_{3d}\eta - \epsilon_{2d}\epsilon_1 + \epsilon_{1d}\epsilon_2 + \eta_d\epsilon_3)^2 \end{aligned} \tag{B.4}$$

[1] O.-E. Fjellstad and T. I. Fossen, 'Quaternion feedback regulation of underwater vehicles,' in 1994 Proceedings of IEEE International Conference on Control and Applications, 1994, 857–862 vol.2. DOI:10.1109/CCA.1994.381209

Appendix C

RQt-graph

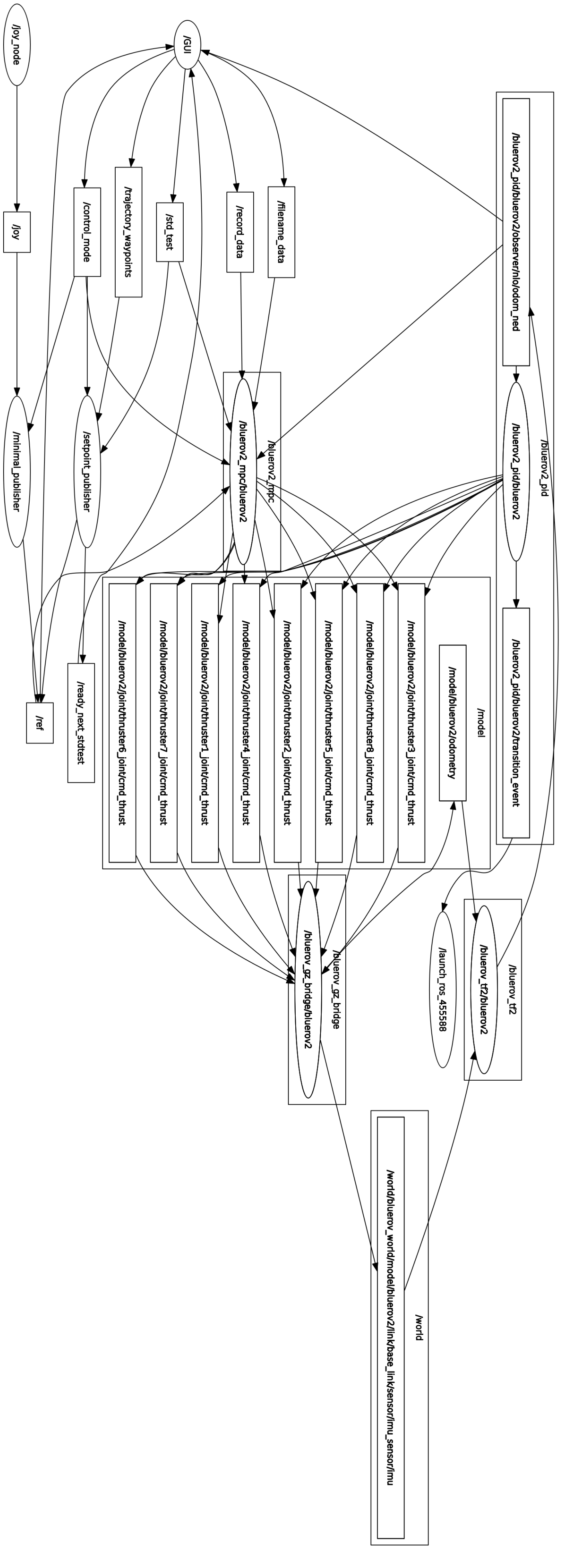

/joy_node
/joy
/minimal_publisher
/GUI
/control_mode
/trajectory_waypoints
/std_test
/record_data
/filename_data
/setpoint_publisher
/ref
/ready_next_stdtest
/bluerov2_mpc
/bluerov2_mpc/bluerov2
/bluerov2_pid
/bluerov2_pid/bluerov2/observer/nlo/odom_ned
/bluerov2_pid/bluerov2
/bluerov2_pid/bluerov2/transition_event
/model
/model/bluerov2/joint/thruster6_joint/cmd_thrust
/model/bluerov2/joint/thruster7_joint/cmd_thrust
/model/bluerov2/joint/thruster1_joint/cmd_thrust
/model/bluerov2/joint/thruster4_joint/cmd_thrust
/model/bluerov2/joint/thruster2_joint/cmd_thrust
/model/bluerov2/joint/thruster5_joint/cmd_thrust
/model/bluerov2/joint/thruster8_joint/cmd_thrust
/model/bluerov2/joint/thruster3_joint/cmd_thrust
/model/bluerov2/odometry
/bluerov_gz_bridge
/bluerov_gz_bridge/bluerov2
/launch_ros_455588
/bluerov_tf2
/bluerov_tf2/bluerov2
/world
/world/bluerov_world/model/bluerov2/link/base_link/sensor/imu_sensor/imu

Figure C.1: ROS topology with one agent

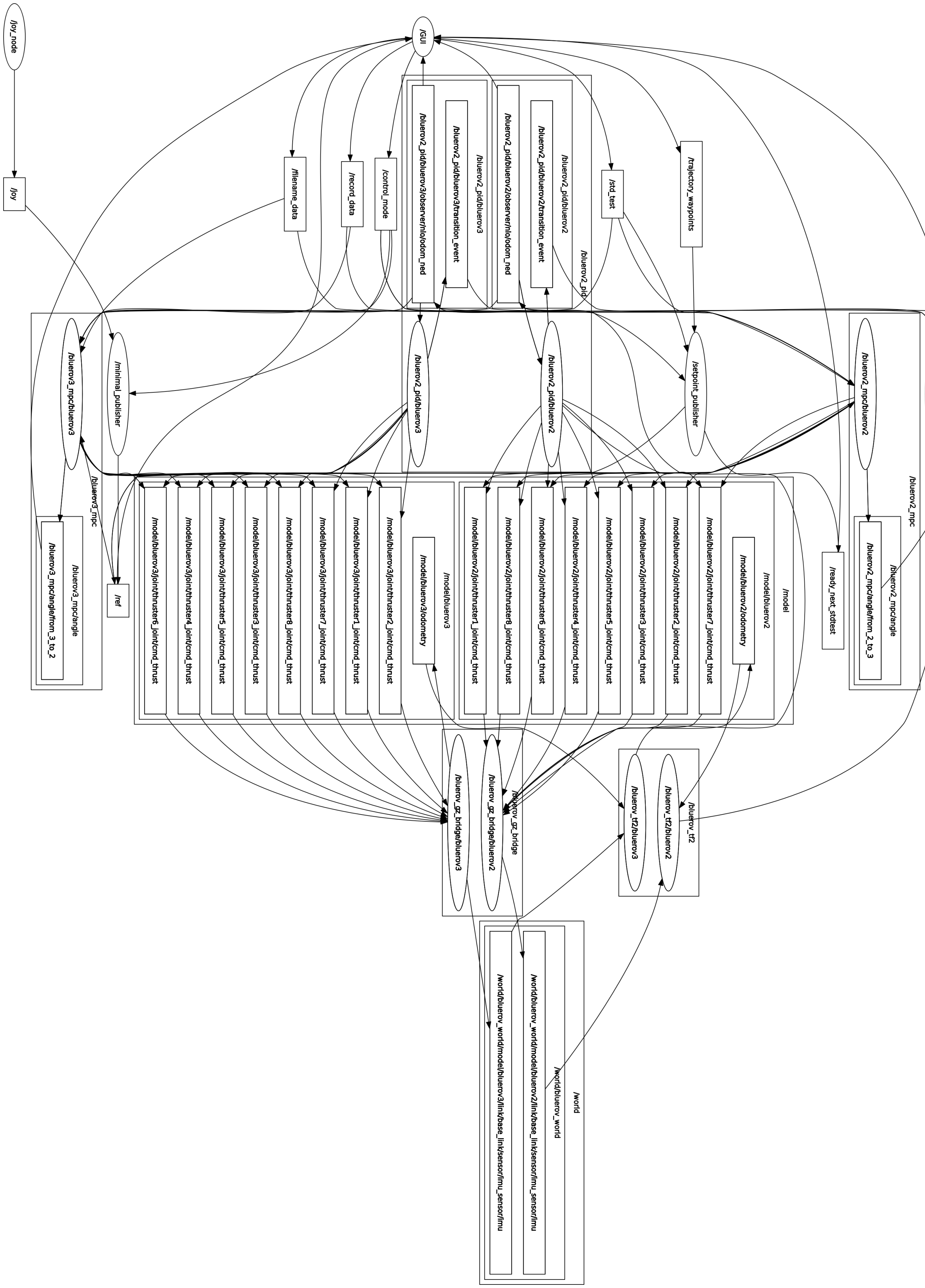

/joy_node
/joy
/GUI
/filename_data
/record_data
/control_mode
/bluerov2_pid/bluerov3/observer/nlo/odom_ned
/bluerov2_pid/bluerov3/transition_event
/bluerov2_pid/bluerov3
/bluerov2_pid/bluerov2/observer/nlo/odom_ned
/bluerov2_pid/bluerov2/transition_event
/bluerov2_pid/bluerov2
/bluerov2_pid
/std_test
/trajectory_waypoints
/bluerov3_mpc/bluerov3
/minimal_publisher
/bluerov2_pid/bluerov3
/bluerov2_pid/bluerov2
/setpoint_publisher
/bluerov2_mpc/bluerov2
/launch_ros_313389
/bluerov3_mpc
/bluerov3_mpc/angle
/bluerov3_mpc/angle/from_3_to_2
/ref
/model/bluerov3/joint/thruster6_joint/cmd_thrust
/model/bluerov3/joint/thruster4_joint/cmd_thrust
/model/bluerov3/joint/thruster5_joint/cmd_thrust
/model/bluerov3/joint/thruster3_joint/cmd_thrust
/model/bluerov3/joint/thruster8_joint/cmd_thrust
/model/bluerov3/joint/thruster7_joint/cmd_thrust
/model/bluerov3/joint/thruster1_joint/cmd_thrust
/model/bluerov3/joint/thruster2_joint/cmd_thrust
/model/bluerov3/odometry
/model/bluerov3
/model/bluerov2/joint/thruster1_joint/cmd_thrust
/model/bluerov2/joint/thruster8_joint/cmd_thrust
/model/bluerov2/joint/thruster6_joint/cmd_thrust
/model/bluerov2/joint/thruster4_joint/cmd_thrust
/model/bluerov2/joint/thruster5_joint/cmd_thrust
/model/bluerov2/joint/thruster3_joint/cmd_thrust
/model/bluerov2/joint/thruster2_joint/cmd_thrust
/model/bluerov2/joint/thruster7_joint/cmd_thrust
/model/bluerov2/odometry
/model/bluerov2
/model
/ready_next_stdtest
/bluerov2_mpc/angle/from_2_to_3
/bluerov2_mpc/angle
/bluerov2_mpc
/bluerov_gz_bridge/bluerov3
/bluerov_gz_bridge/bluerov2
/bluerov_gz_bridge
/bluerov_tf2/bluerov3
/bluerov_tf2/bluerov2
/bluerov_tf2
/world/bluerov_world/model/bluerov3/link/base_link/sensor/imu_sensor/imu
/world/bluerov_world/model/bluerov2/link/base_link/sensor/imu_sensor/imu
/world/bluerov_world
/world

Figure C.2: ROS topology with two agents

Appendix D

Computer Specifications

Computer Specifications for Computer 1

Model	Custom
Processor	AMD Ryzen 7 5800X CPU @ 4.80GHz
Memory	32000MB
Graphics	Nvidia RTX 3080
Operating System	Ubuntu 22.04.2 LTS

Computer Specifications for Computer 2

Model	Lenovo Yoga Slim 7
Processor	AMD Ryzen 7 4700U @ 2.00GHz
Memory	16000MB
Graphics	AMD Radeon RX Vega 8
Operating System	Ubuntu 22.04.2 LTS

Computer Specifications for Computer 3

Model	Dell Inc. OptiPlex 7050
Processor	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
Memory	32734MB
Operating System	Ubuntu 22.04.2 LTS

Appendix E

List of tests done

Test name	Sample size	PC	Note
Python simulator	1	NA	Used in results
Default test	632	All	Used in results (several computers)
Current 0.5 m/s	228	3	Used in results
Current 0.25 m/s	216	3	Used in results
Circle setpoint	210	2	Used in results
70% Packet loss	206	2	Used in results
Double added mass	206	1	Used in results
Halved Added mass	198	1	Used in results
Double damping	216	2	Used in results
Halved damping	206	2	Used in results
Doubled mass	242	3	Used in results
50% Packet loss	50	2	Packet loss 70% was prioritised
Current 0.3m/s x-direction	2	2	0.25 m/s and 0.5 m/s was prioritised
Current 0.3m/s -x-direction	2	2	0.25 m/s and 0.5 m/s was prioritised
Current 0.3m/s y-direction	2	2	0.25 m/s and 0.5 m/s was prioritised
Current 0.3m/s -y-direction	2	2	0.25 m/s and 0.5 m/s was prioritised
Current 0.3m/s z-direction	2	2	0.25 m/s and 0.5 m/s was prioritised
Current 0.3m/s -z-direction	2	2	0.25 m/s and 0.5 m/s was prioritised
Delayed signal between ROVs 0.5s	2	2	Packet loss was prioritised
Delayed signal between ROVs 1s	2	2	Packet loss was prioritised
Delayed signal between ROVs 2s	2	2	Packet loss was prioritised
Delayed signal between ROVs 4s	2	2	Packet loss was prioritised
Delayed signal between ROVs 5s	2	2	Packet loss was prioritised
Delayed signal between ROVs 7s	2	2	Packet loss was prioritised
Delayed signal between ROVs 10s	2	2	Packet loss was prioritised
One ROV stopped	2	2	Packet loss was prioritised
White noise positioning signal	200	2	Packet loss was prioritised
Mass 1kg	2	2	Improbable to occur
Mass 100kg	2	2	Improbable to occur
Lowered CG 2cm under CB	2	1	Had little impact on test
Double inertia	2	1	Had little impact on test
Halved inertia	2	1	Had little impact on test
Halved mass	2	2	Improbable to occur
Stochastic noise in the signal. ±1	2	2	Packet loss was prioritised
Stochastic noise in the signal. ±2	2	2	Packet loss was prioritised
Stochastic noise in the signal. ±3	2	2	Packet loss was prioritised
Worst case test	248	3	Not prioritized for report

Appendix F

Complete Collection of Results from the Python Simulator

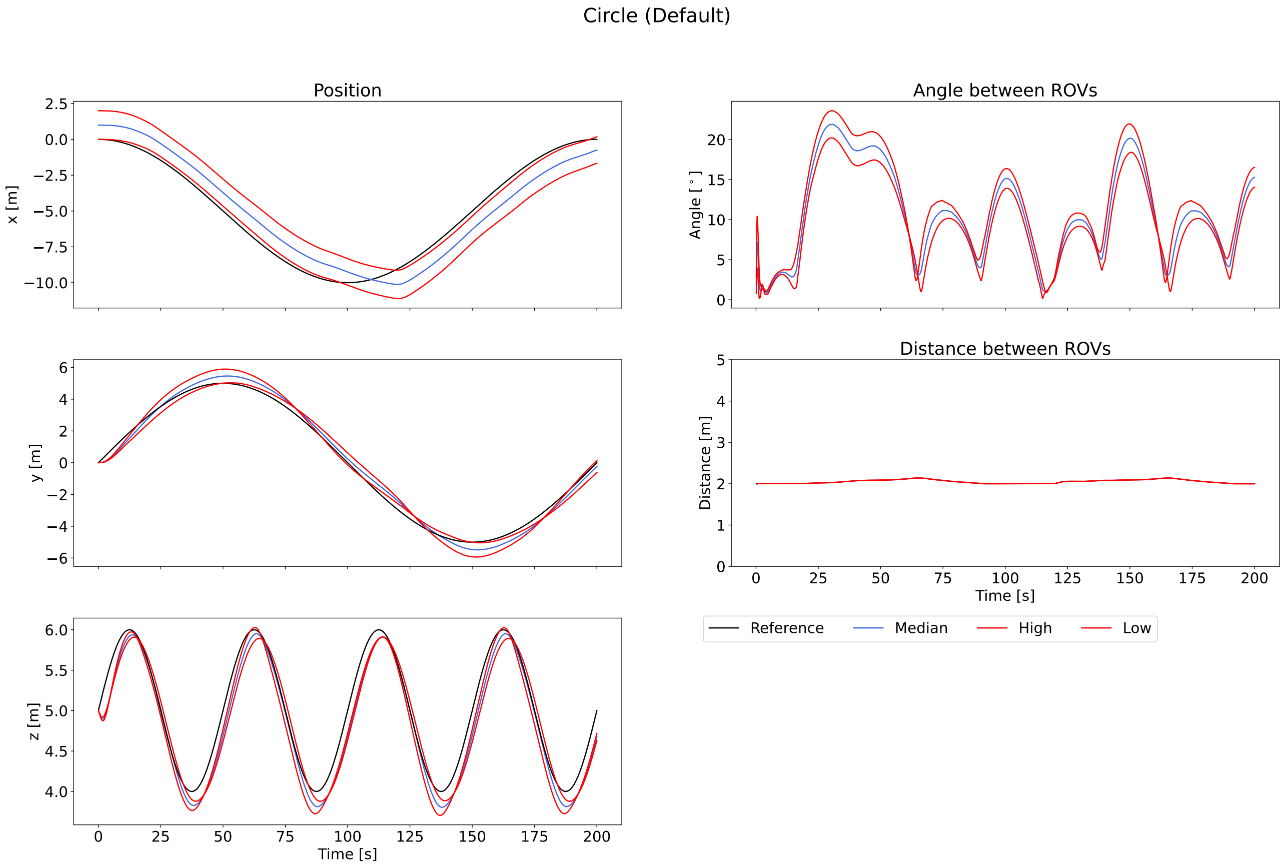


Figure F.1: Circle test Python simulator

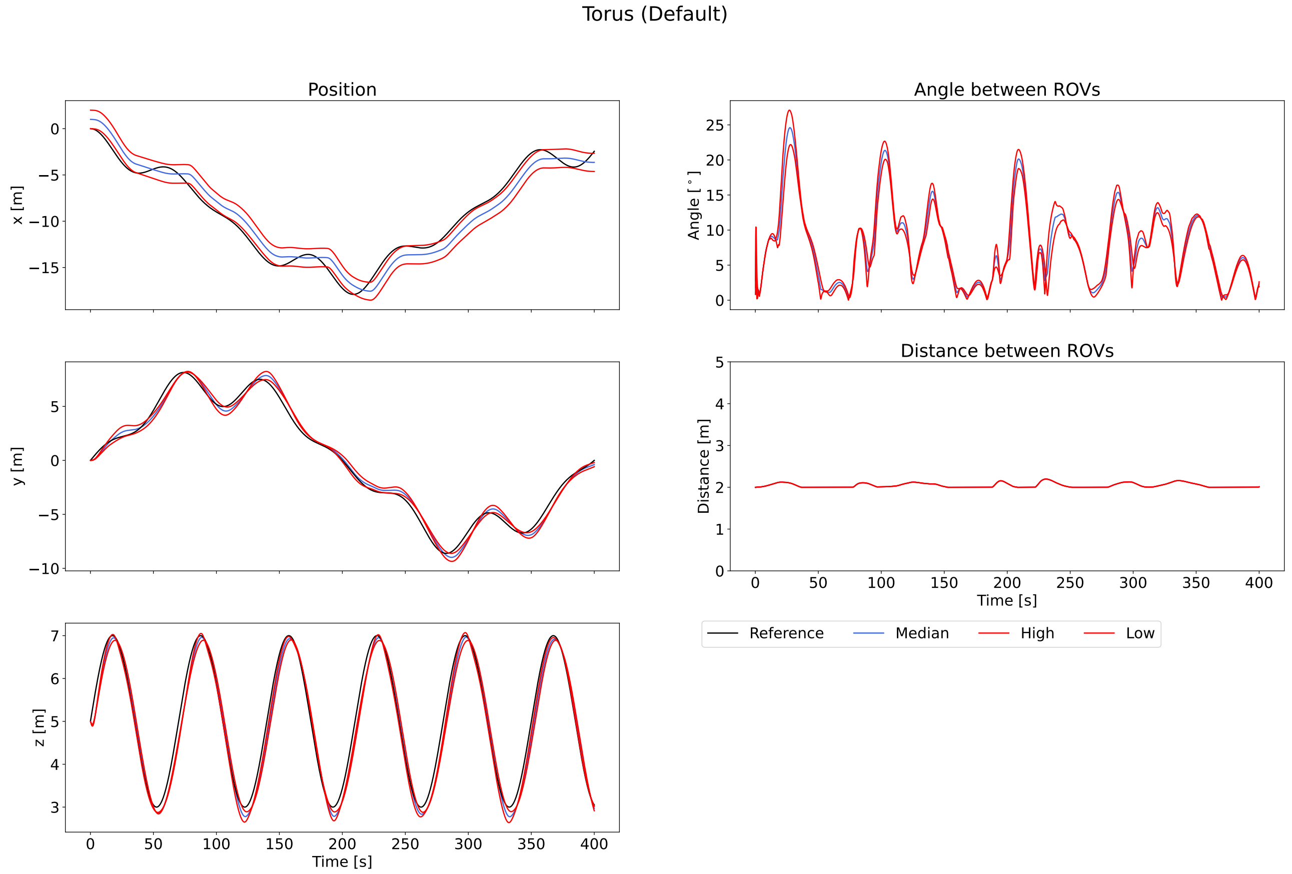


Figure F.2: Torus test Python simulator

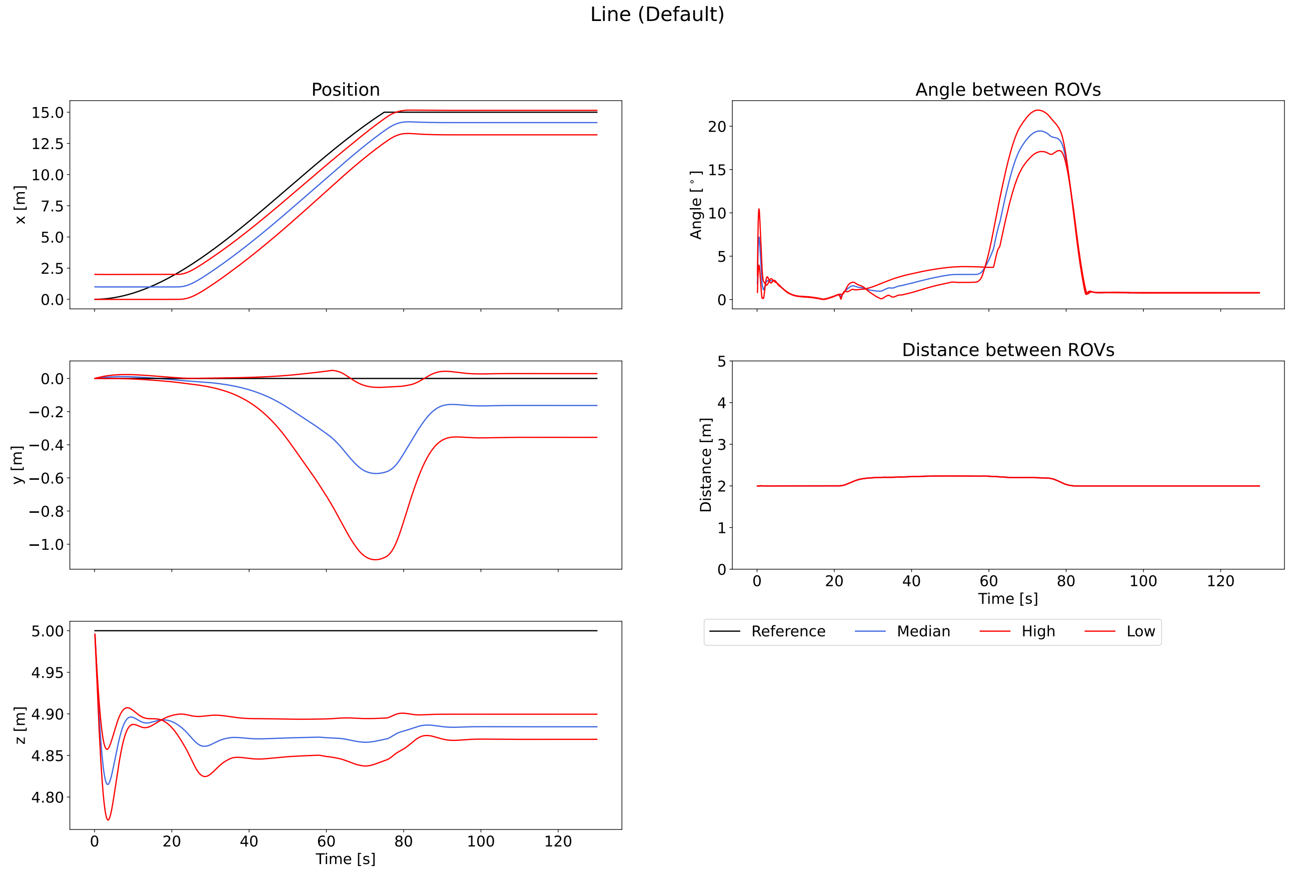


Figure F.3: Line test Python simulator

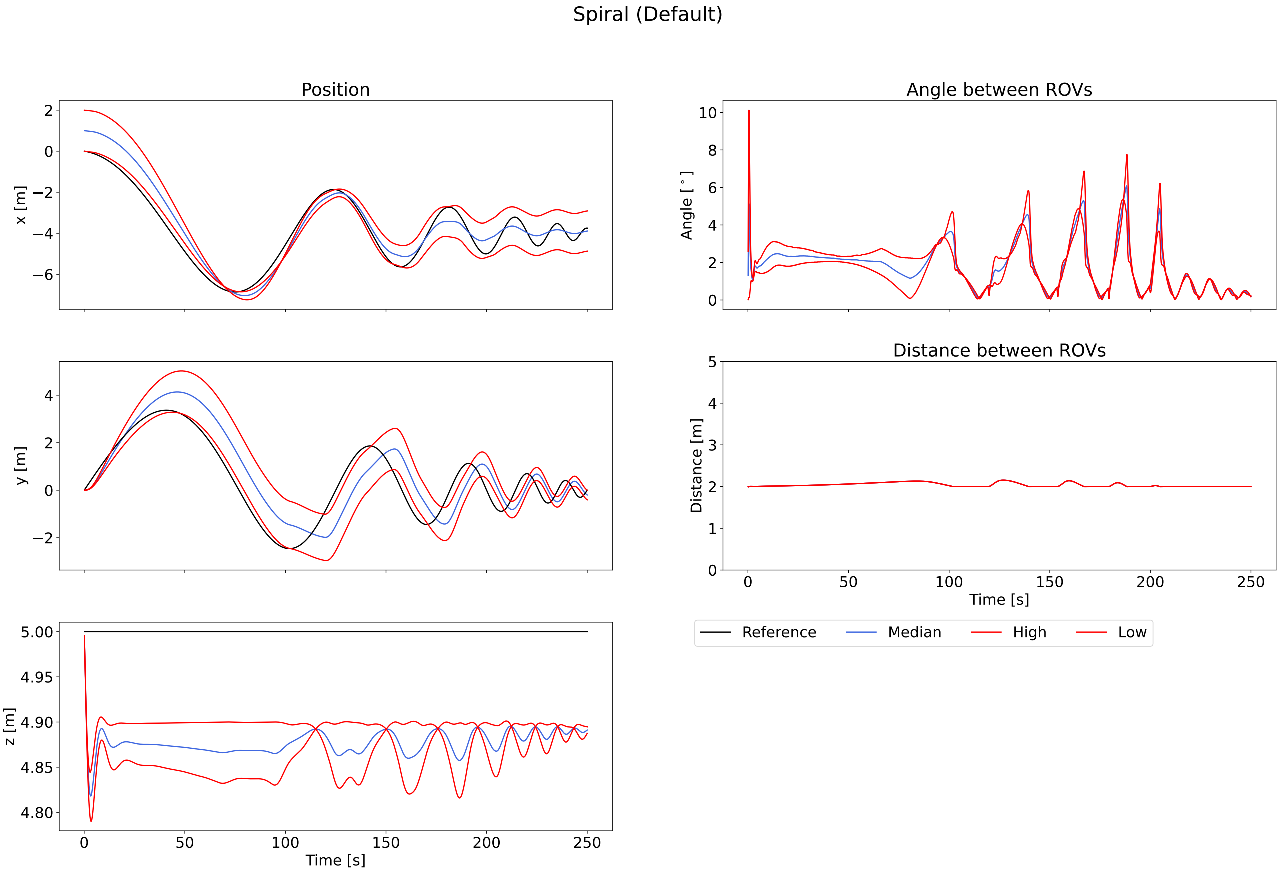


Figure F.4: Spiral test Python simulator

Appendix G

Complete Collection of Results from the Gazebo Simulator

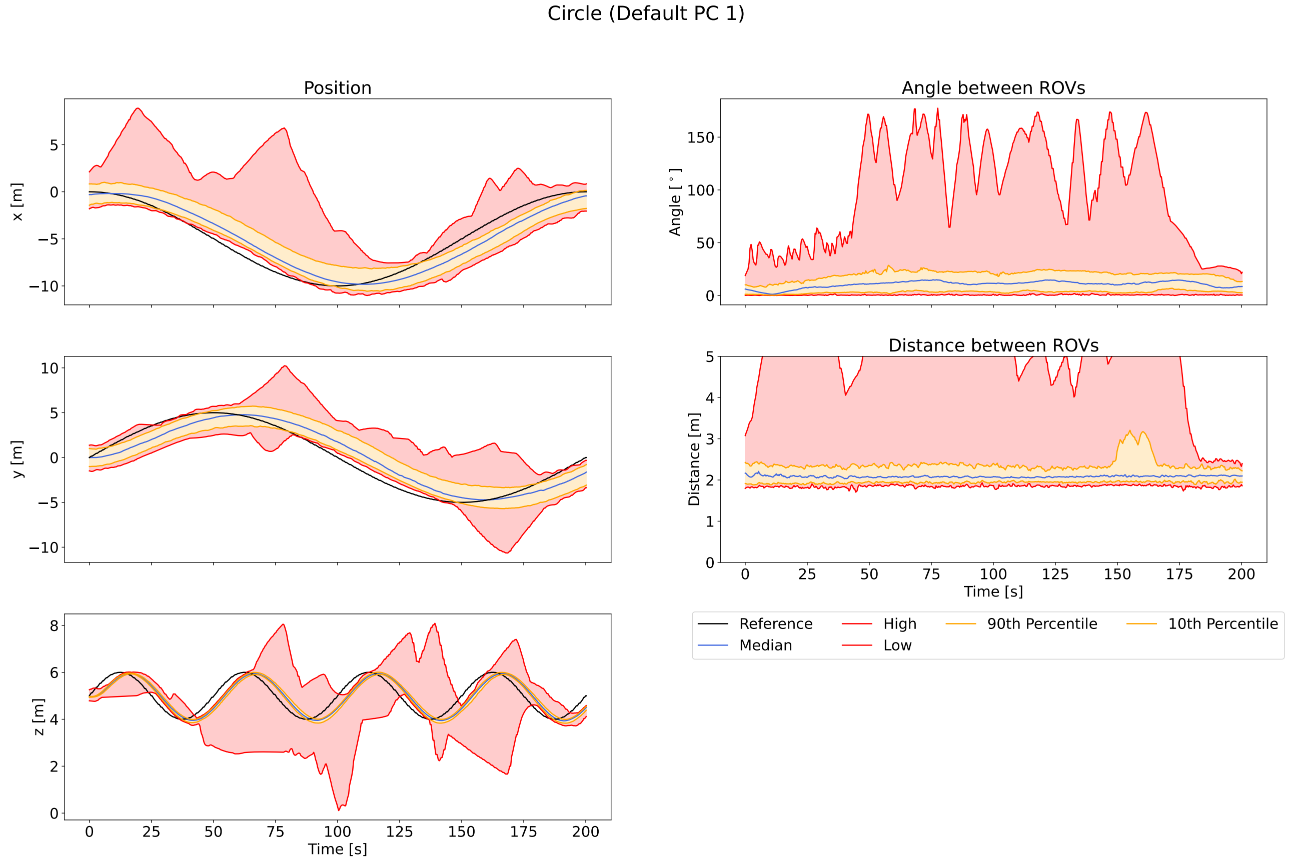


Figure G.1: Circle test under default conditions (PC 1)

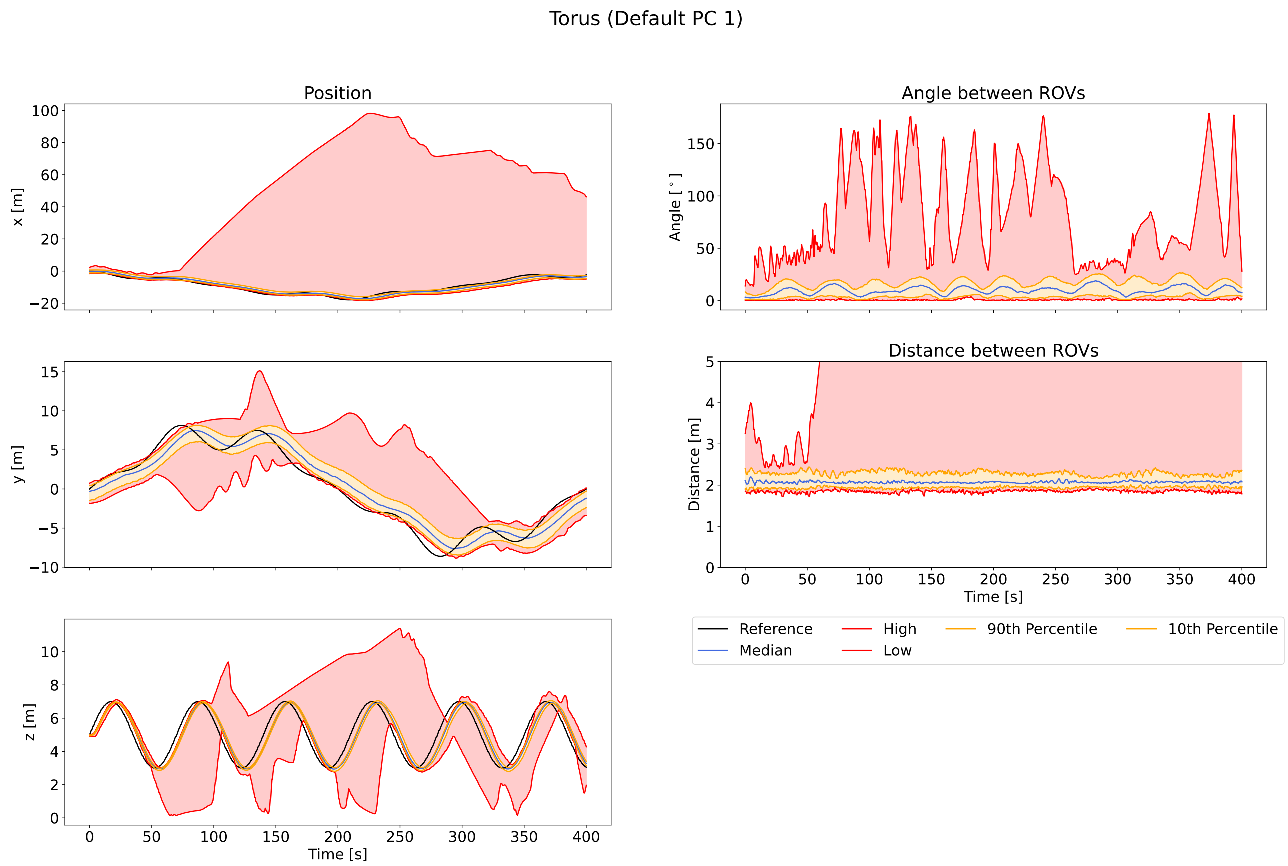


Figure G.2: Torus test under default conditions (PC 1)

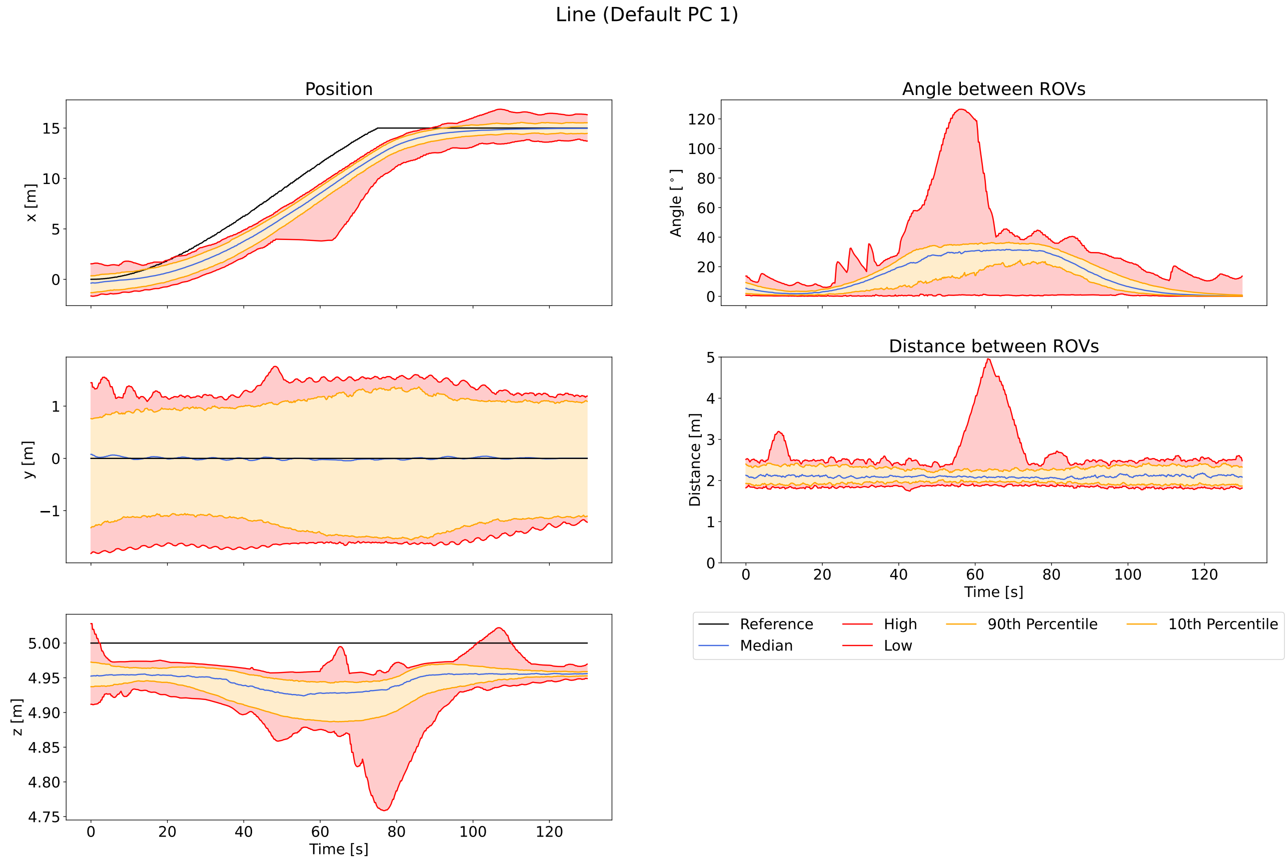


Figure G.3: Line test under default conditions (PC 1)

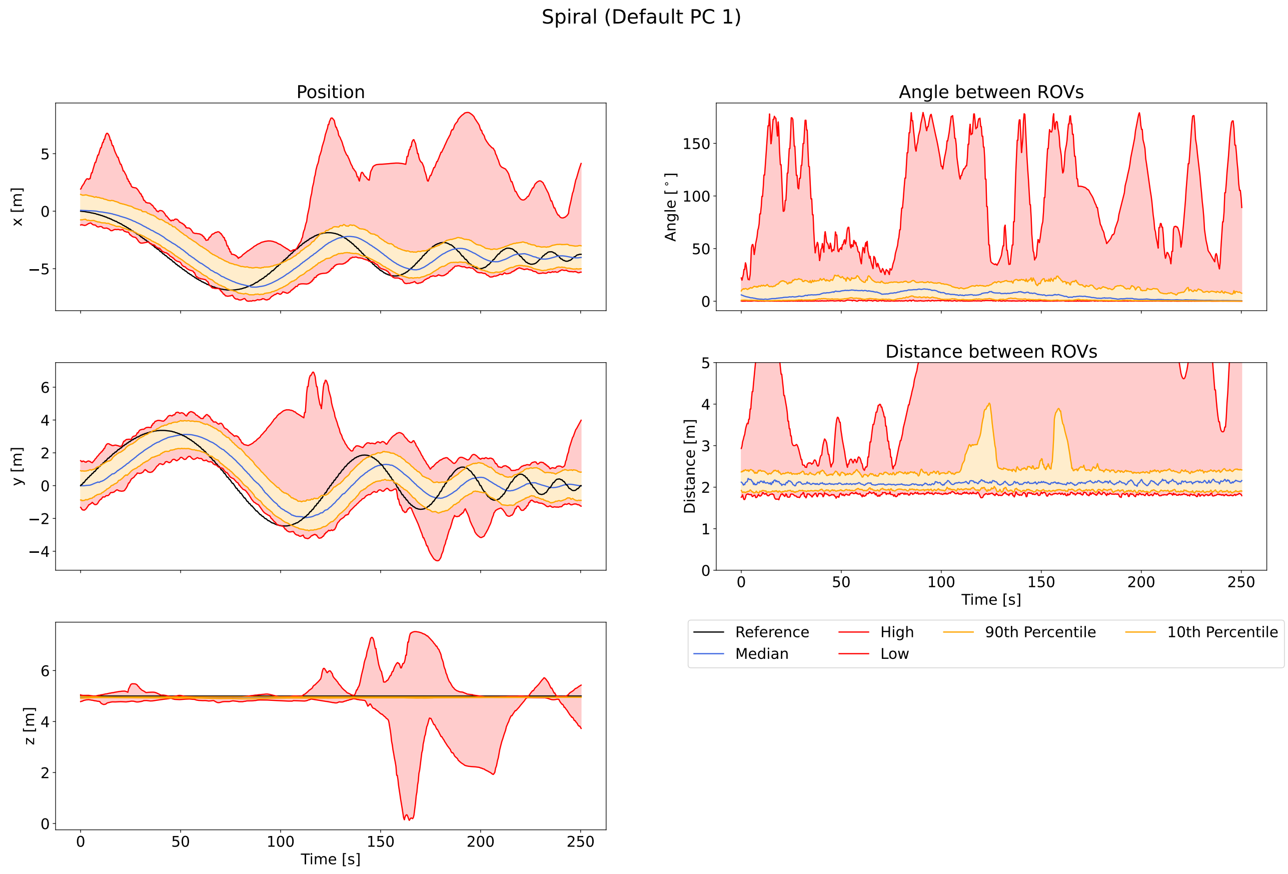


Figure G.4: Spiral test under default conditions (PC 1)

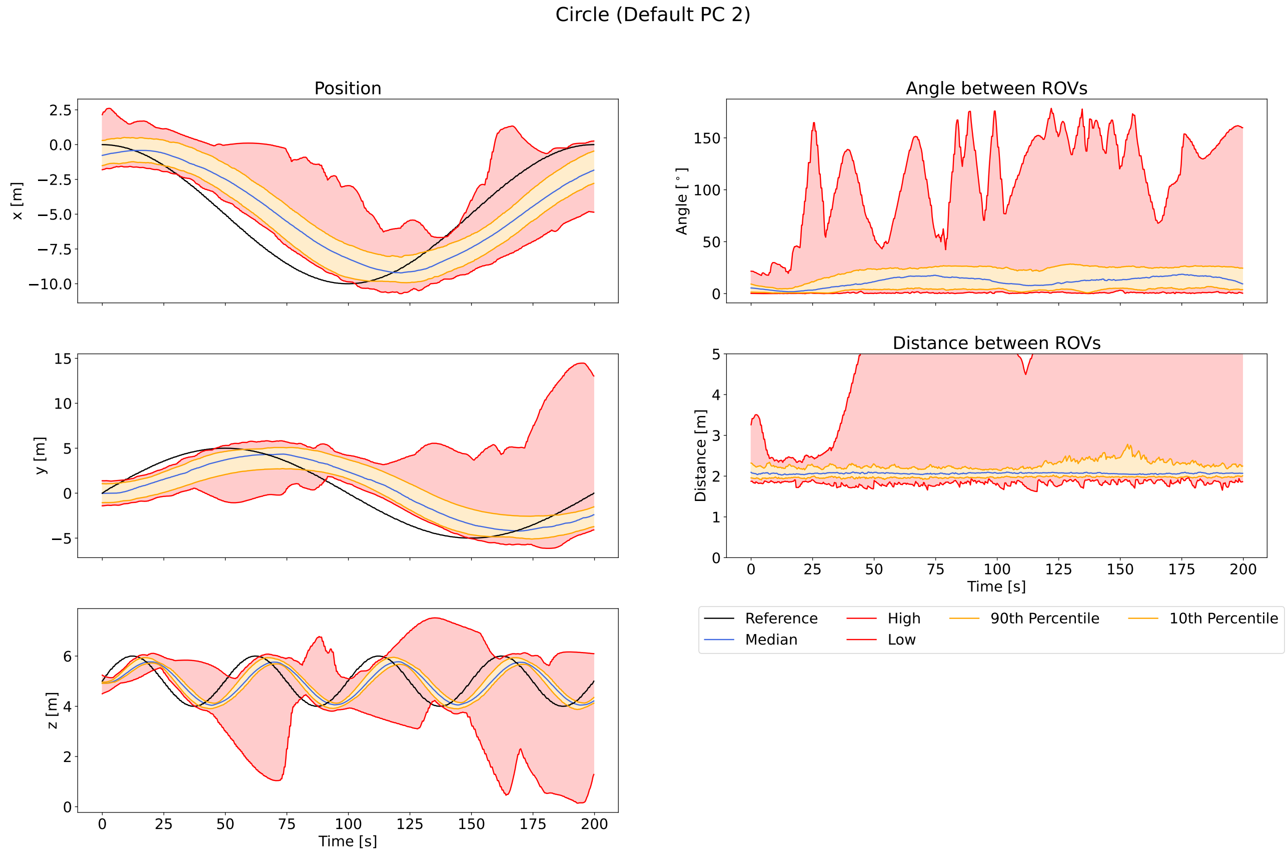


Figure G.5: Circle test under default conditions (PC 2)

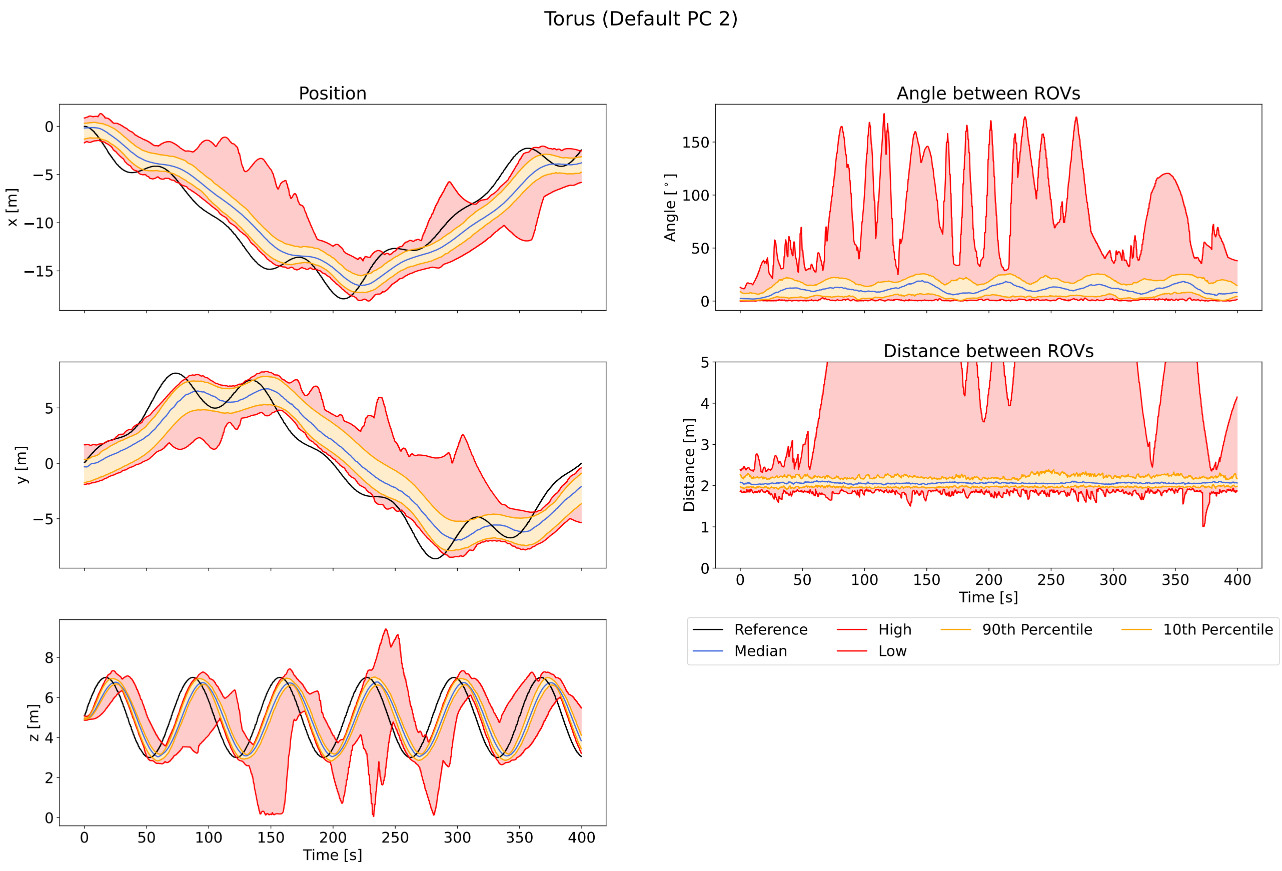


Figure G.6: Torus test under default conditions (PC 2)

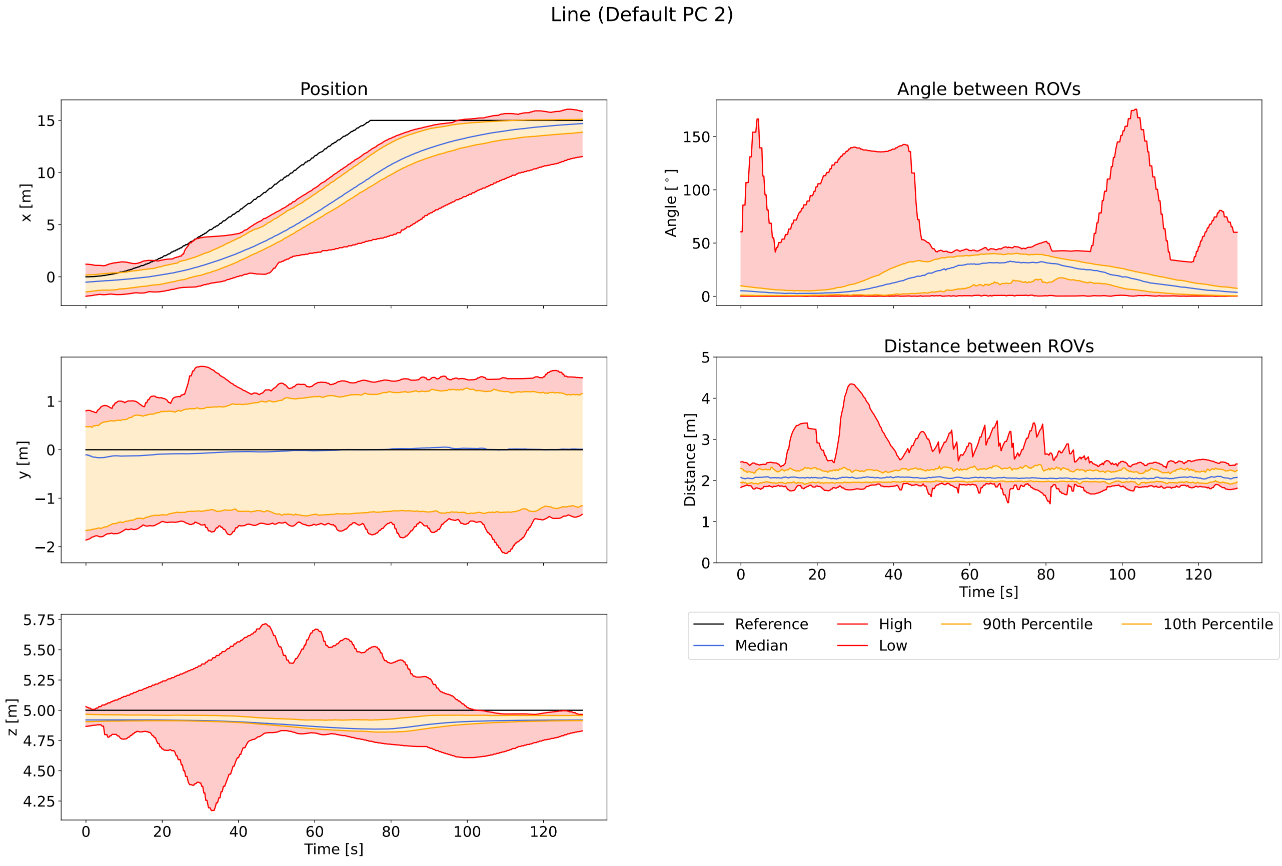


Figure G.7: Line test under default conditions (PC 2)

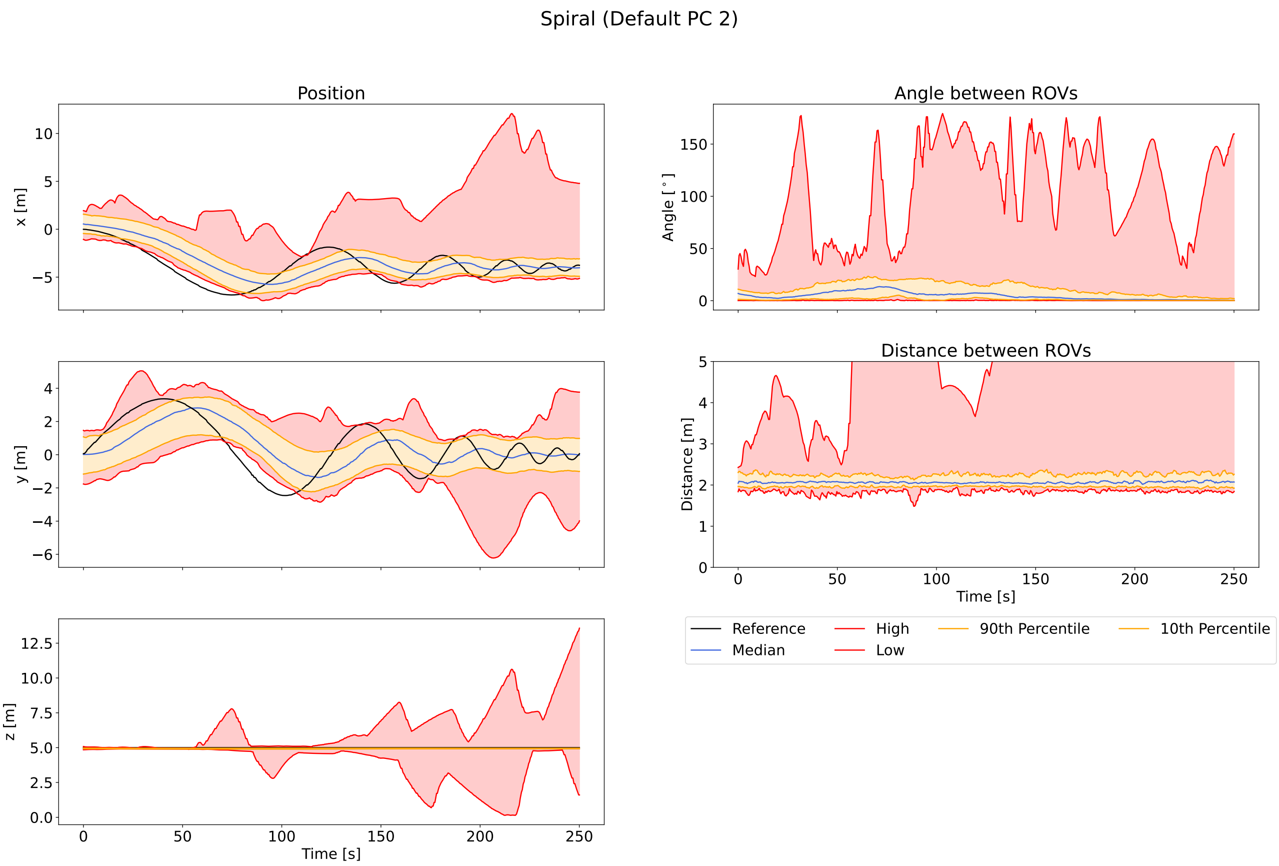


Figure G.8: Spiral test under default conditions (PC 2)

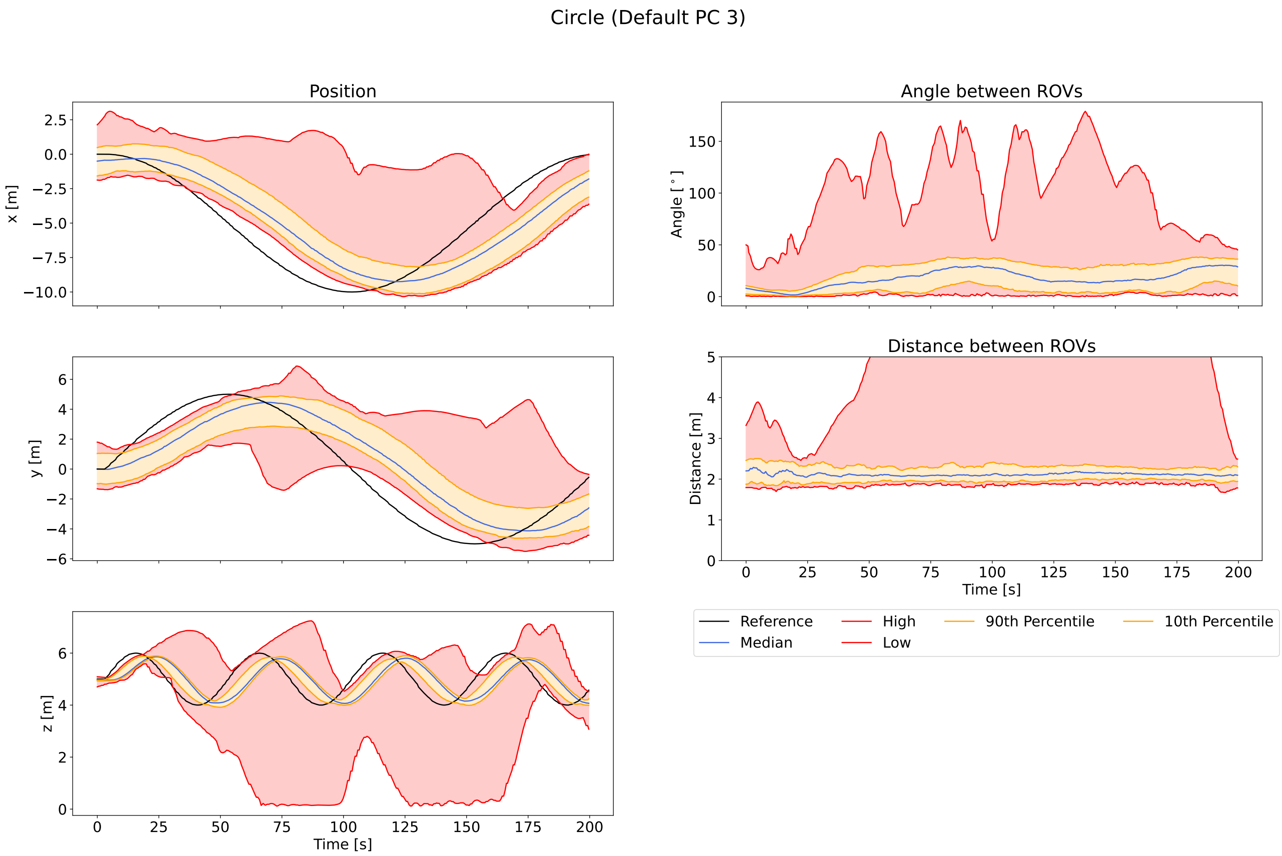


Figure G.9: Circle test under default conditions (PC 3)

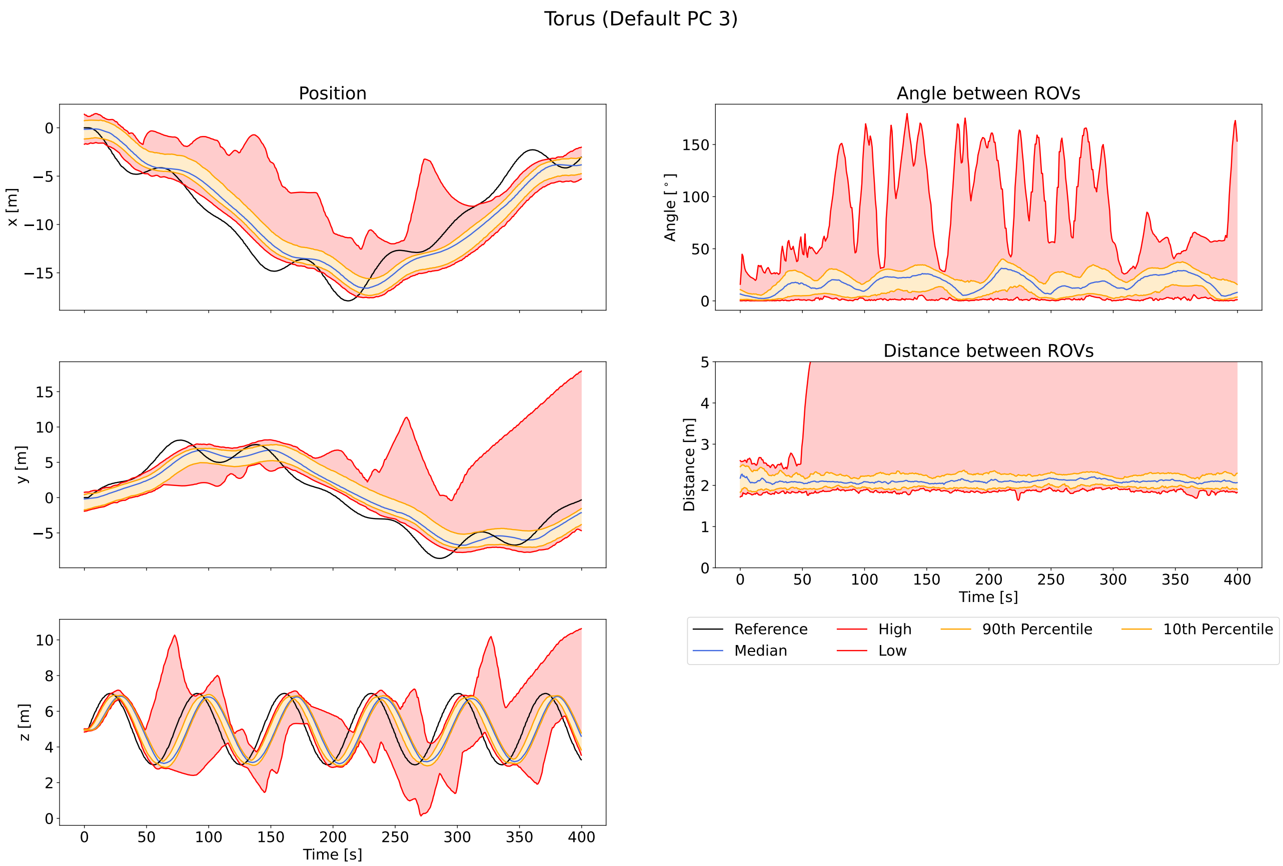


Figure G.10: Torus test under default conditions (PC 3)

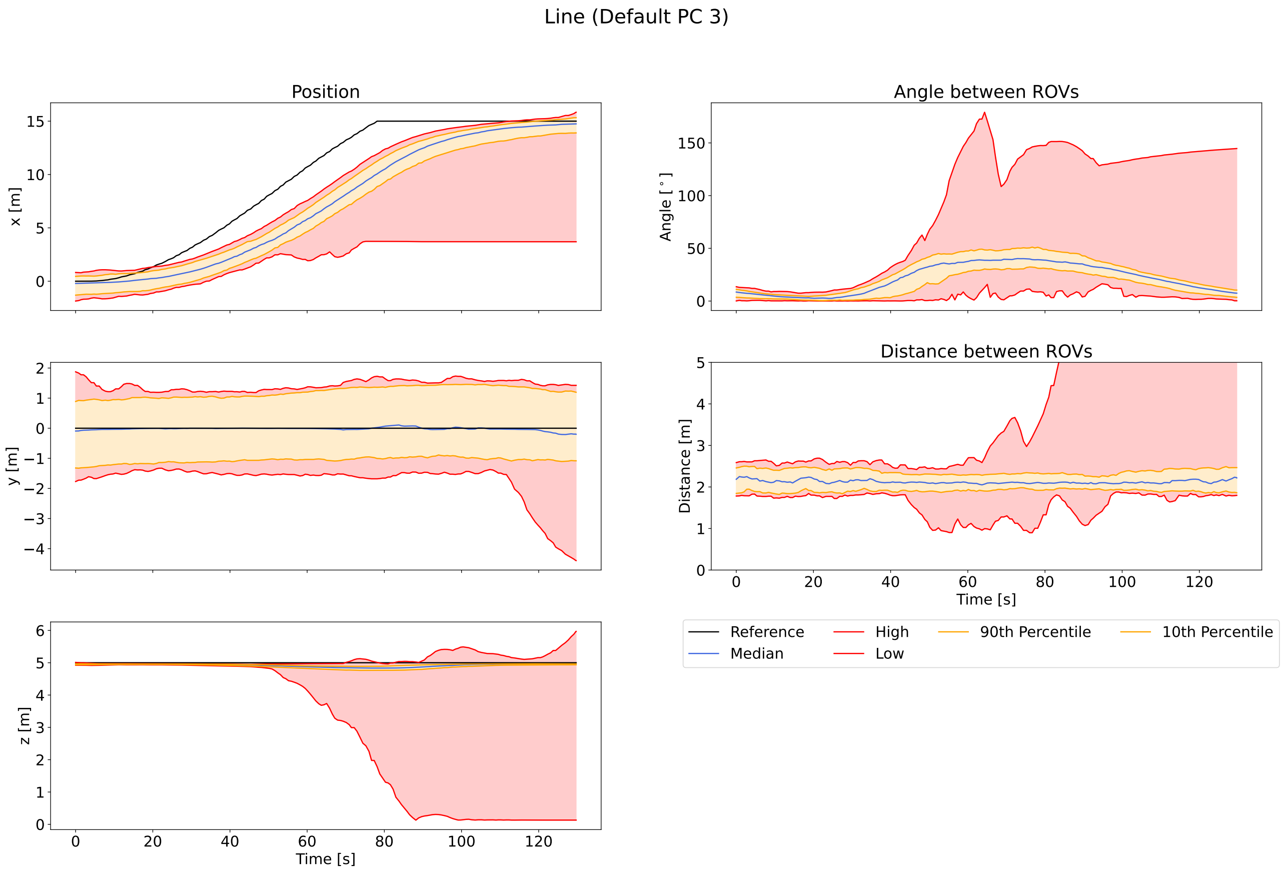


Figure G.11: Line test under default conditions (PC 3)

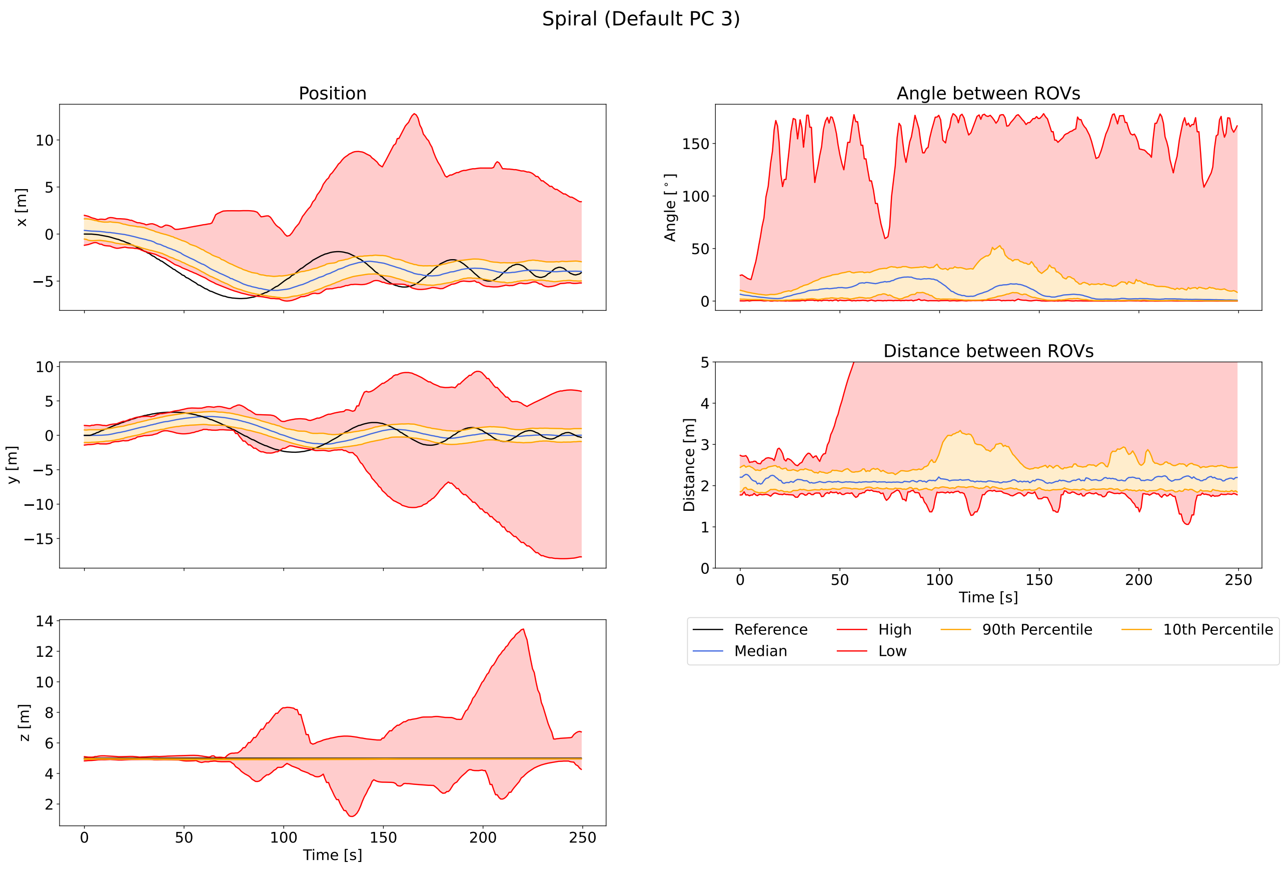


Figure G.12: Spiral test under default conditions (PC 3)

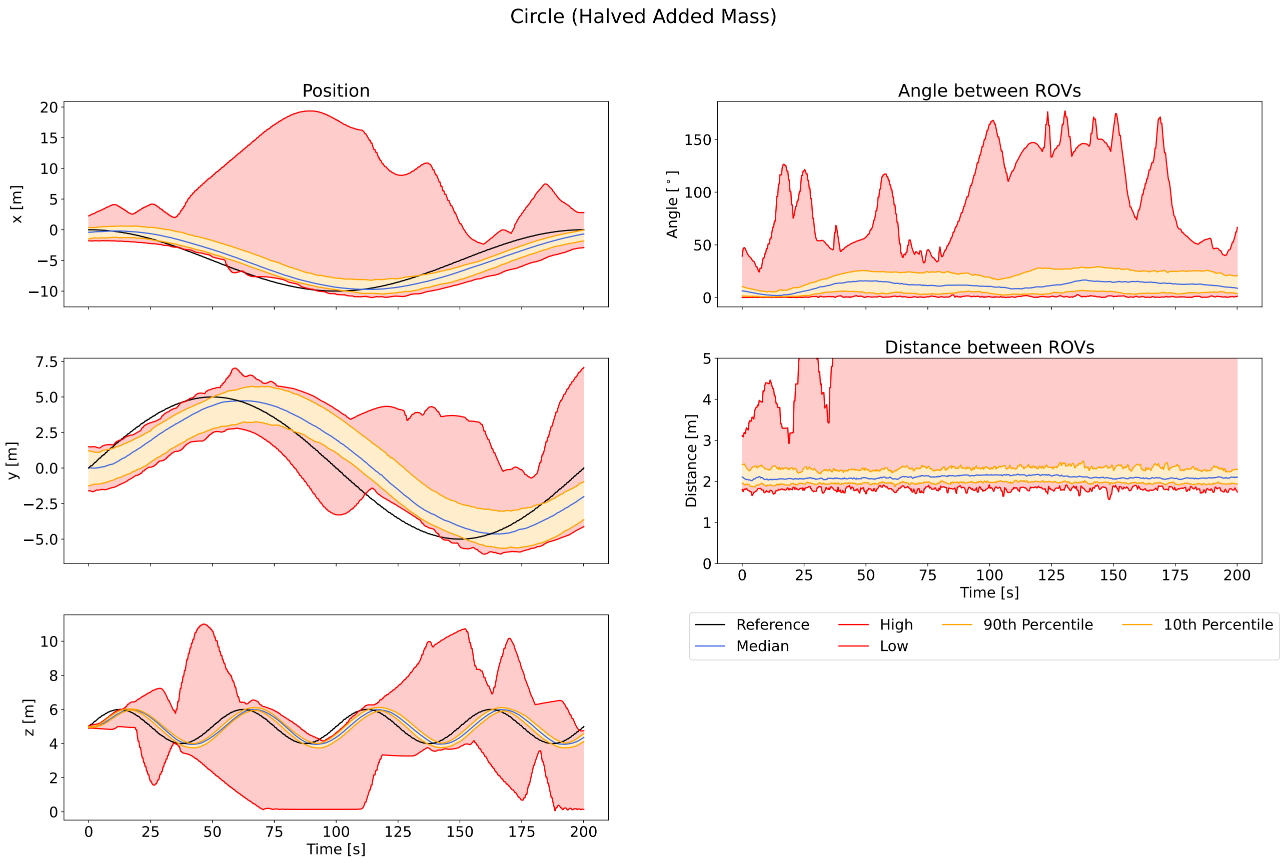


Figure G.13: Circle test with added mass parameters halved

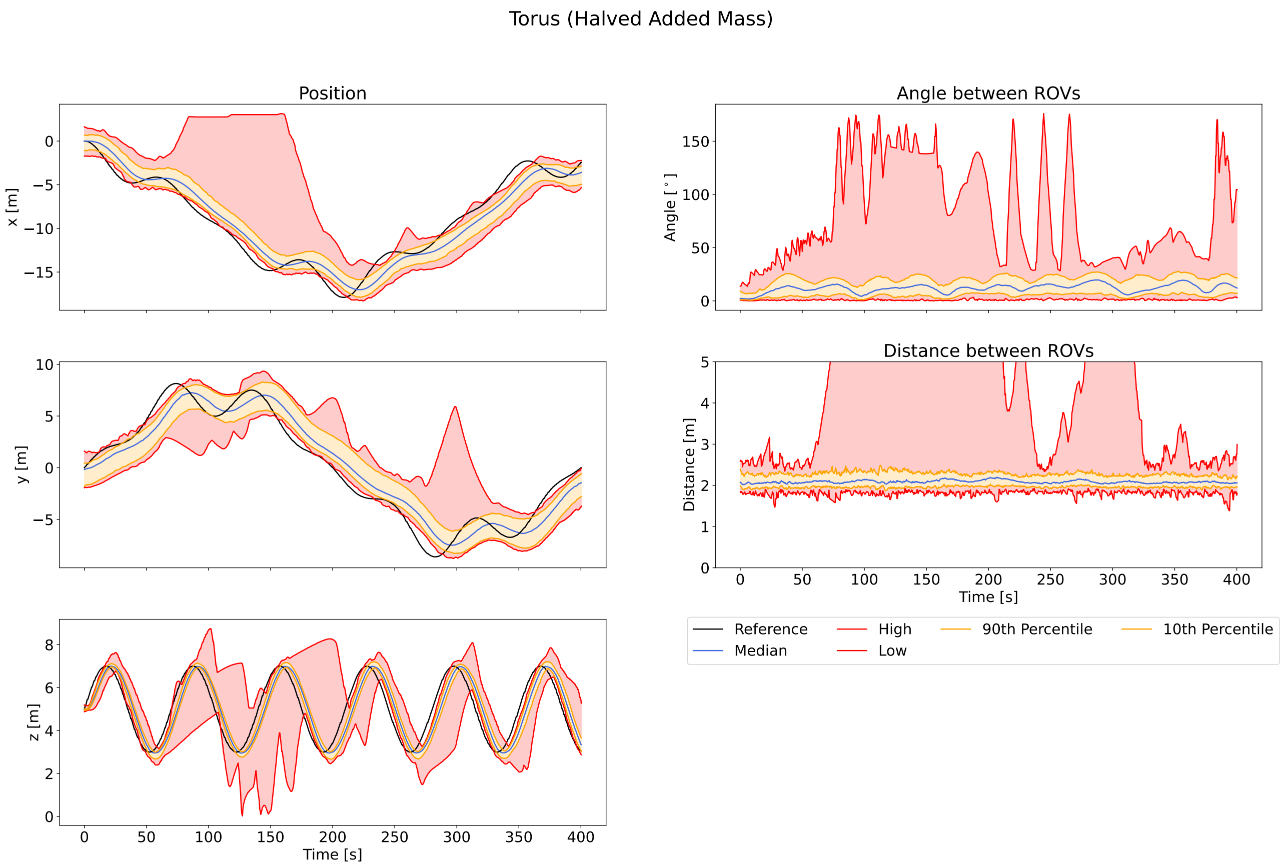


Figure G.14: Torus test with added mass parameters halved

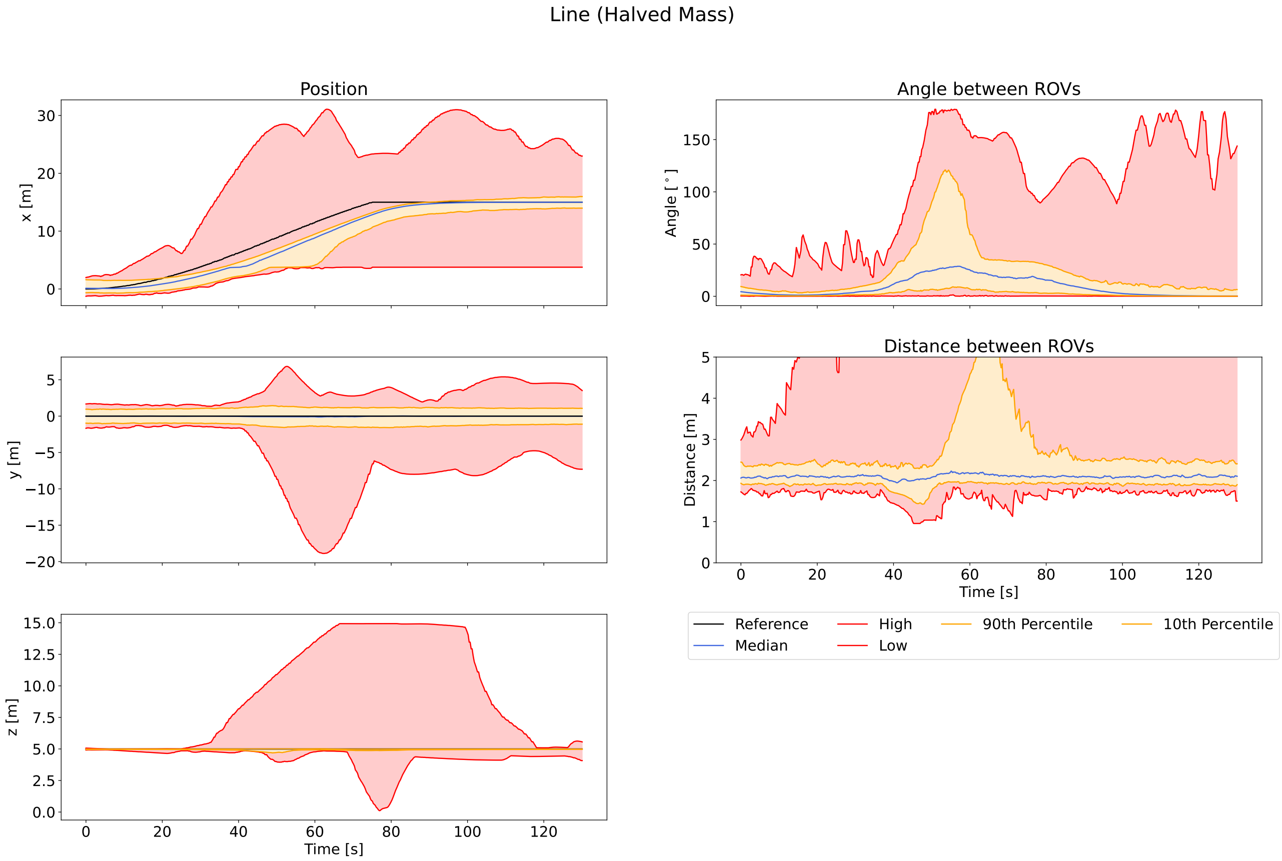


Figure G.15: Line test with added mass parameters halved

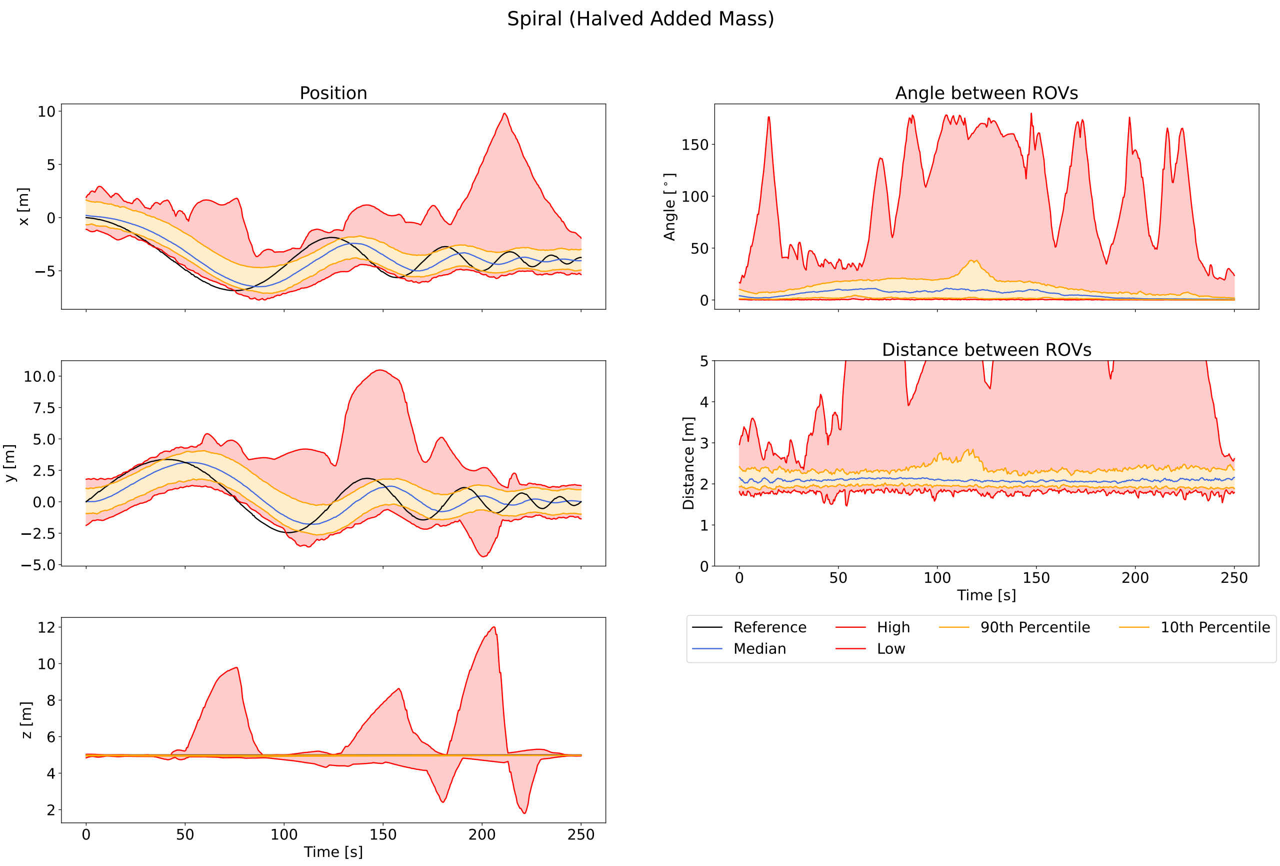


Figure G.16: Spiral test with added mass parameters halved

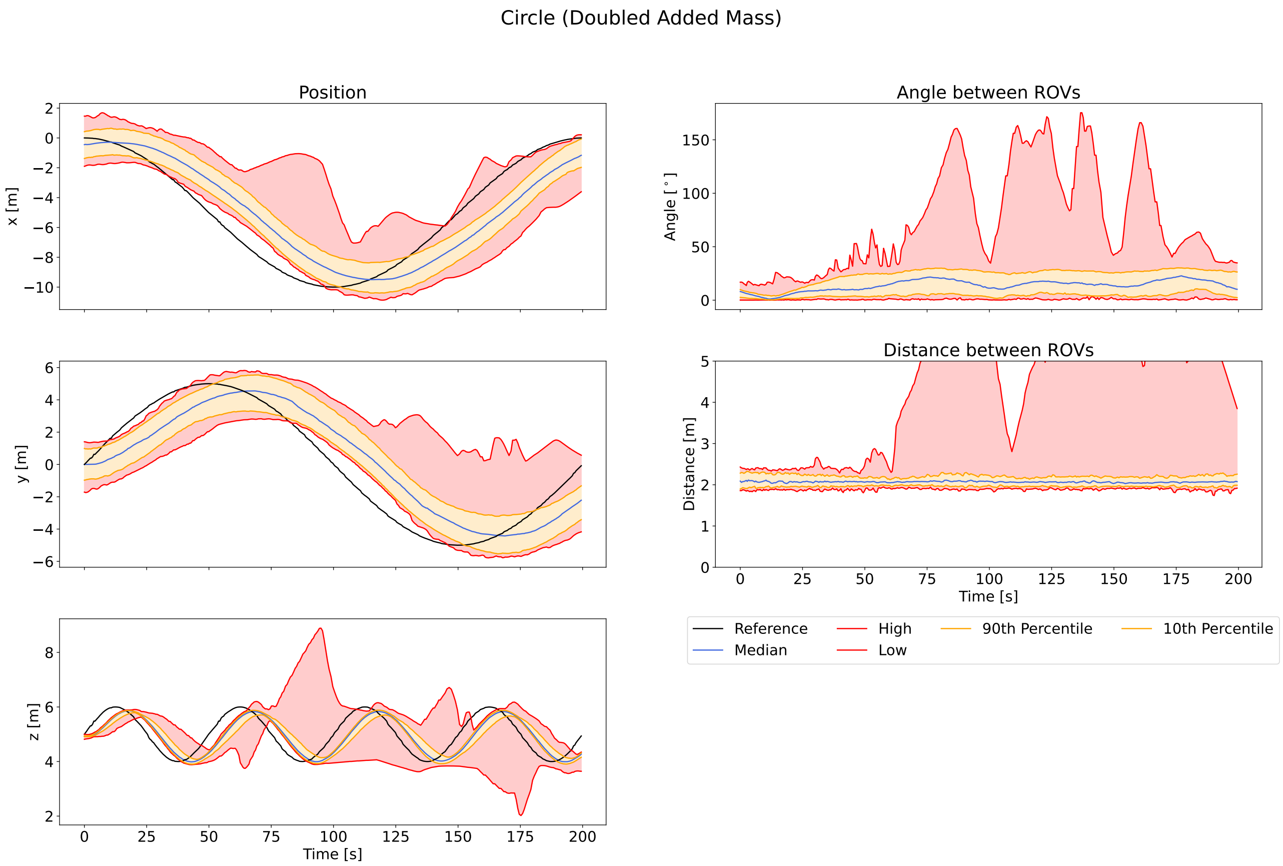


Figure G.17: Circle test with added mass parameters doubled

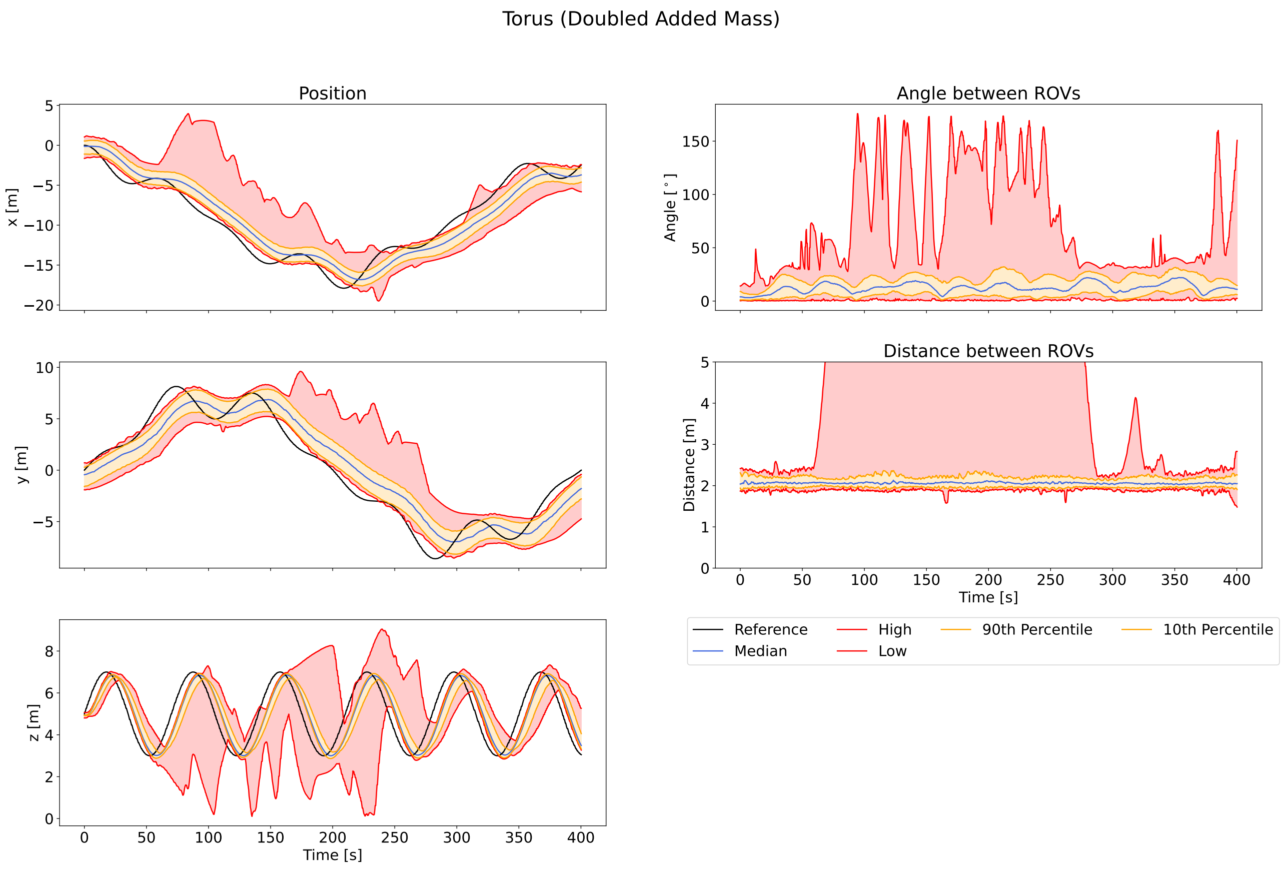


Figure G.18: Torus test with added mass parameters doubled

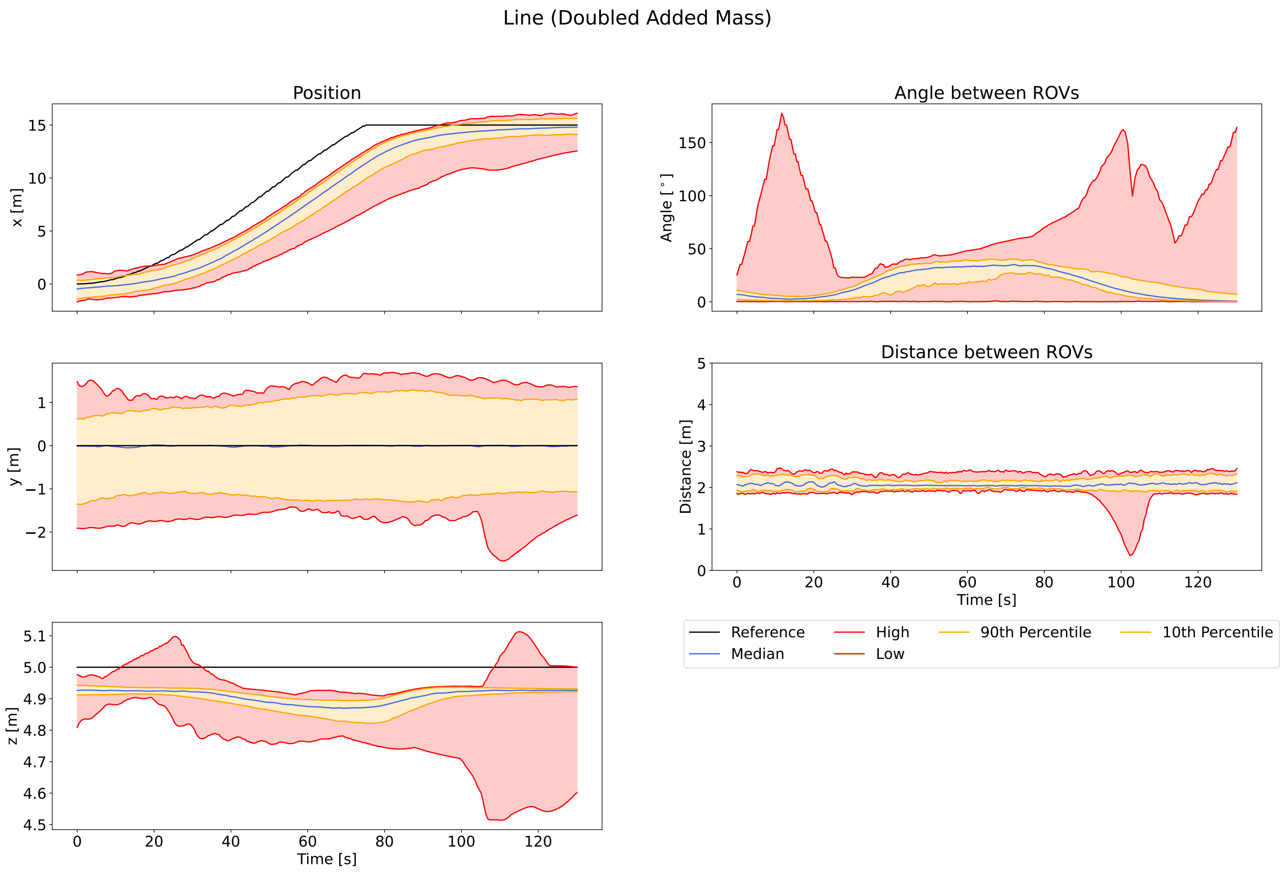


Figure G.19: Line test with added mass parameters doubled

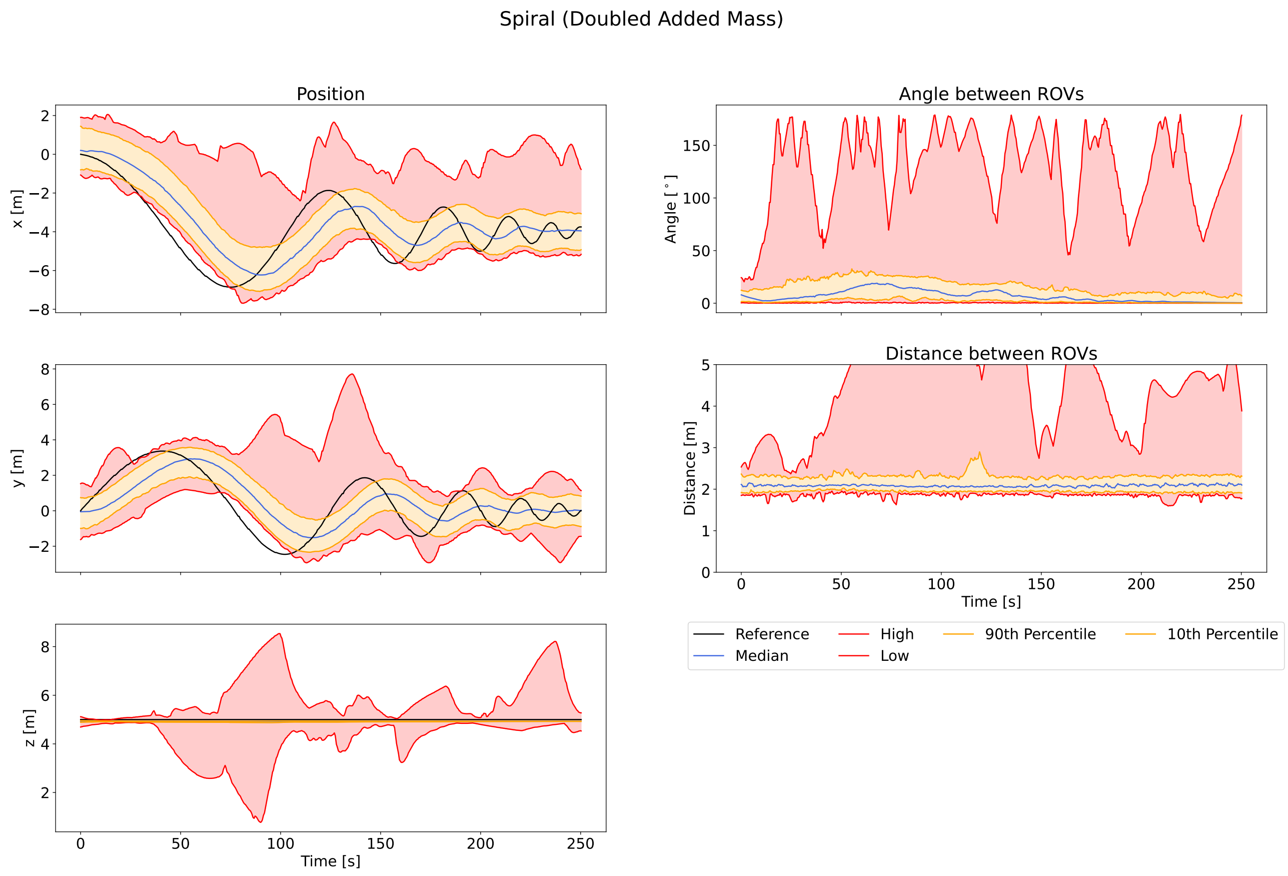


Figure G.20: Spiral test with added mass parameters doubled

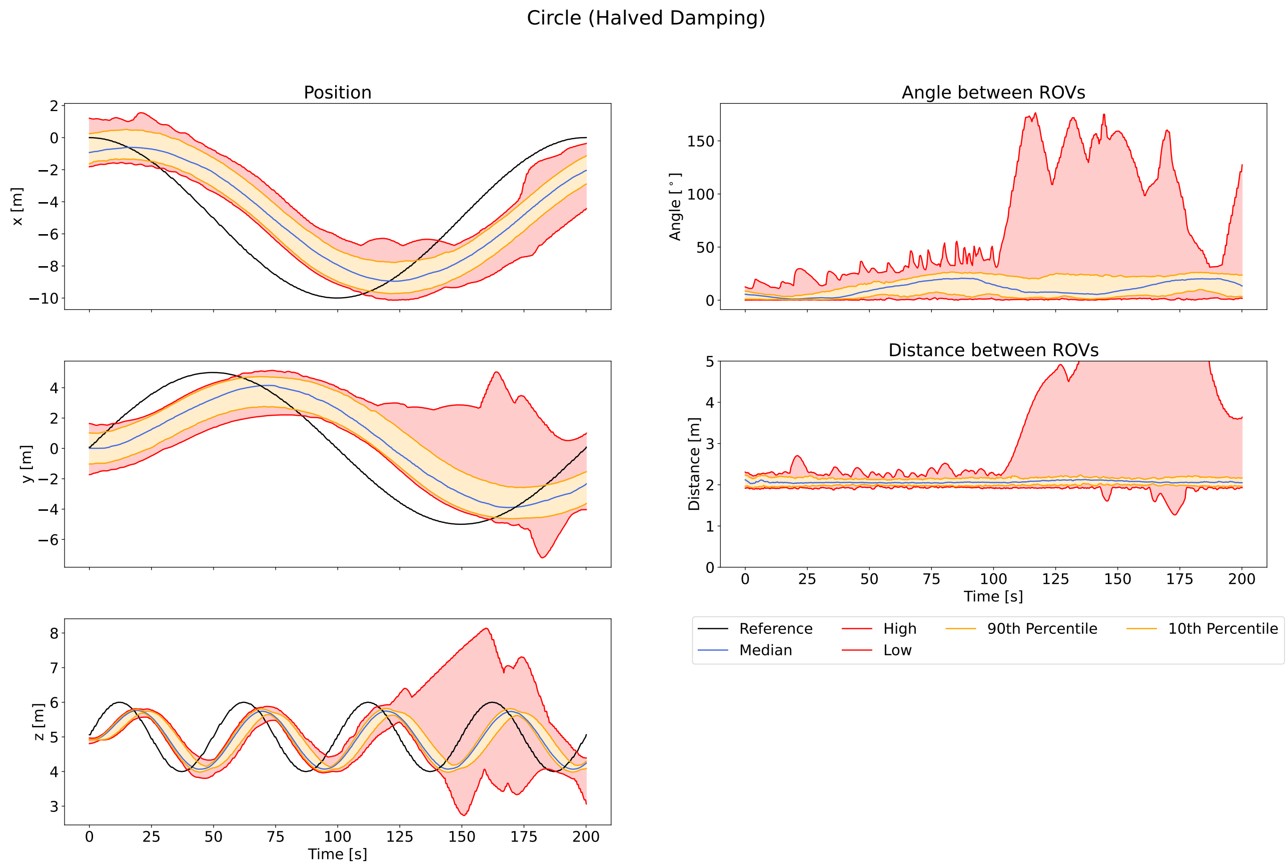


Figure G.21: Circle test with damping halved

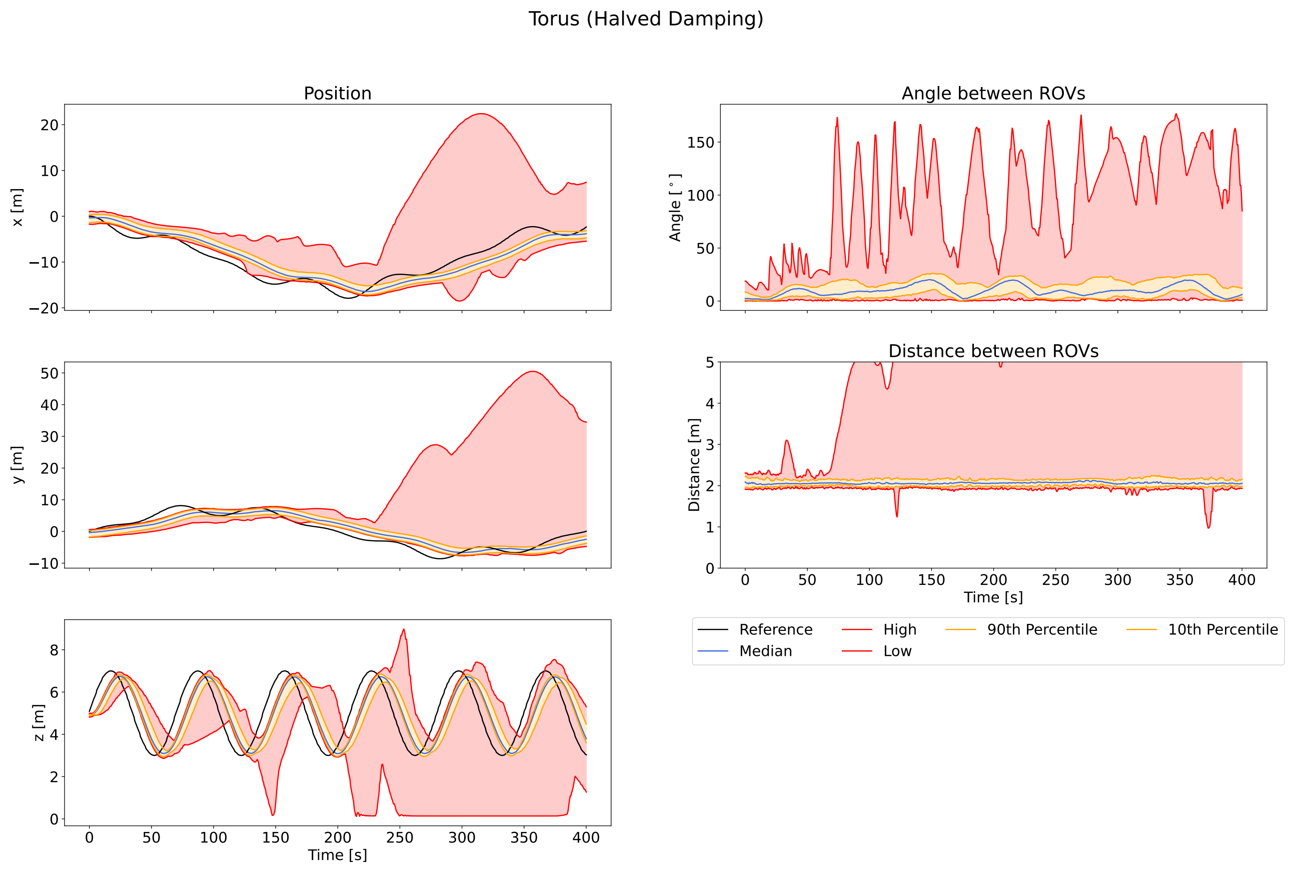


Figure G.22: Torus test with damping halved

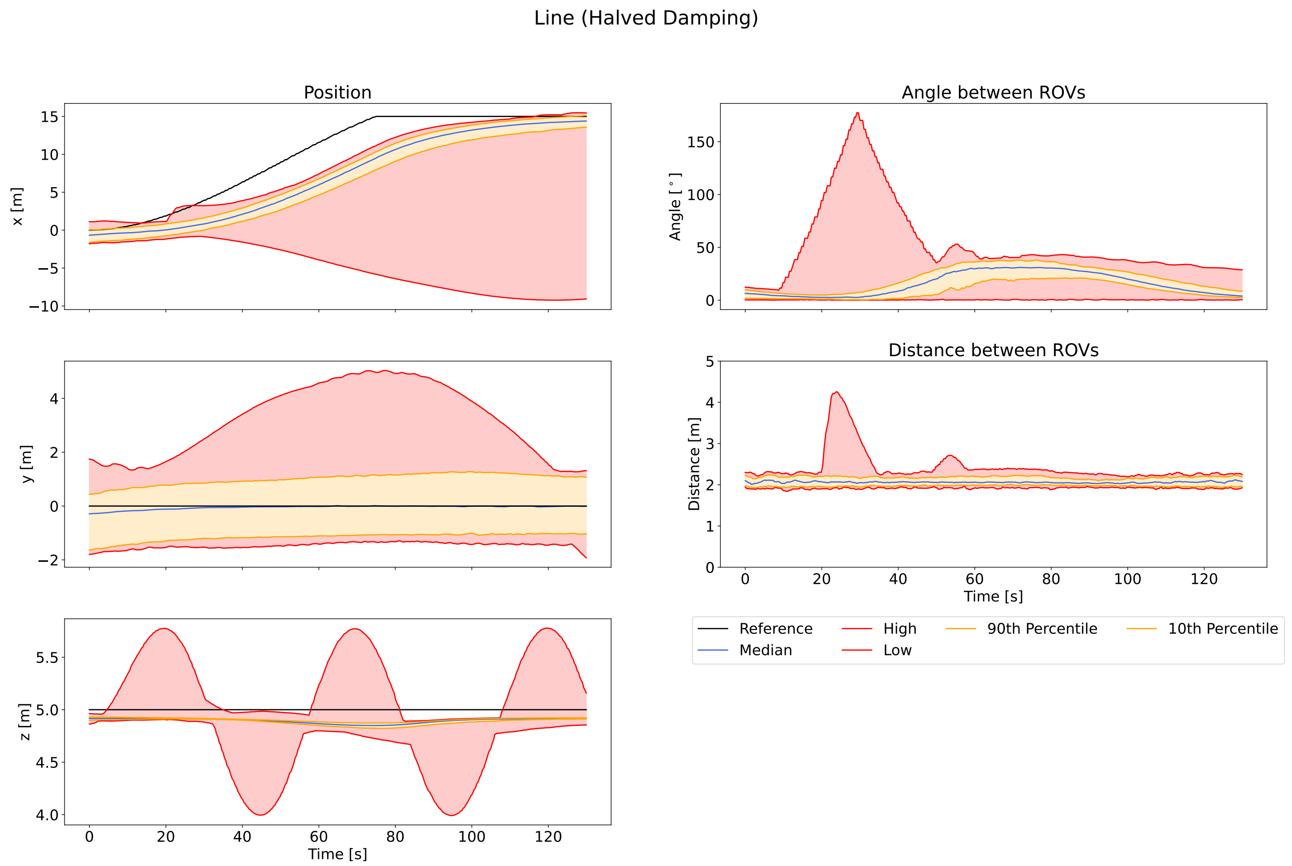


Figure G.23: Line test with damping halved

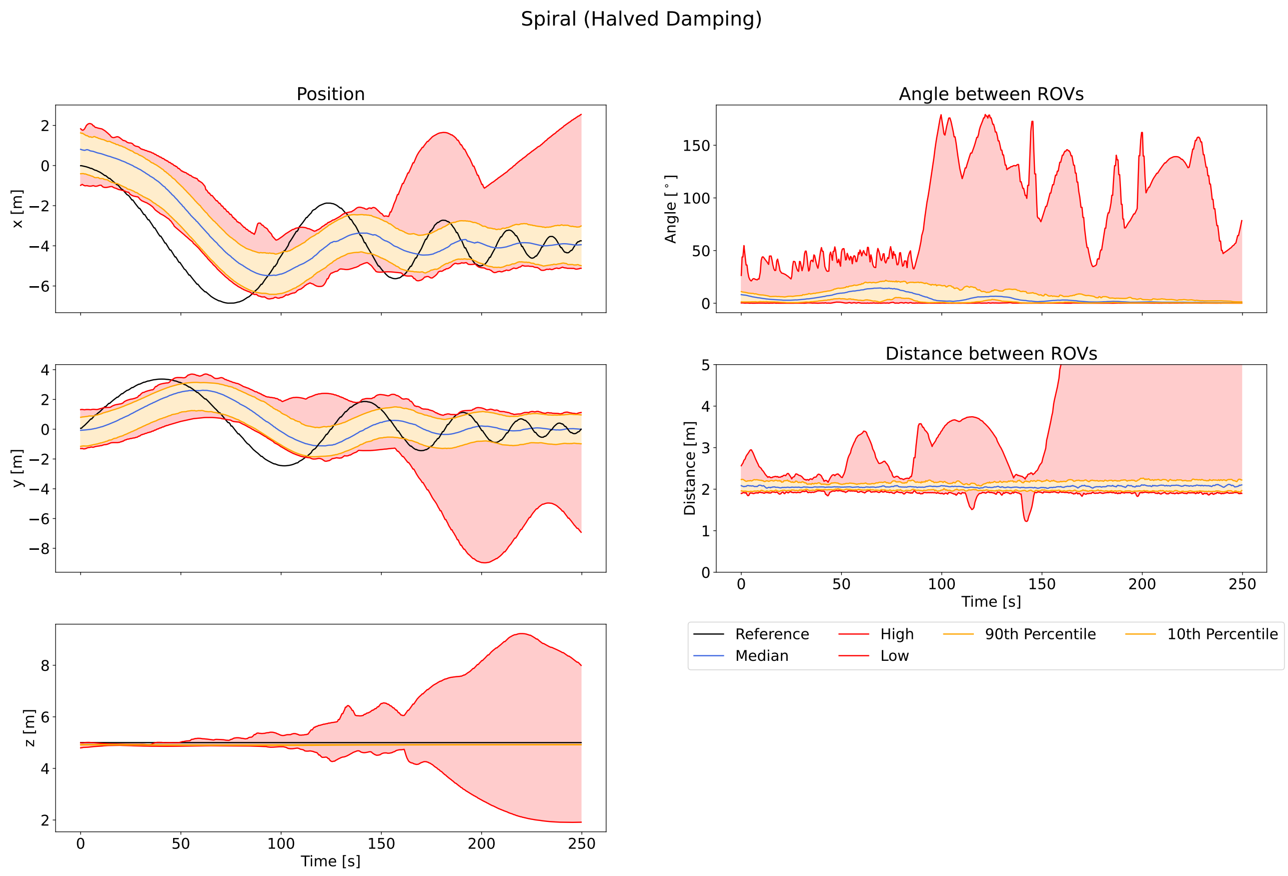


Figure G.24: Spiral test with damping halved

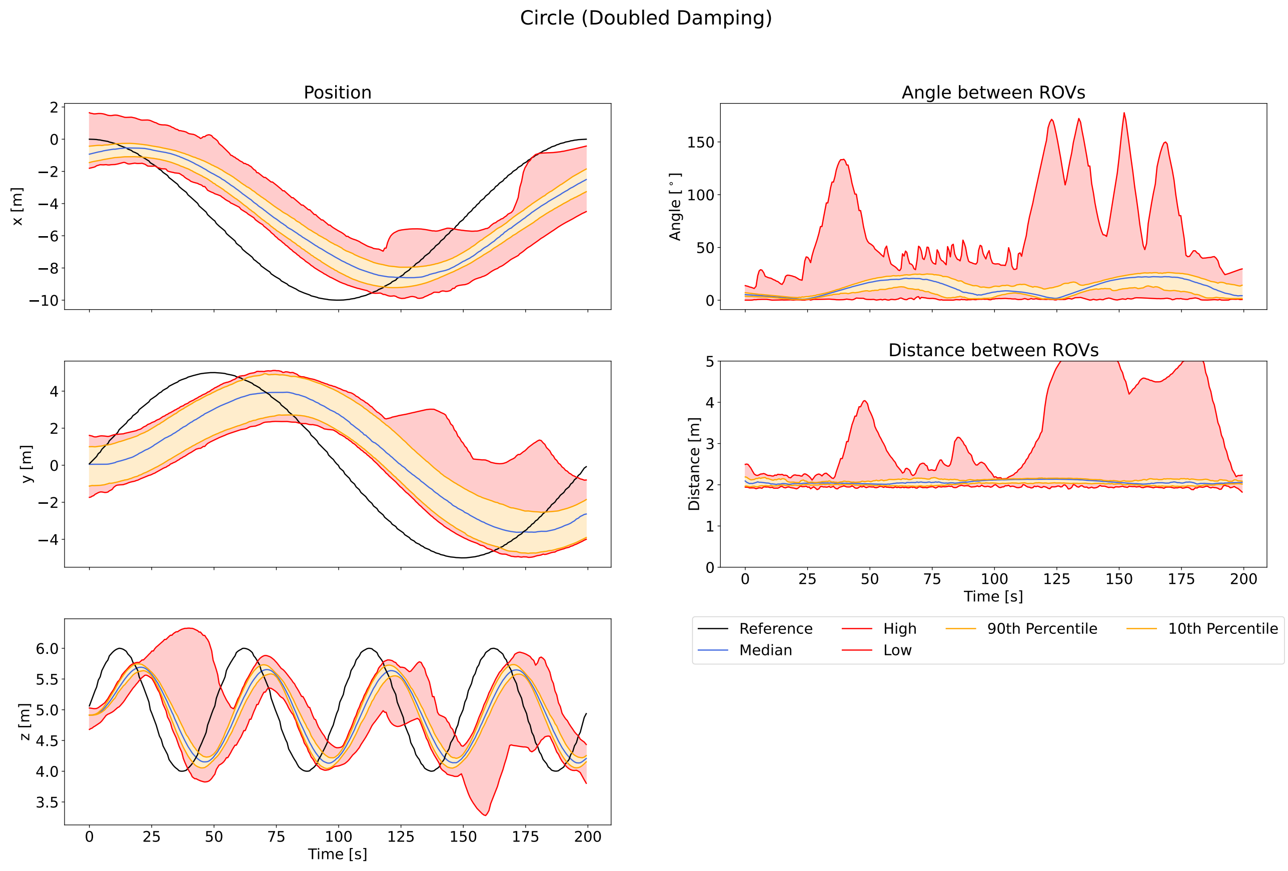


Figure G.25: Circle test with damping doubled

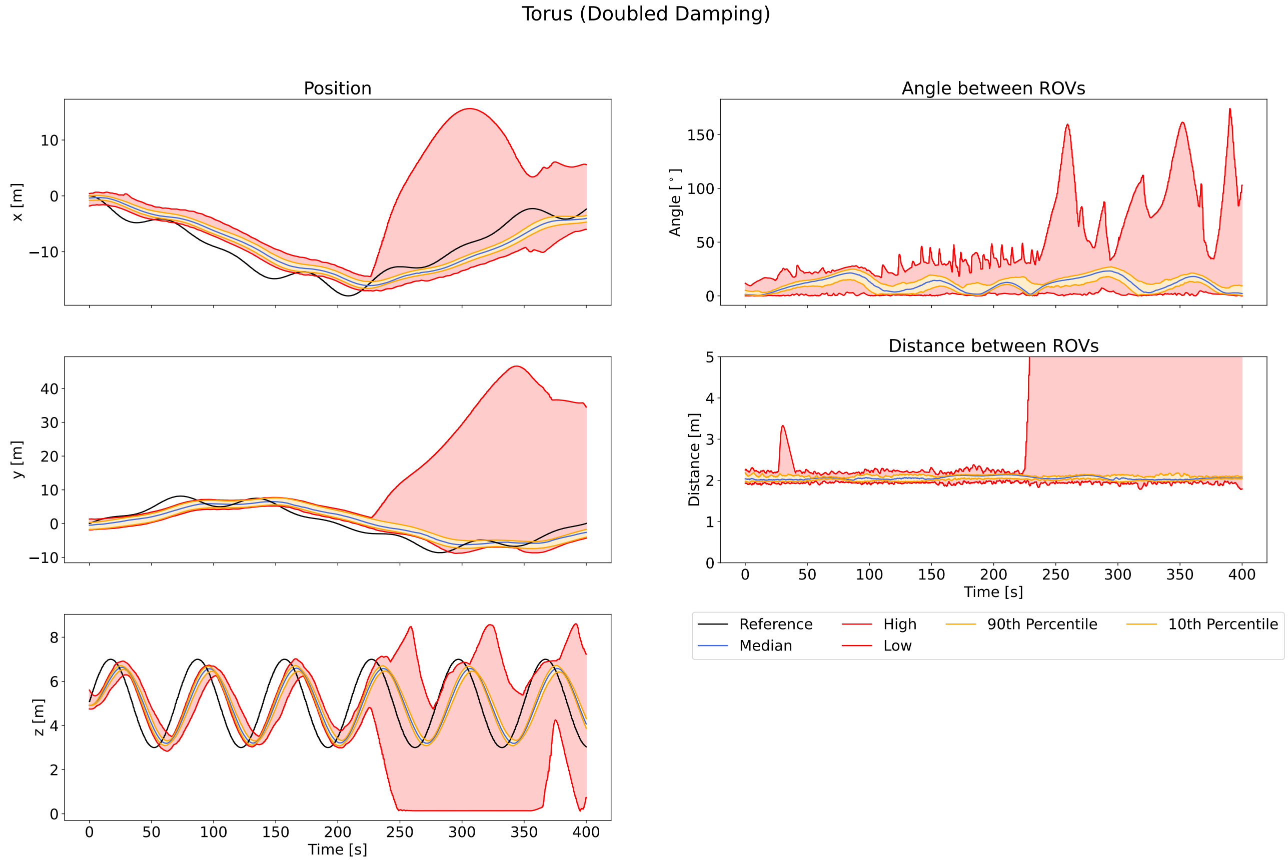


Figure G.26: Torus test with damping doubled

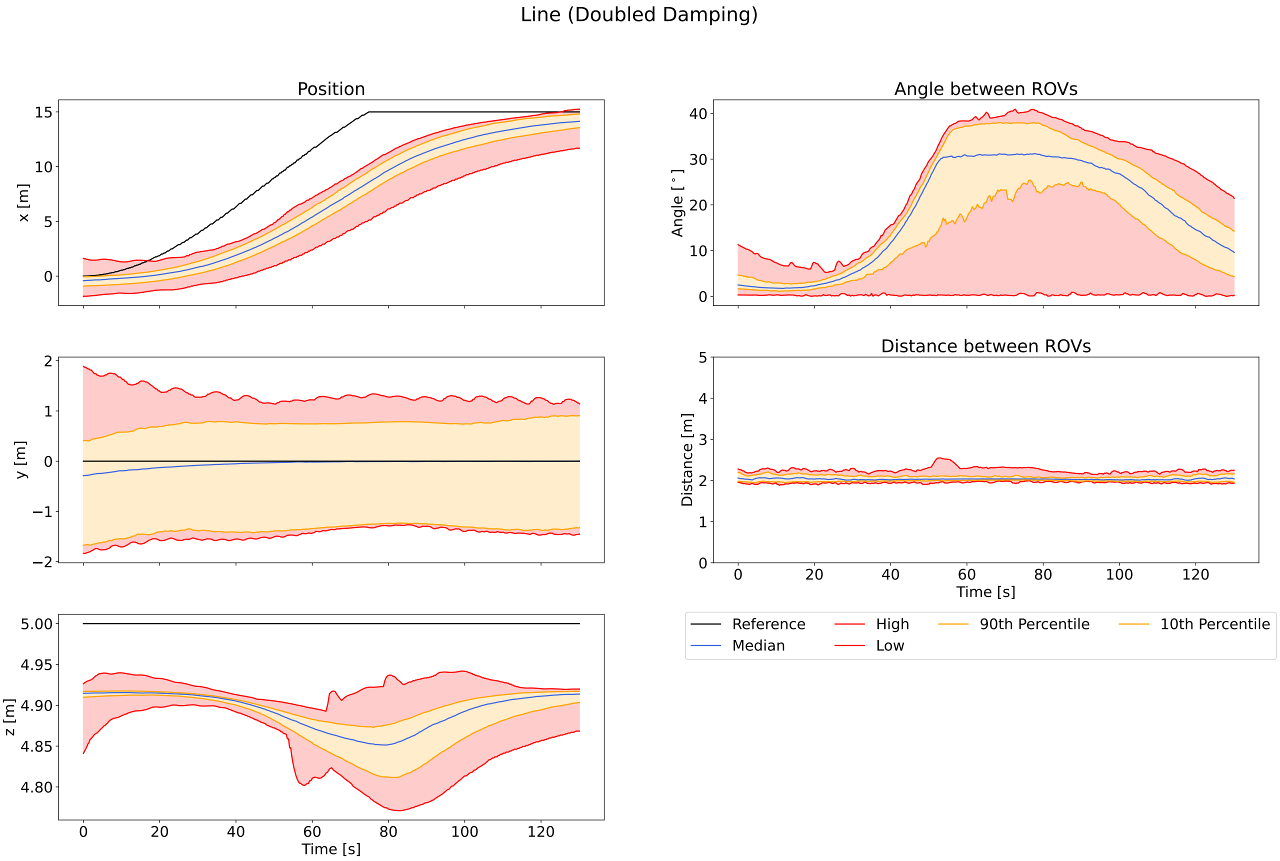


Figure G.27: Line test with damping doubled

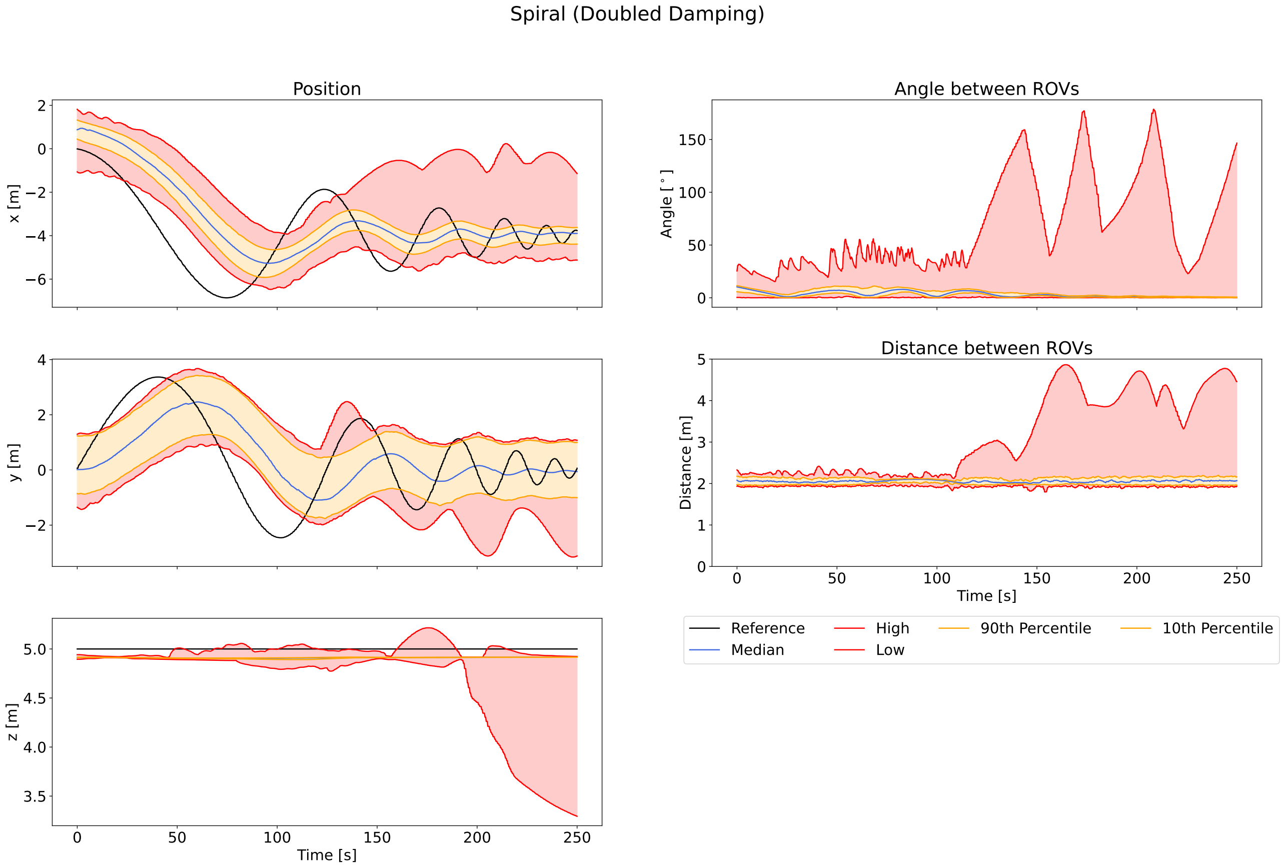


Figure G.28: Spiral test with damping doubled

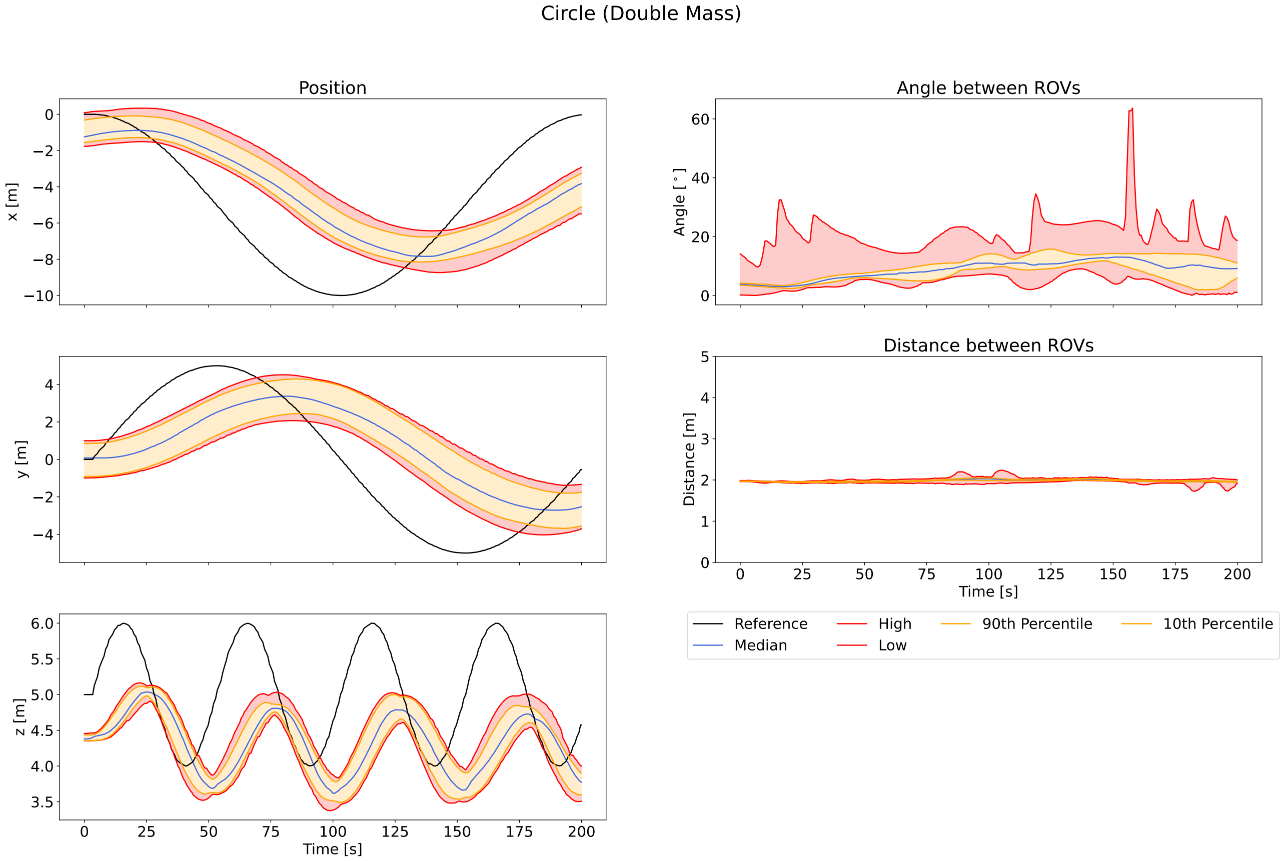


Figure G.29: Circle test with doubled mass

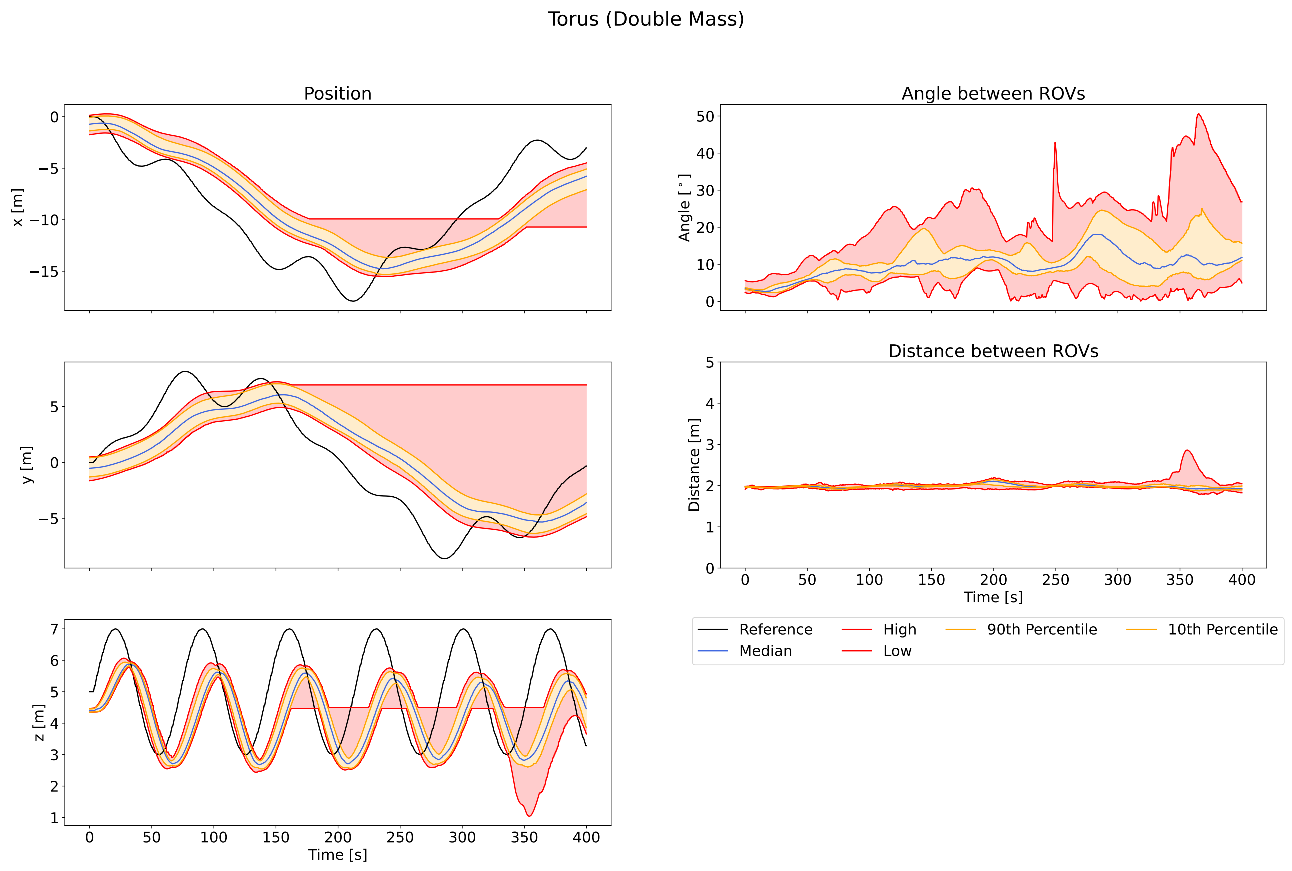


Figure G.30: Torus test with doubled mass

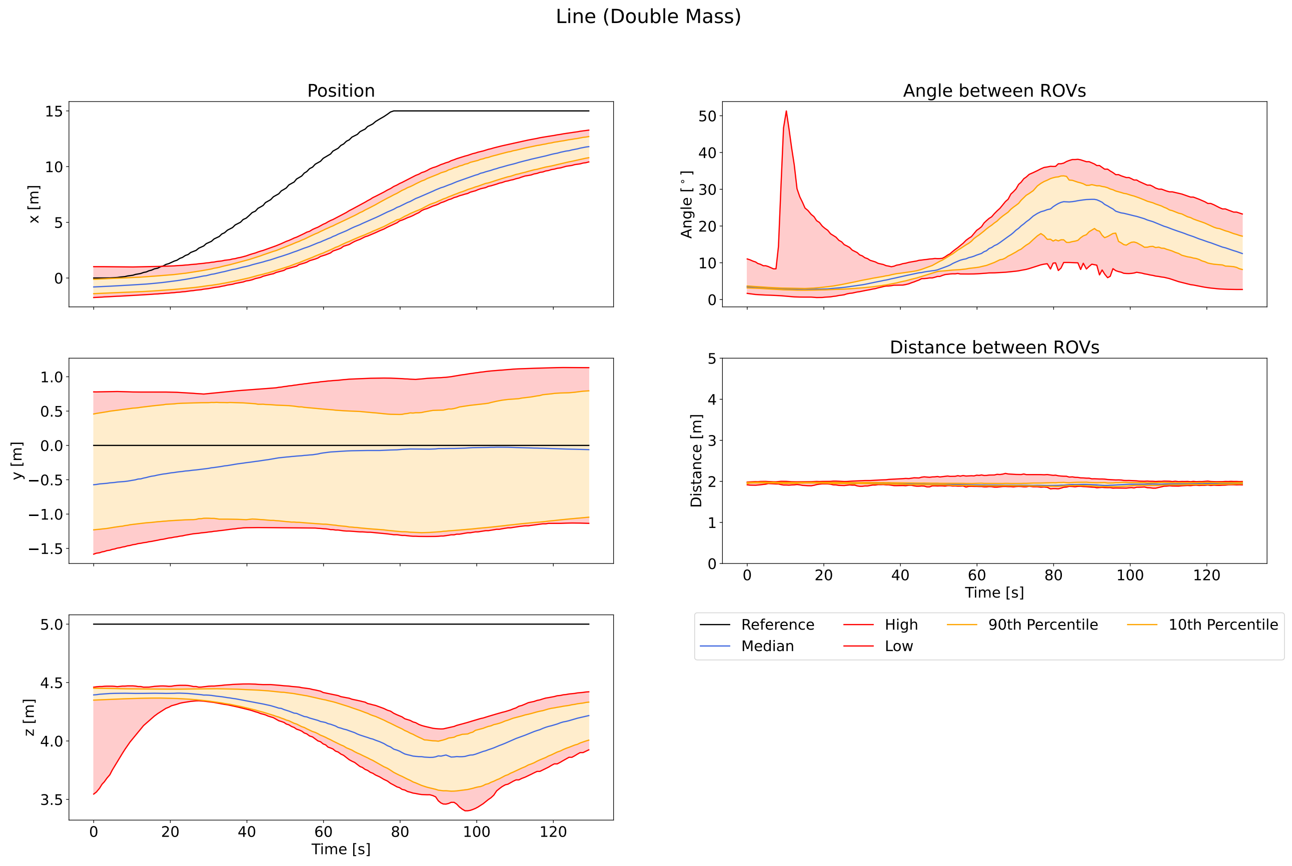


Figure G.31: Line test with doubled mass

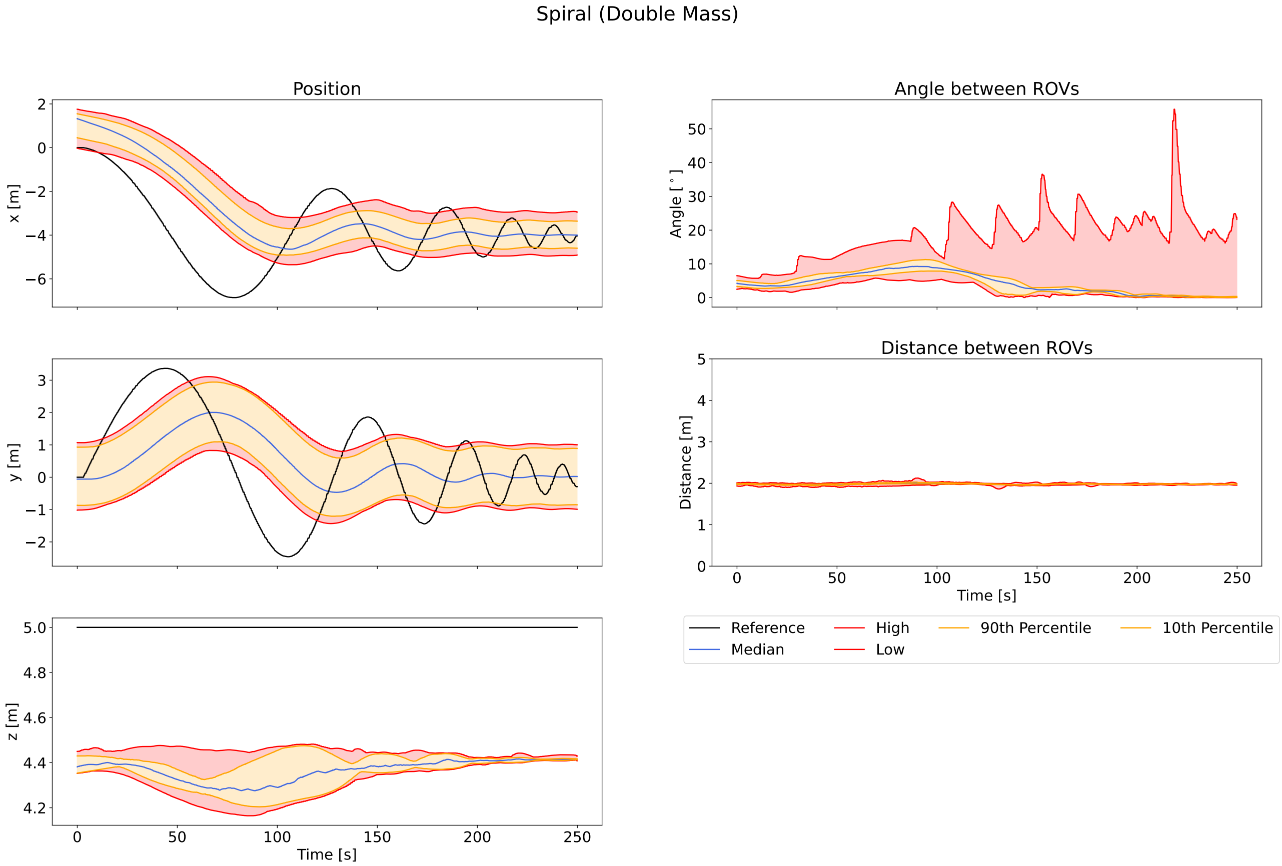


Figure G.32: Spiral test with doubled mass

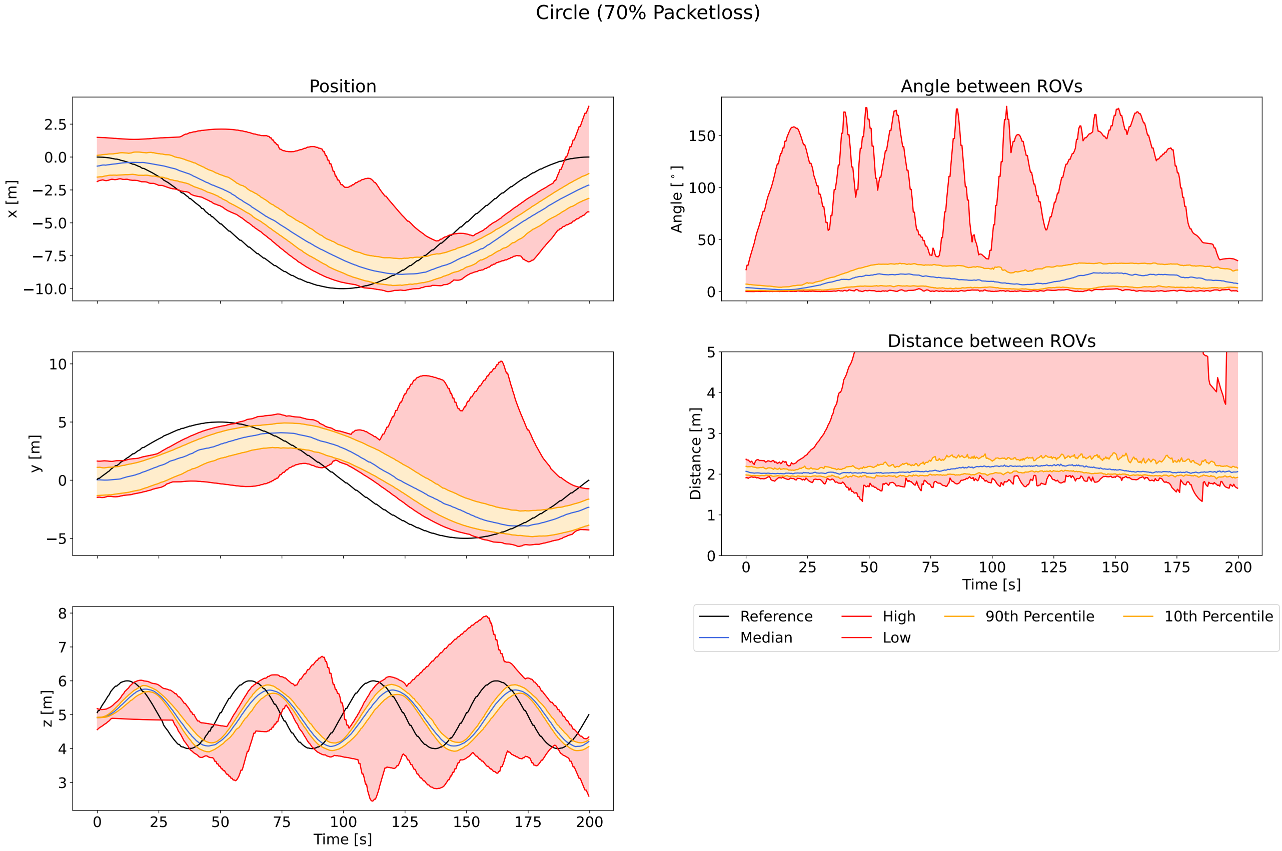


Figure G.33: Circle test with 70% packet loss

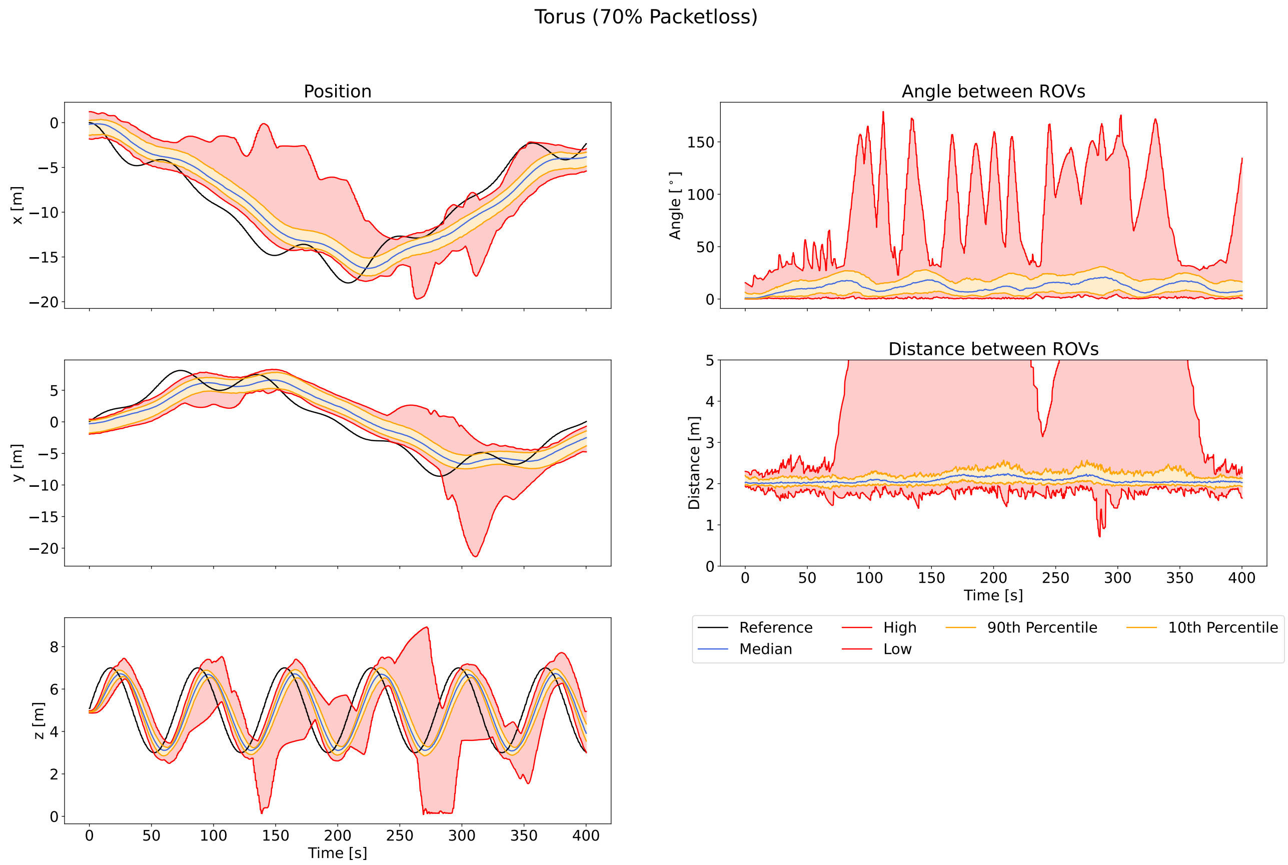


Figure G.34: Torus test with 70% packet loss

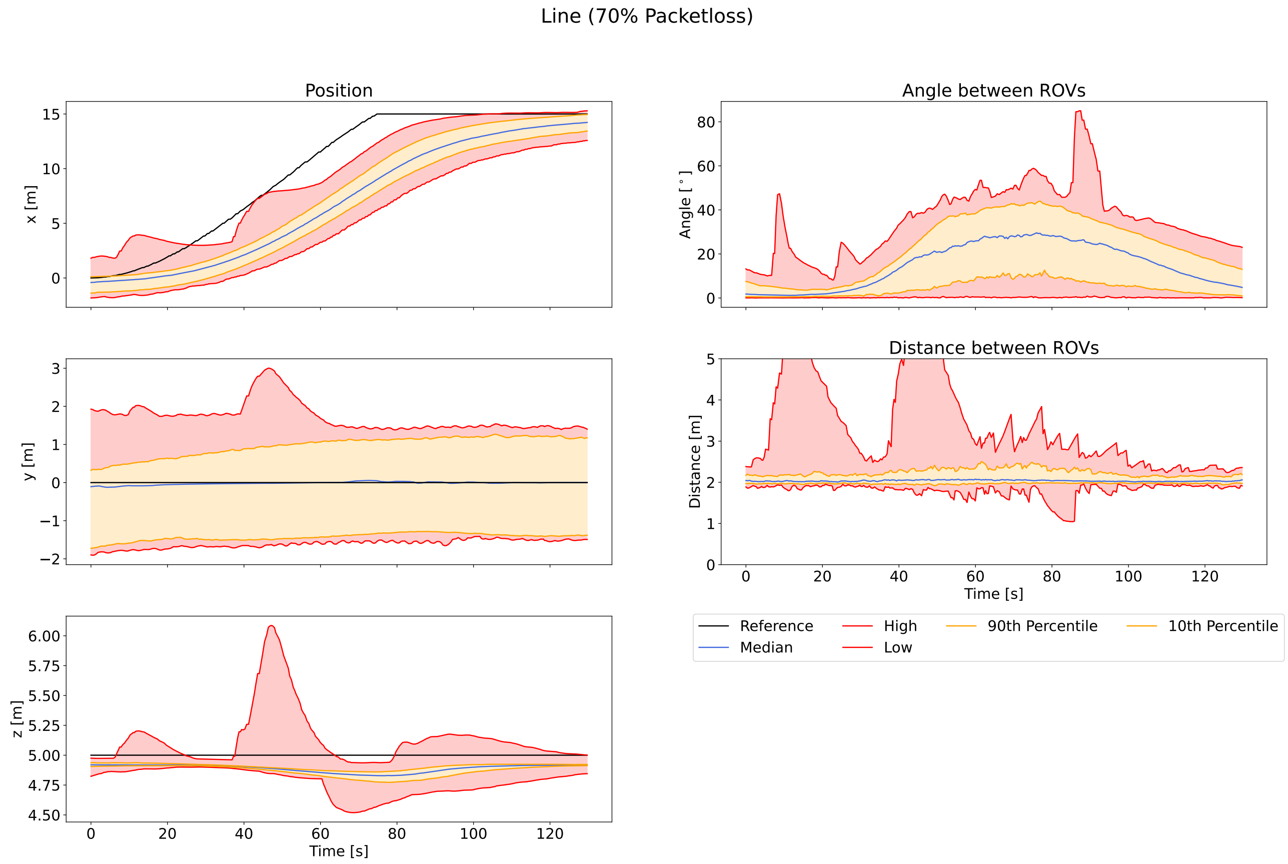


Figure G.35: Line test with 70% packet loss

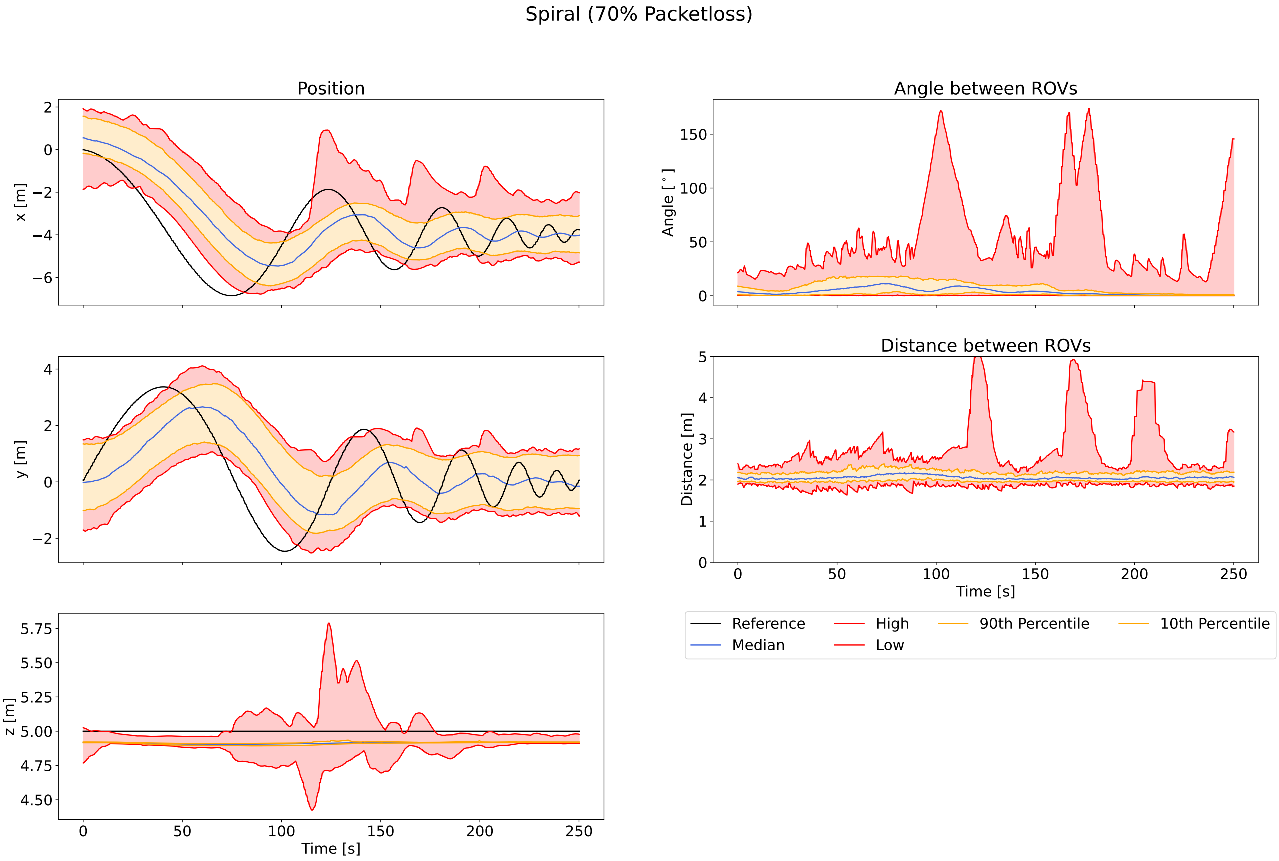


Figure G.36: Spiral test with 70% packet loss

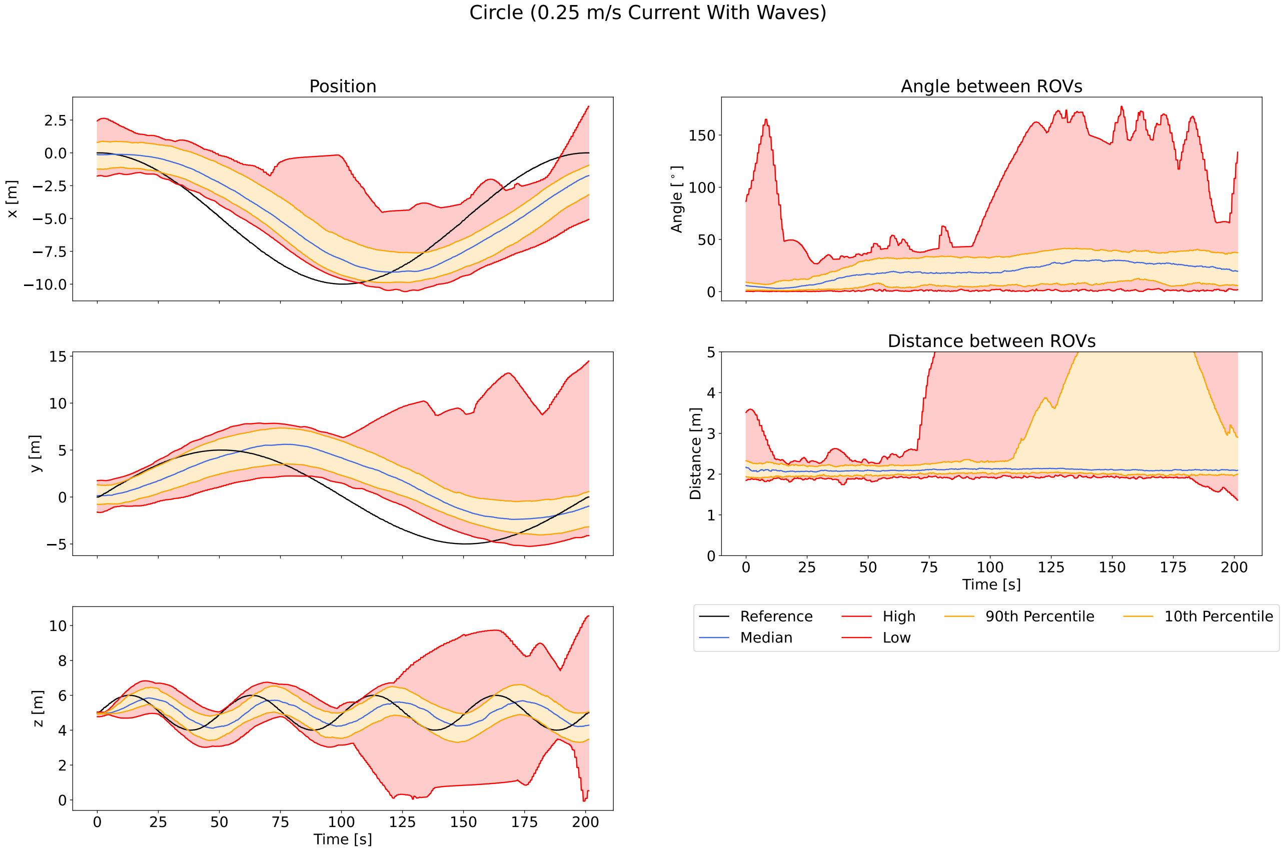


Figure G.37: Circle test with 0.25 m/s current and waves

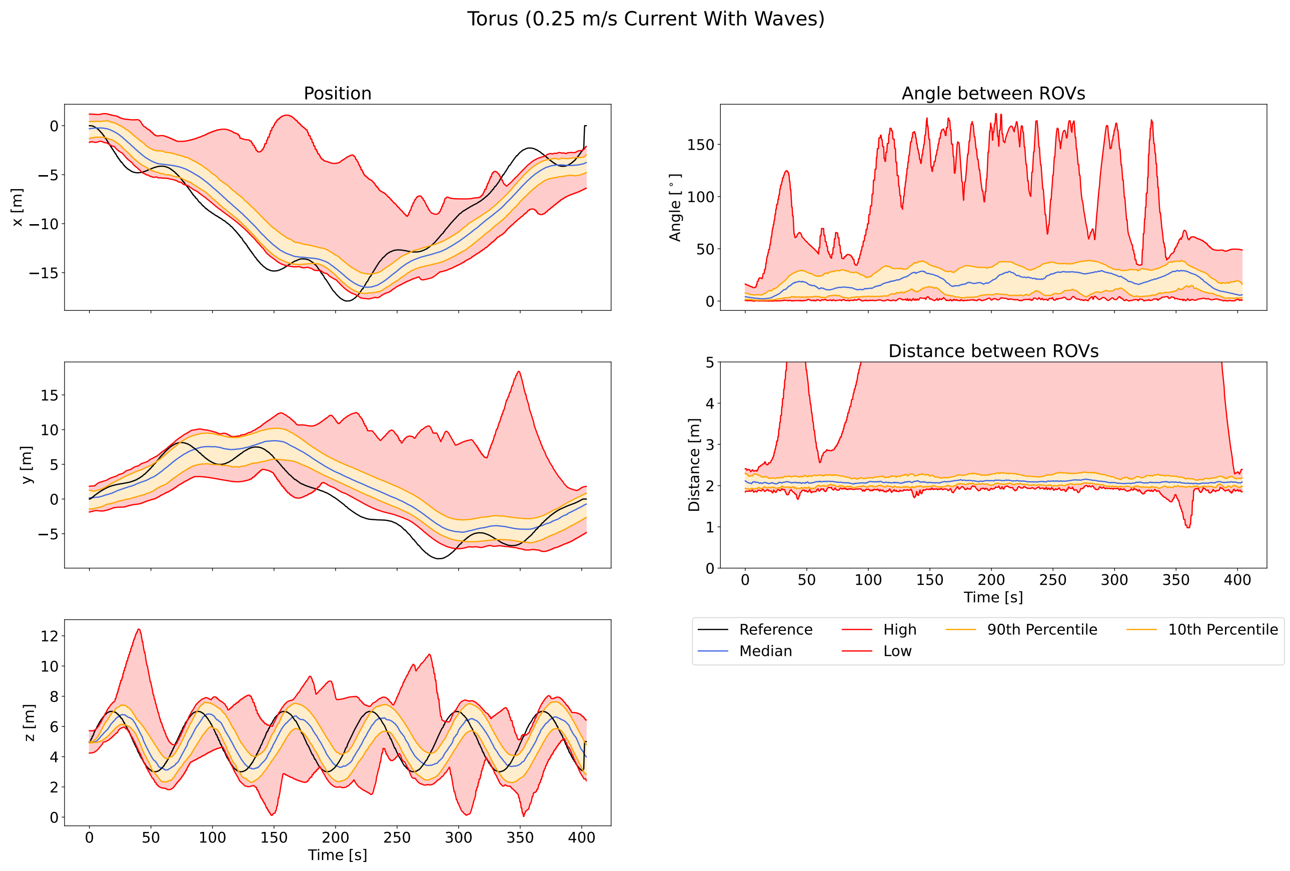


Figure G.38: Torus test with 0.25 m/s current and waves

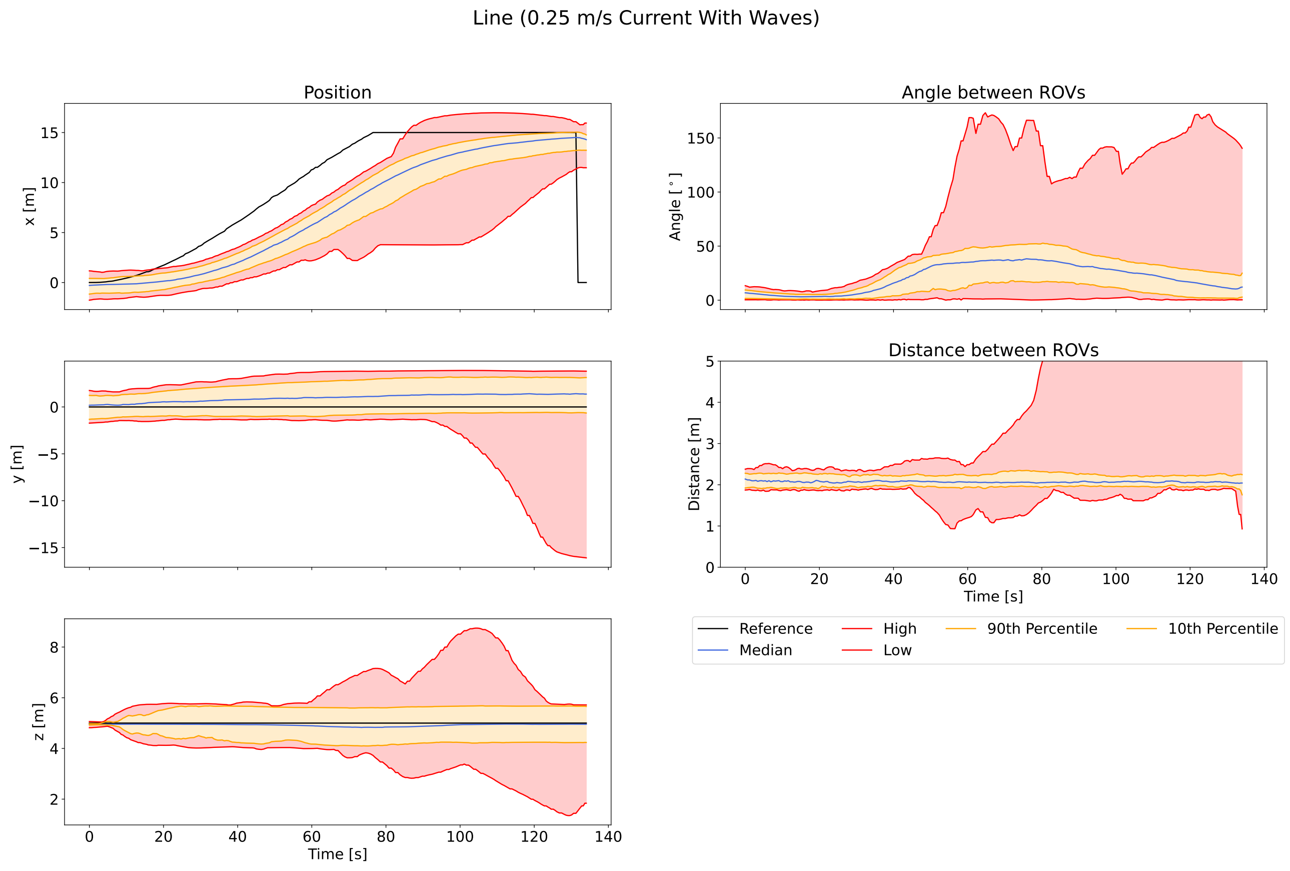


Figure G.39: Line test with 0.25 m/s current and waves

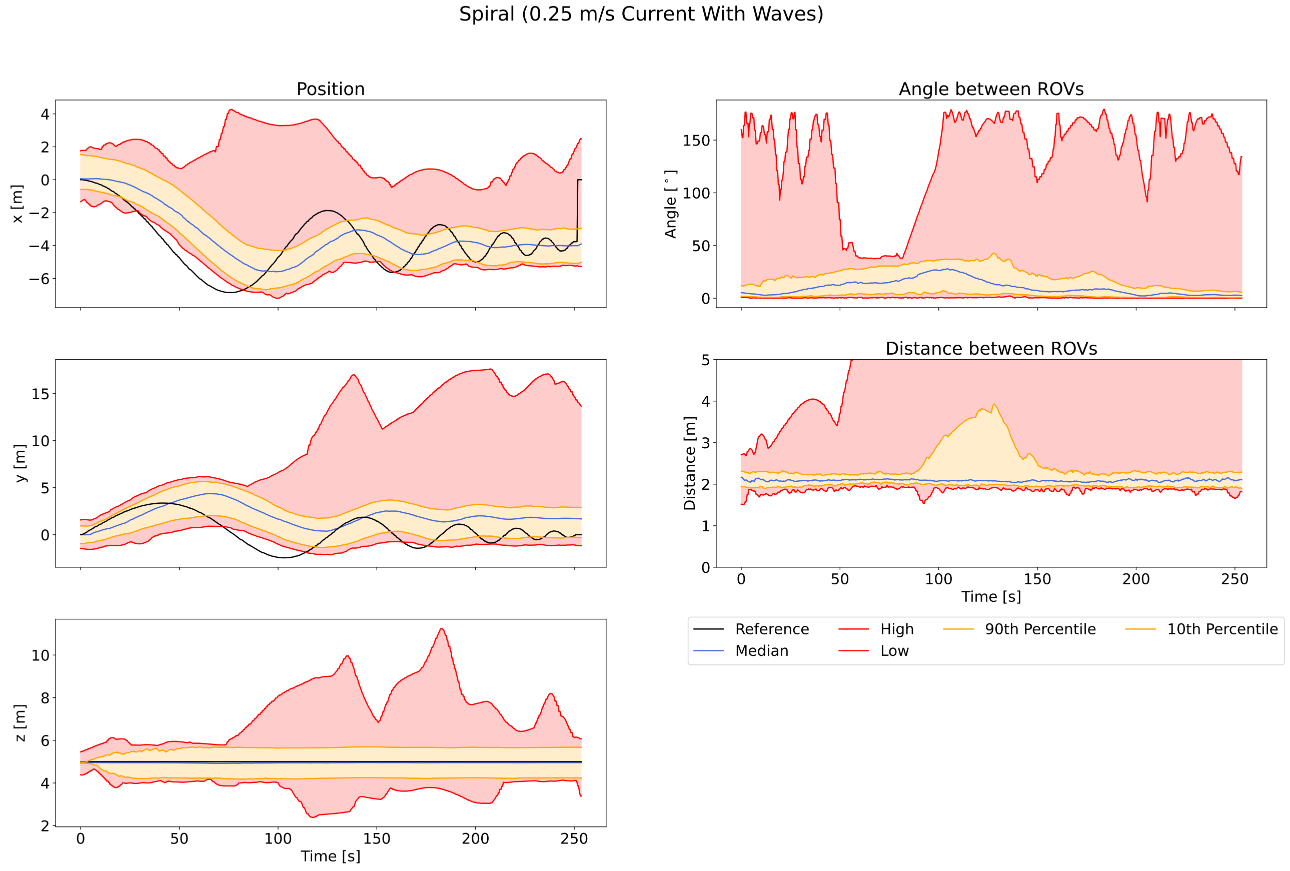


Figure G.40: Spiral test with 0.25 m/s current and waves

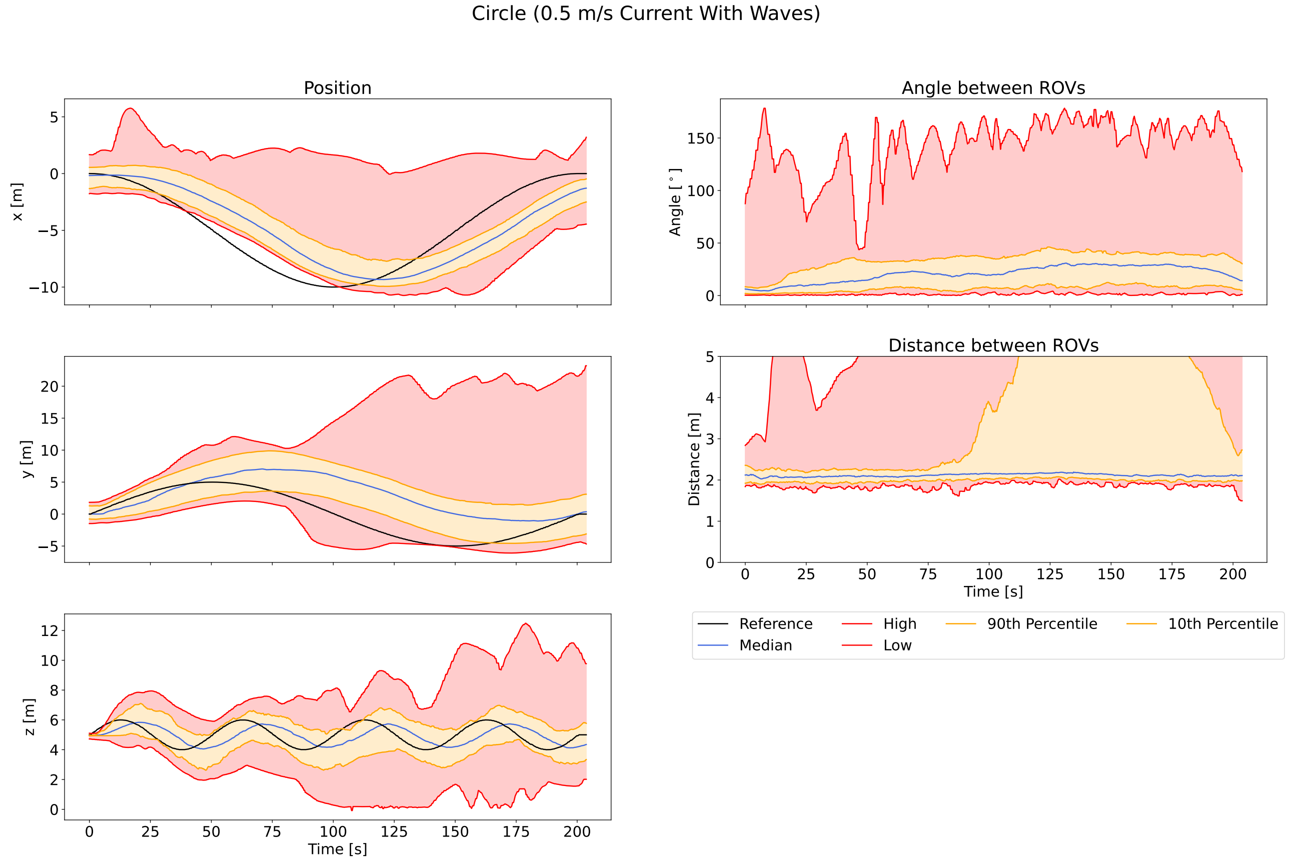


Figure G.41: Circle test with 0.5 m/s current and waves

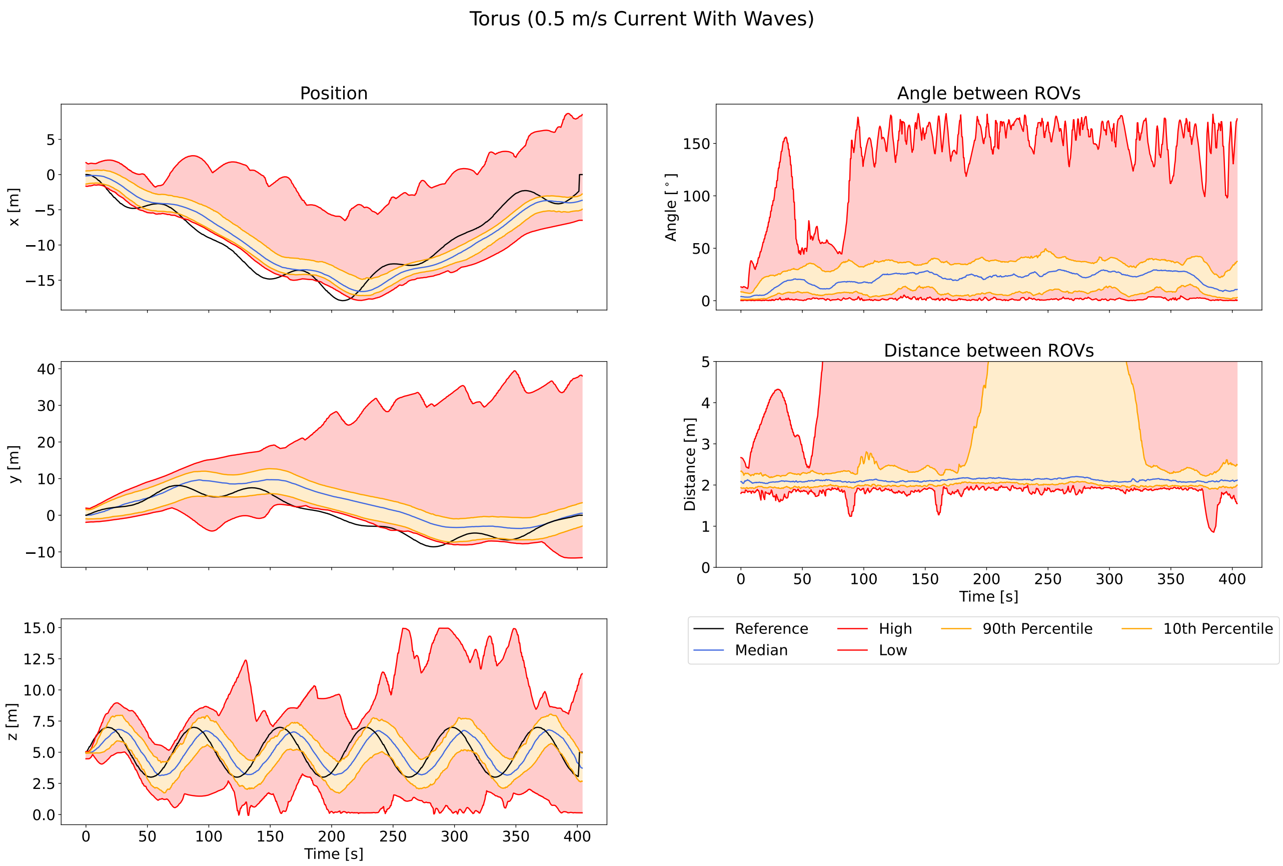


Figure G.42: Torus test with 0.5 m/s current and waves

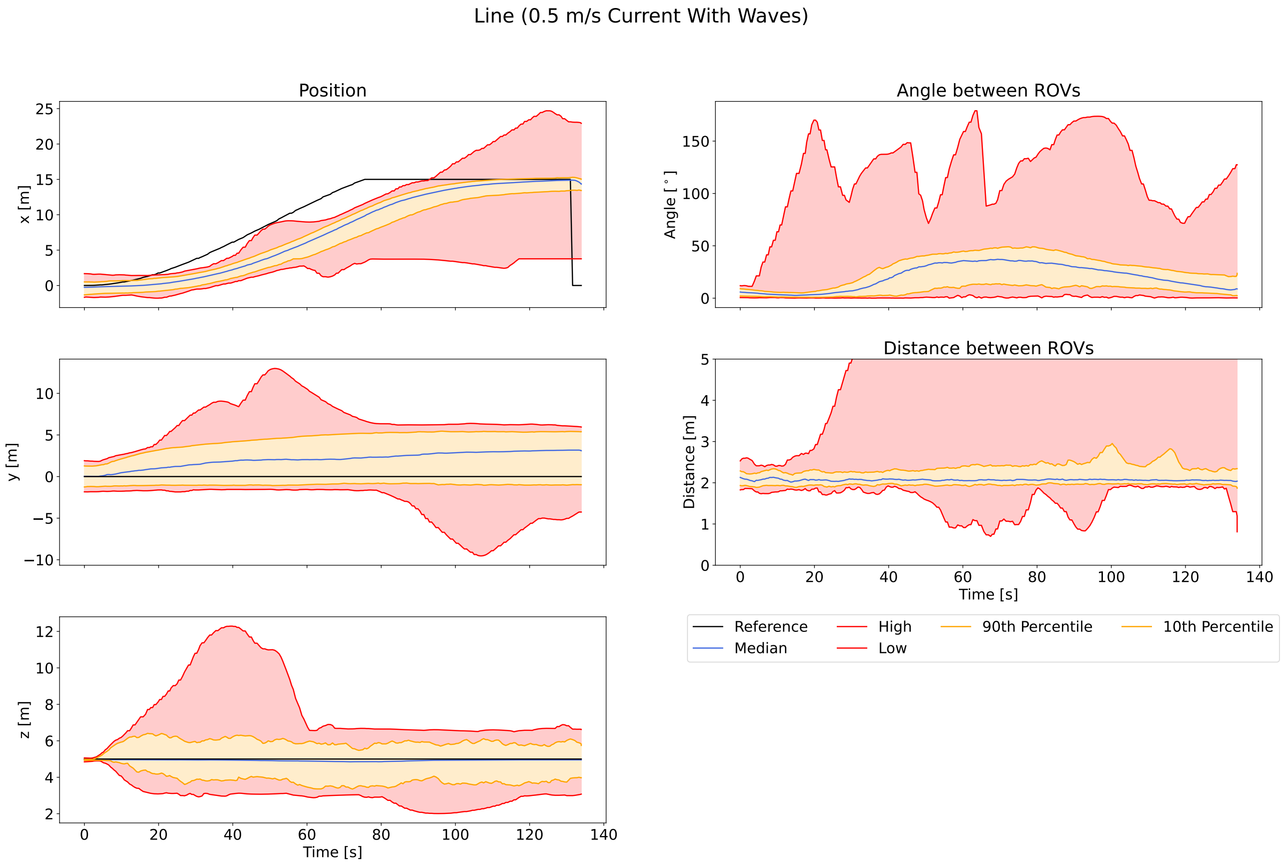


Figure G.43: Line test with 0.5 m/s current and waves

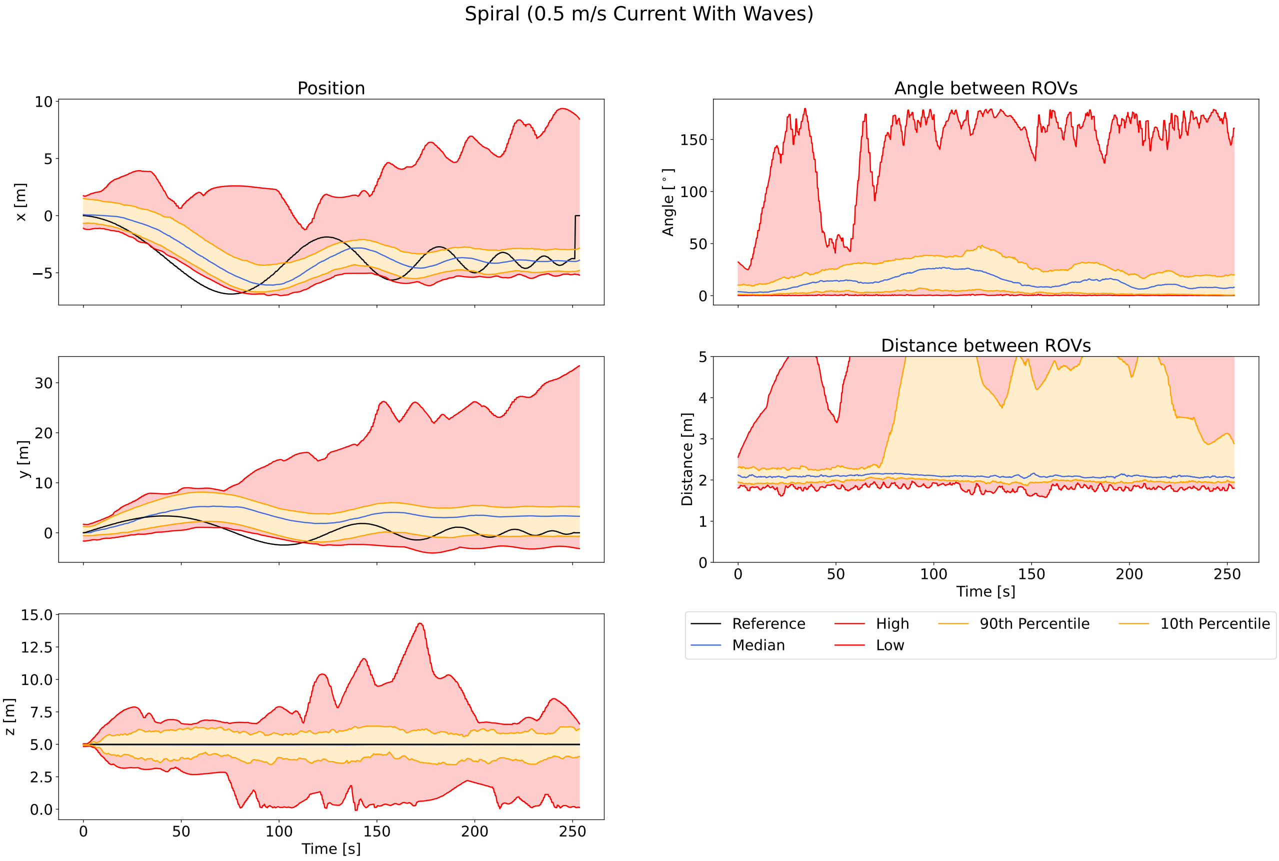


Figure G.44: Spiral test with 0.5 m/s current and waves

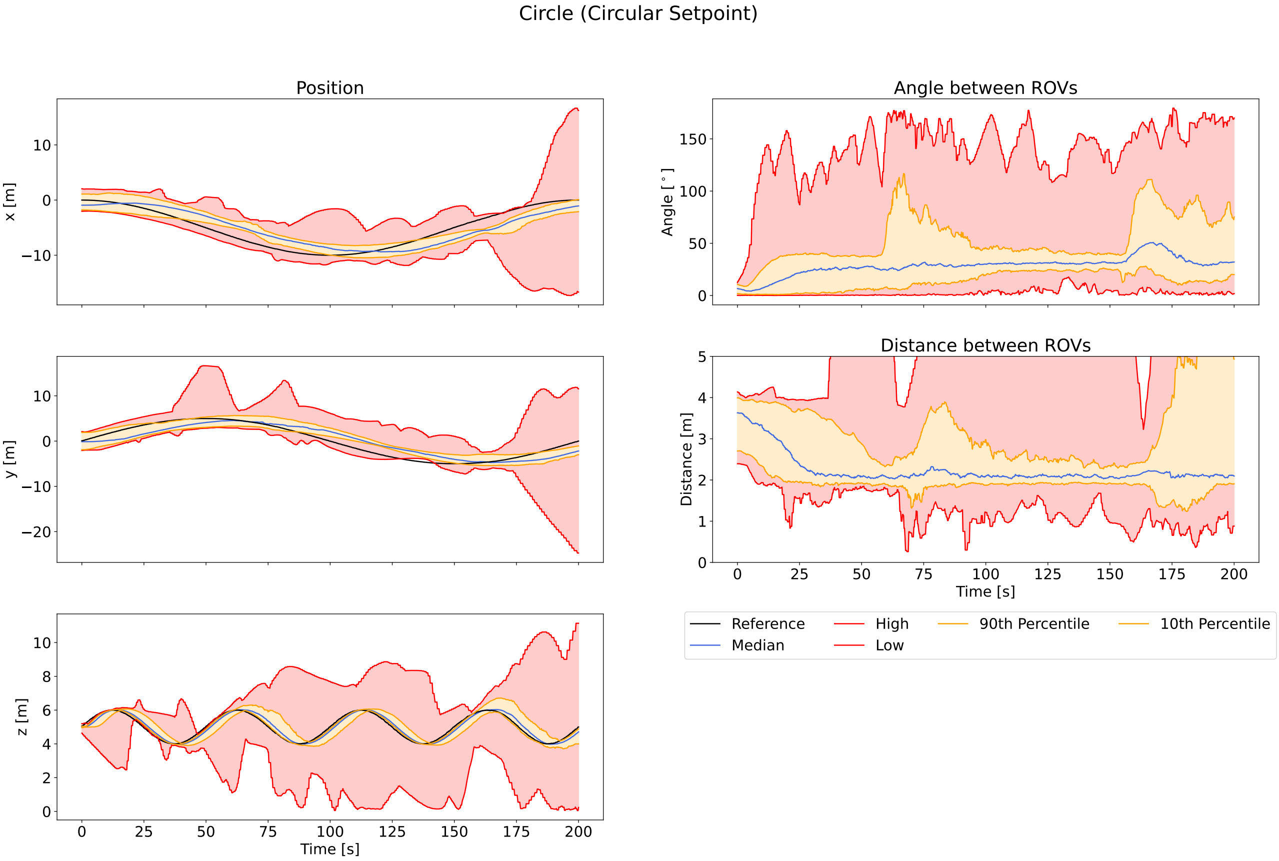


Figure G.45: Circle test with circular setpoint

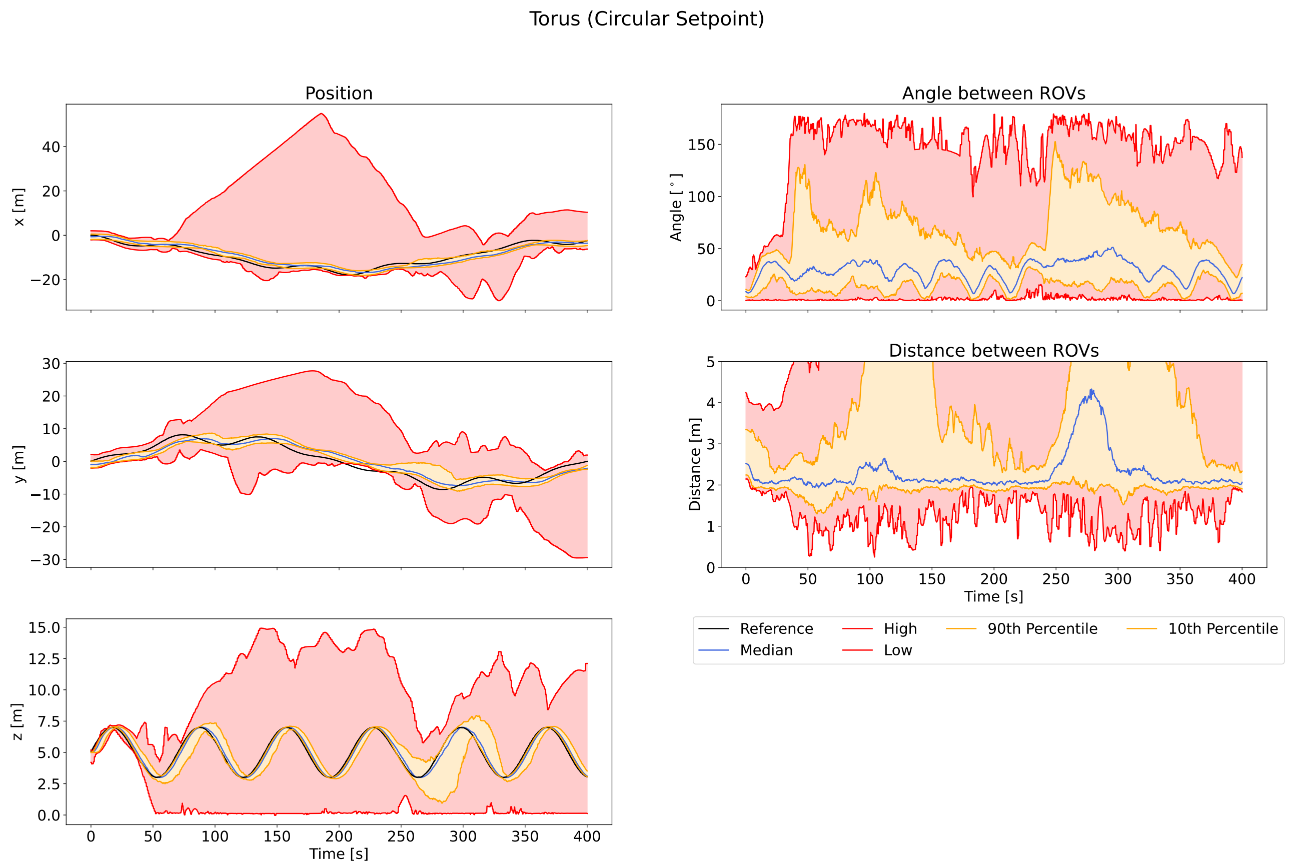


Figure G.46: Torus test with circular setpoint

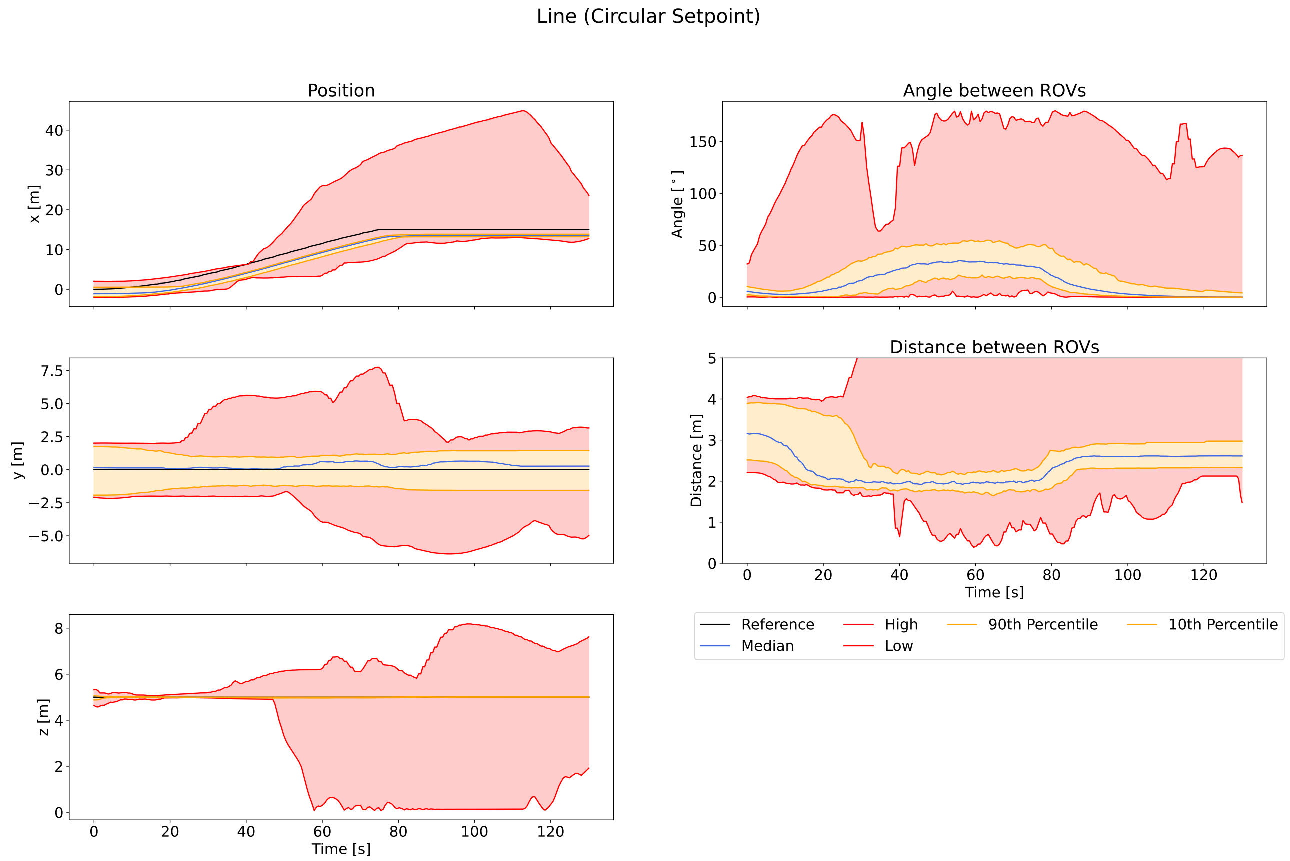


Figure G.47: Line test with circular setpoint

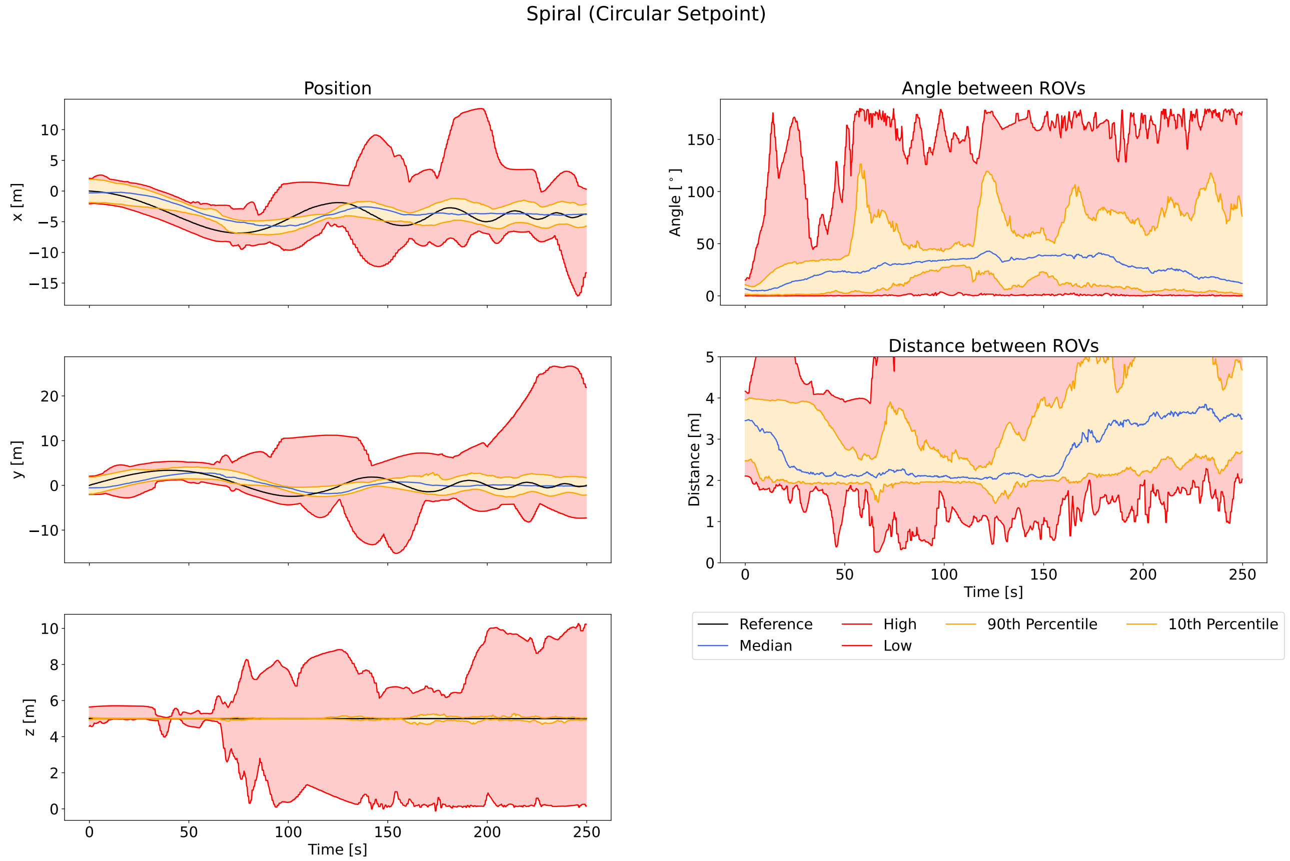


Figure G.48: Spiral test with circular setpoint

Appendix H

Showcase of Median Values from Tests With Disturbances

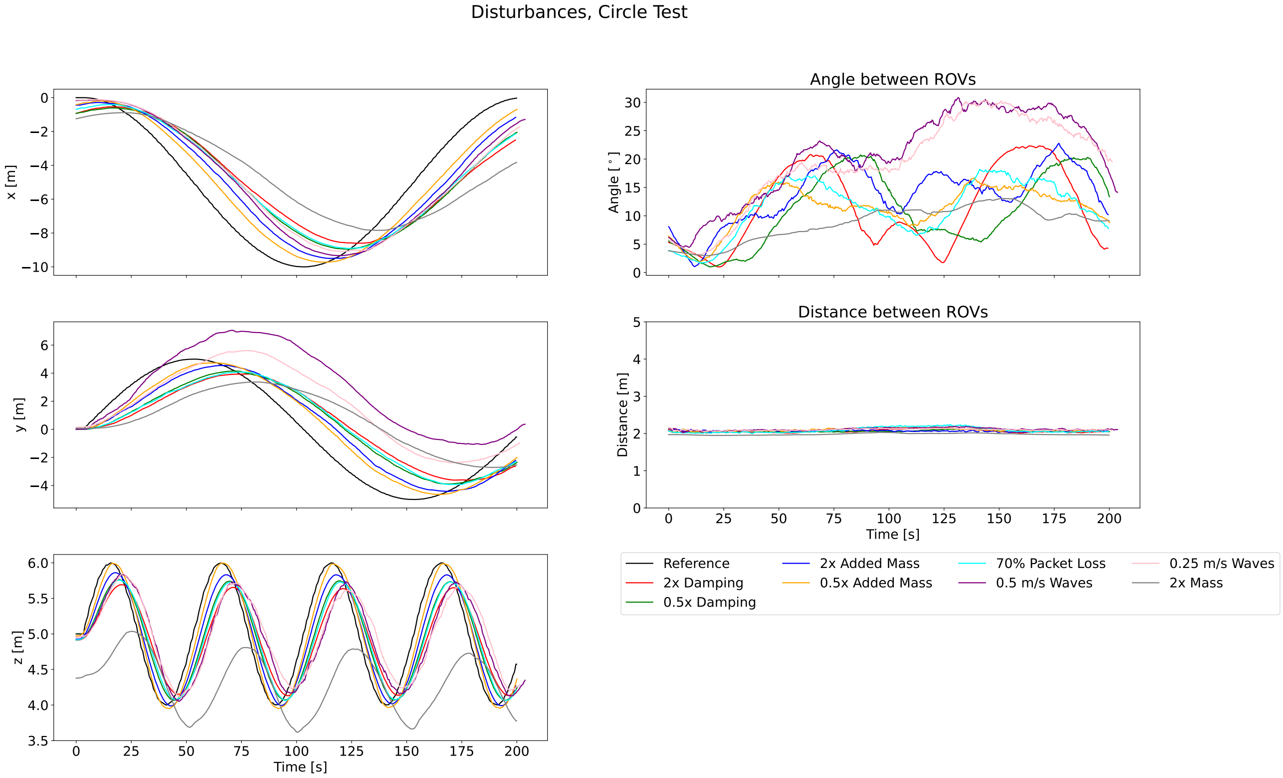


Figure H.1: Showcase of disturbances, circle test

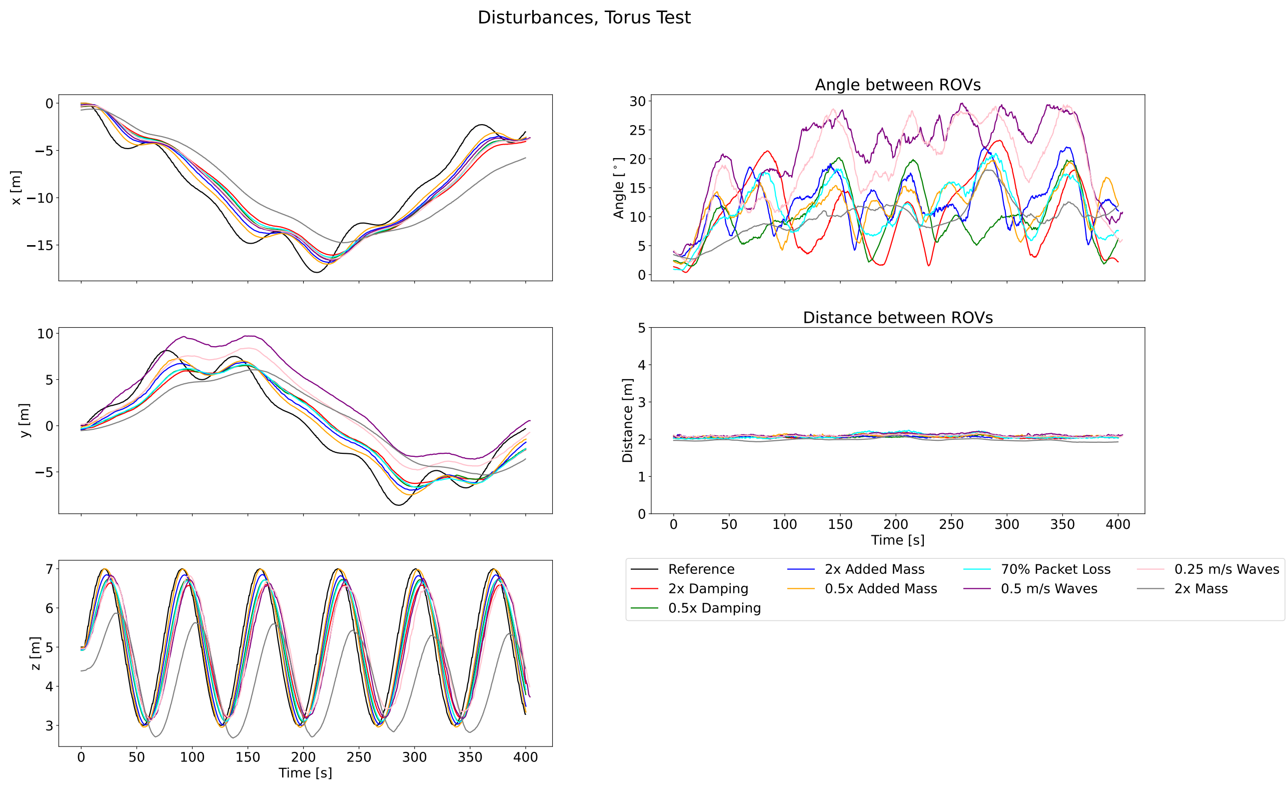


Figure H.2: Showcase of disturbances, torus test

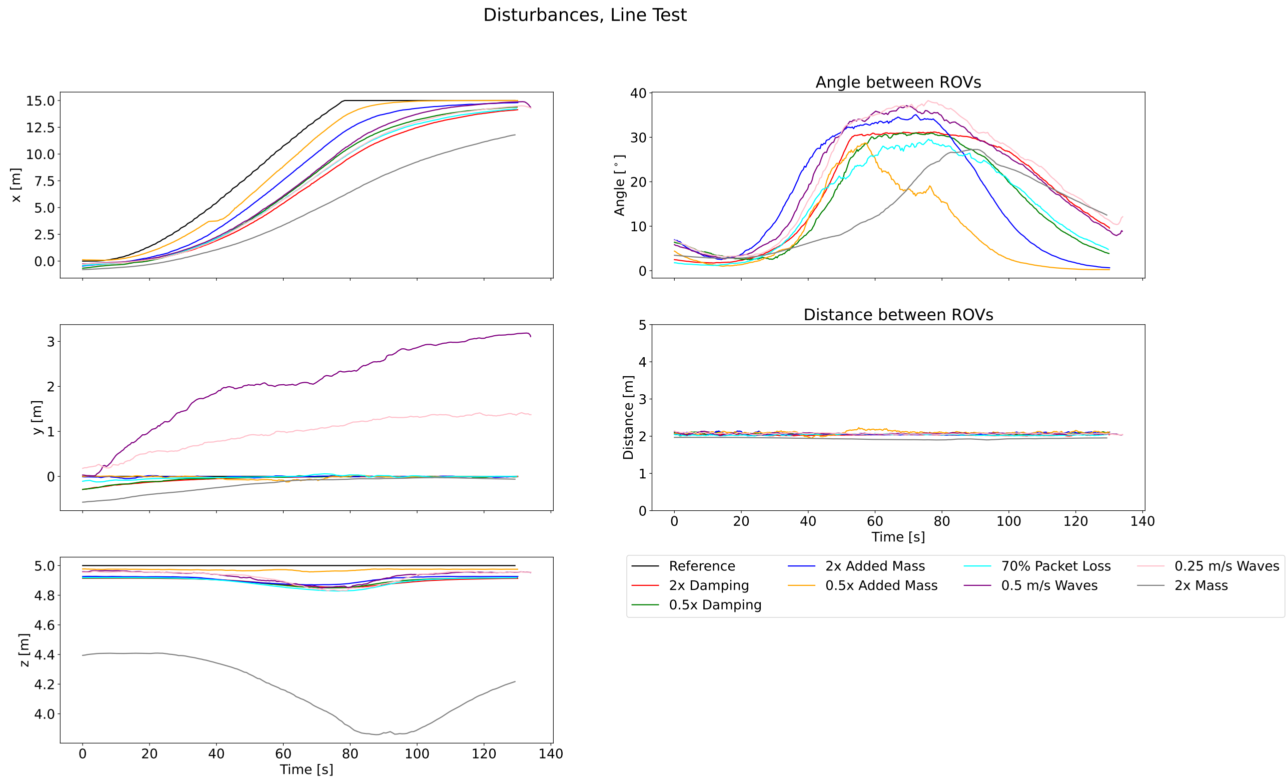


Figure H.3: Showcase of disturbances, line test

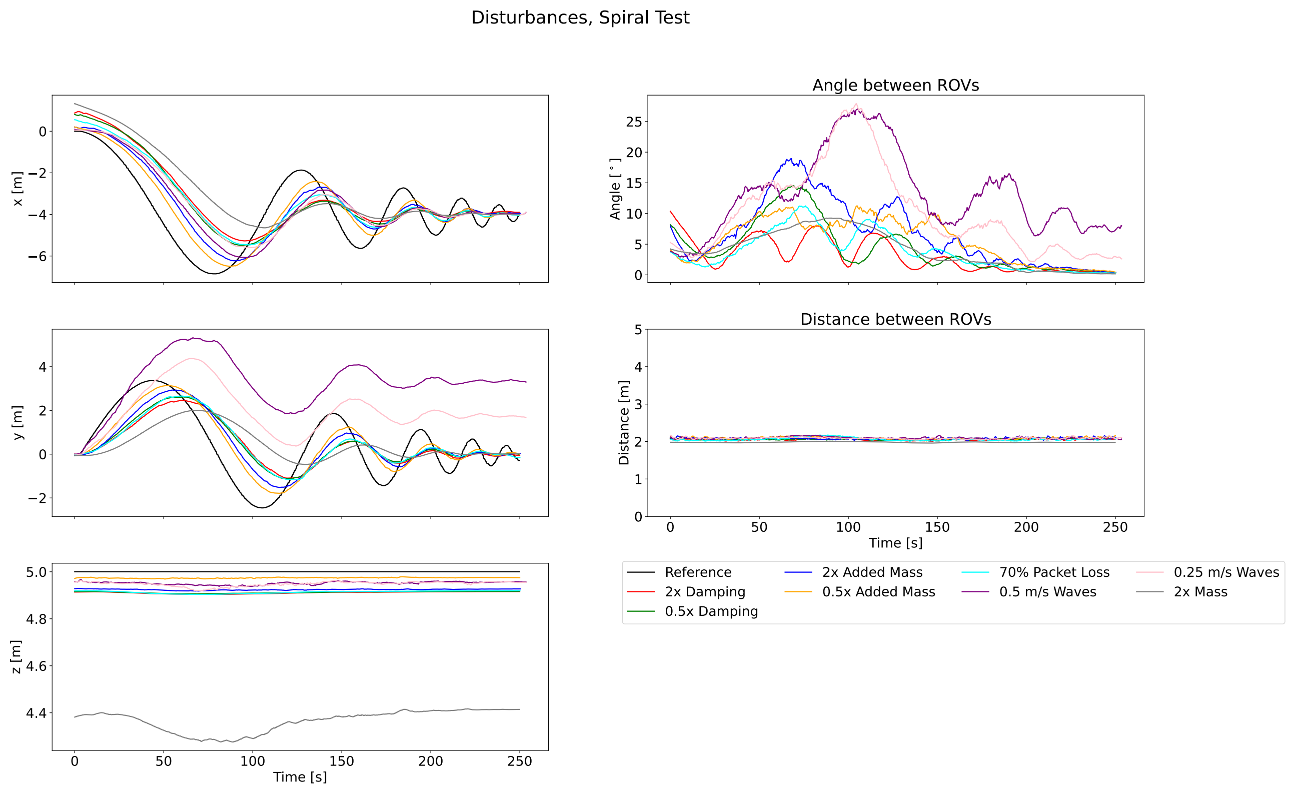


Figure H.4: Showcase of disturbances, spiral test

Appendix I

Comparison Between Gazebo and Python

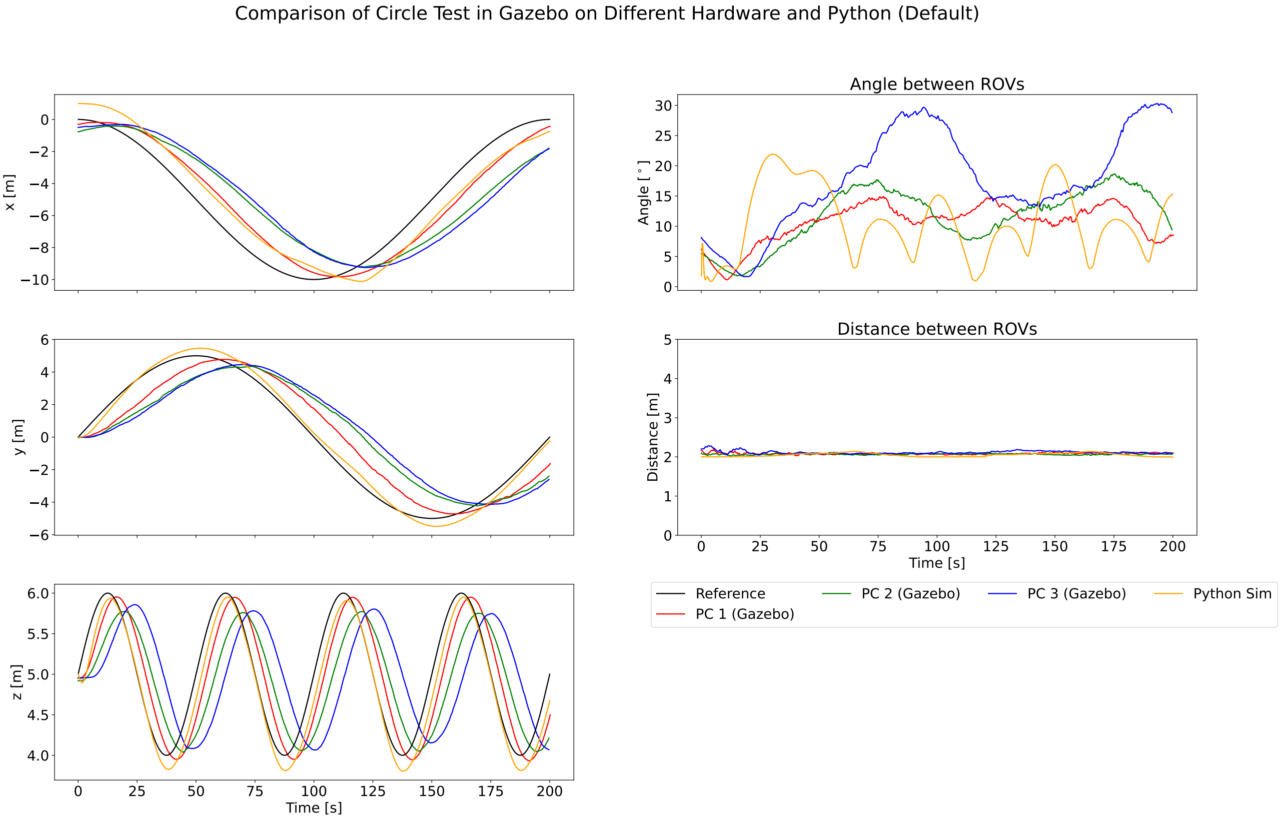


Figure I.1: Comparison between Gazebo and Python, circle test (median)

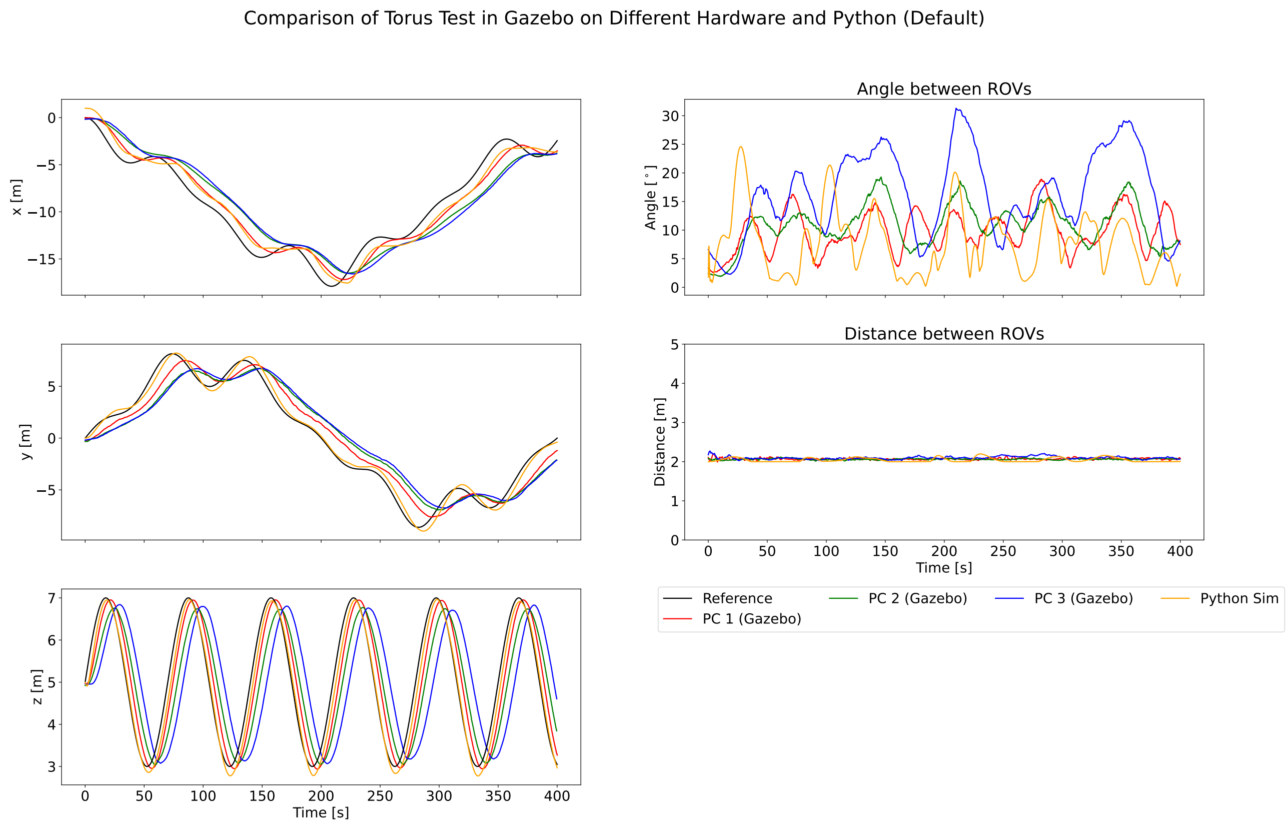


Figure I.2: Comparison between Gazebo and Python torus test (median)

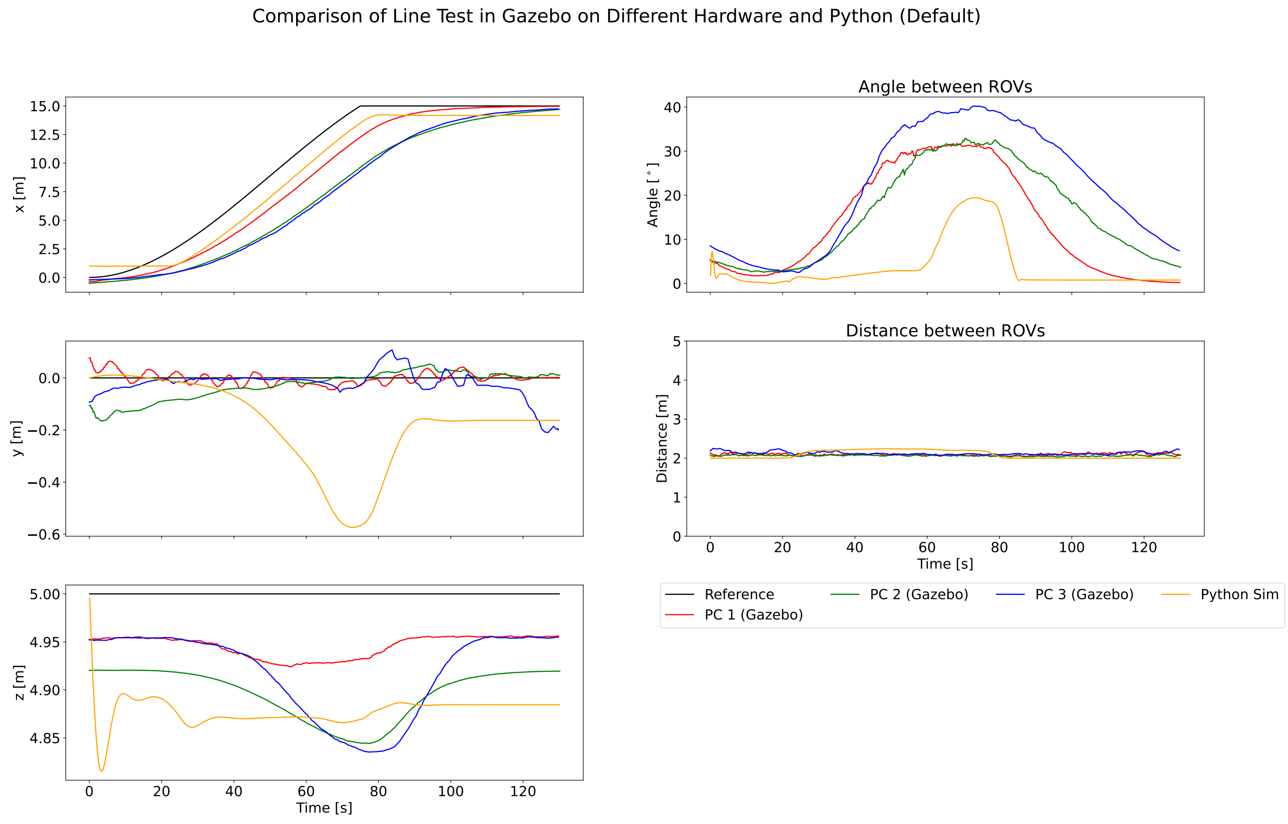


Figure I.3: Comparison between Gazebo and Python line test (median)

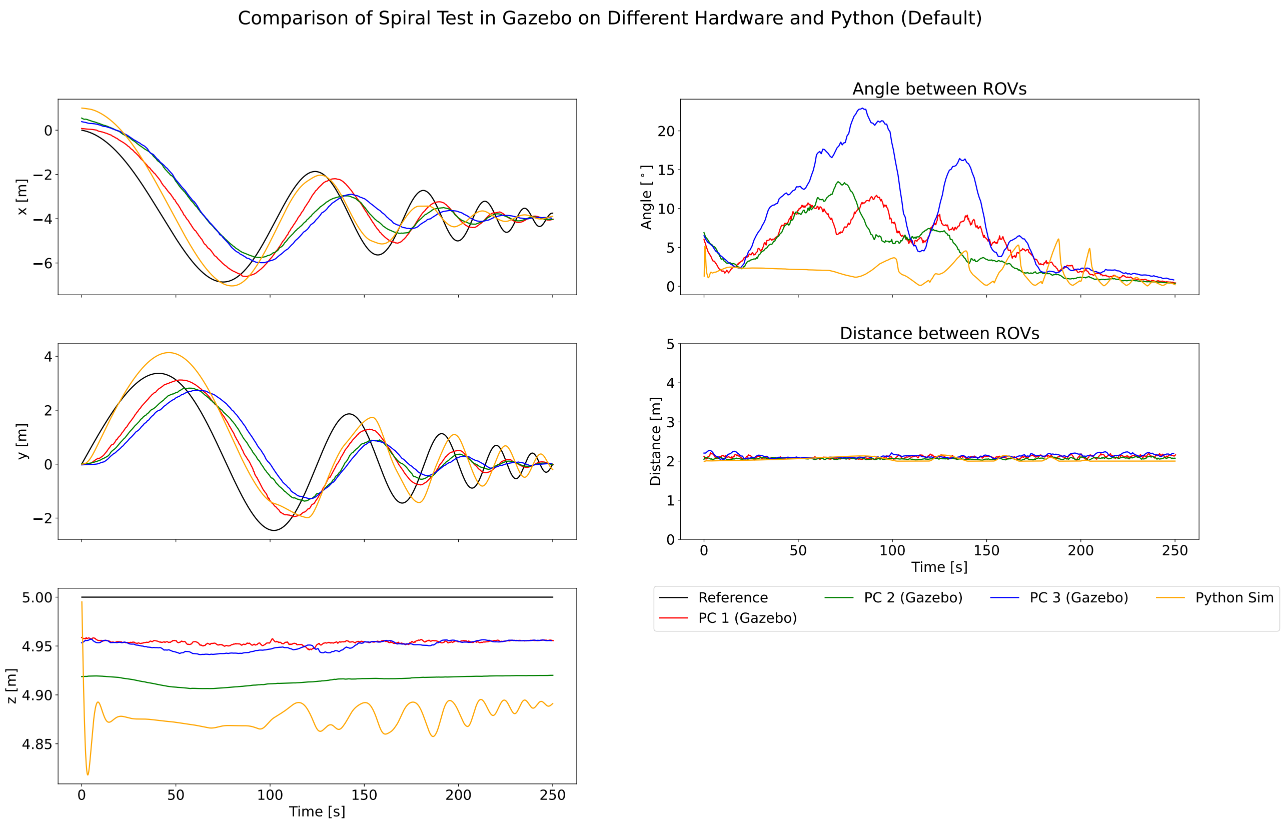


Figure I.4: Comparison between Gazebo and Python spiral test (median)

Appendix J

Poster

NTNU

Decentralized Model Predictive Control for Increased Autonomy in Fleets of ROVs

Lauritz R. Fosso, Kristian A. Johannesen, Pål K. Kjærem, Tor-Harald Staurnes
Department of Engineering Cybernetics - NTNU, Trondheim Norway

NTNU

Summary

This project dealt with the design and implementation of a decentralized model predictive control architecture to be implemented in a set of BlueROV2 Heavys.

The ROVs communicate through optical sensors, and therefore the line of sight between the ROVs is a prerequisite for communication. In that context, a model predictive controller is designed that achieves this by having access to the other ROV's position. The system is implemented in ROS 2.

The simulations were performed in Gazebo, and a set of four tests were designed to evaluate the controllers' robustness, these tests were run with different disturbances such as ocean currents, with packet loss, and modifications of system parameters.

Mathematical Model

The mathematical model utilises Fossen's robot-inspired equations to model the ROVs' dynamics [1, p. 15]. The accuracy of the mathematical model is critical to the performance of the controller since it relies on this model to predict future dynamics.

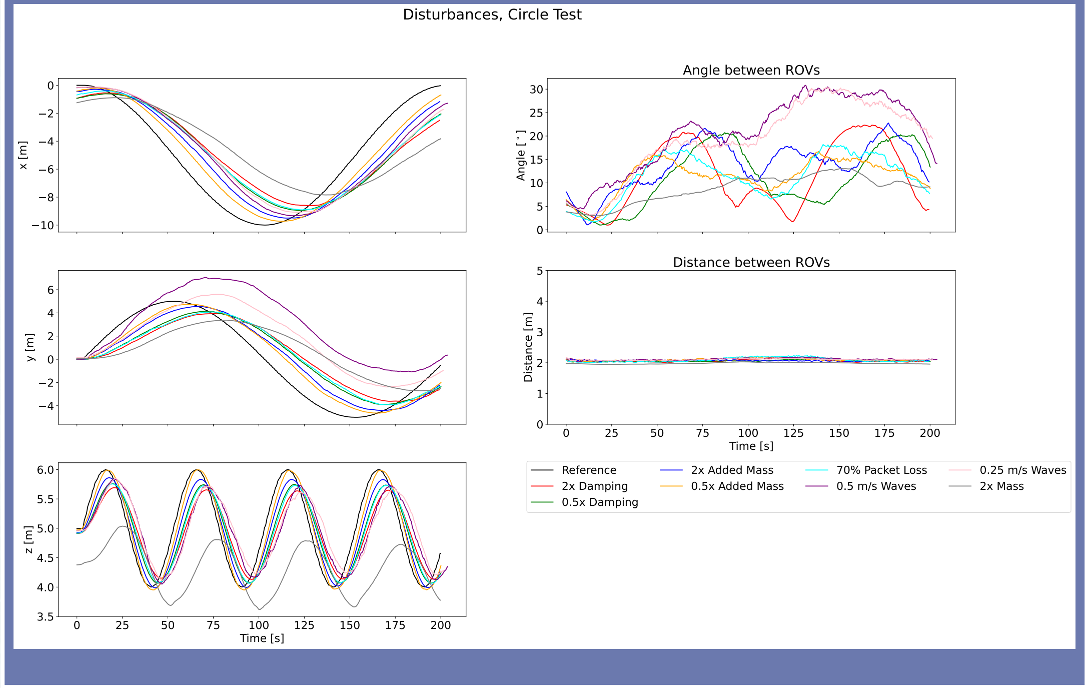
MPC

The model predictive controller, implemented in the Python toolbox do-mpc[2], is designed to perform crucial tasks for multi-agent systems. Namely, line of sight for communication, collision avoidance and path following. These tasks were the basis for the design of the cost function and constraints set for the controller. A solution for predictable positioning referred to as circular setpoint is also proposed.

ROS 2

The system is implemented in ROS2, which is an open-source software development kit used in robotics, with the goal of enabling increased modularity. ROS 2 is founded on the concept of *nodes* that perform a specific task and communicate with each other through standardised protocols.

Results


Disturbances, Circle Test

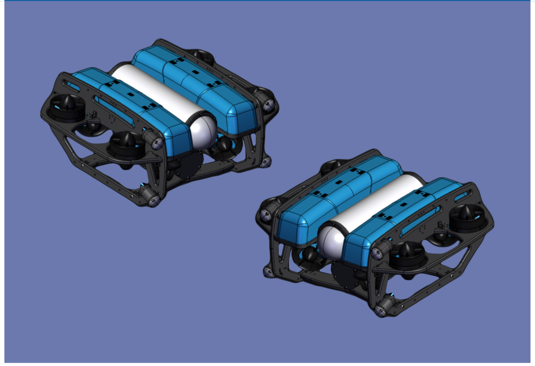
Simulation

To test the control system, two simulators were used. Firstly, a Python simulator was used which used the simulating functionality in do-mpc, which allowed for testing with an ideal model. A script was created to visualise the data through Matplotlib animations that allowed for easy interpretation of the data.

Secondly, a Gazebo simulator was used to test the system in a real-time environment, and with a physics engine that simulated the behaviour of submerged movement.

The results from the tests were used to analyse the robustness of the controller. Figure *Disturbances, Circle Test* displays the median values of the tests done with different disturbances.

BlueROV2 Heavy



Video from Simulations

This QR code links to a video showing some of the simulations in Gazebo and Python.

References

[1] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*, 2nd ed. Chichester, UK: John Wiley & Sons Inc, 2021.

[2] S. Lucia and F. Fiedler, *Model predictive control python toolbox*, 2023. [Online]. Available: https://www.do-mpc.com/en/latest, (Accessed: 03/04/2023).

NTNU

Norwegian University of
Science and Technology