Bruno Scaglione; Pedro Marzagão

Integrating a high fidelity ship maneuvering simulator with ROS: a path-following case study

USP - São Paulo

June 2021

Bruno Scaglione; Pedro Marzagão

Integrating a high fidelity ship maneuvering simulator with ROS: a path-following case study

Monograph submitted to the Polytechnic School of the University of São Paulo, for the obtention of the Bachelor's degree in Mechatronics Engineering.

Concentration Area: Robotics and Control Systems

Advisor: Prof. Dr. Eduardo Aon Tannuri

USP – São Paulo
June 2021

Bruno Scaglione; Pedro Marzagão

Integrating a high fidelity ship maneuvering simulator with ROS: a path-following case study

Monograph submitted to the Polytechnic School of the University of São Paulo, for the obtention of the Bachelor's degree in Mechatronics Engineering.

Concentration Area: Robotics and Control Systems

Advisor: Prof. Dr. Eduardo Aon Tannuri

USP – São Paulo
June 2021

ACKNOWLEDGEMENTS

To professor Eduardo Aon Tannuri, for giving us directions, help and advice throughout this year. To Humberto Makiyama and Benedito Moraes from TPN, who helped us with pydyna. We thank all of you.

To my parents, Claudio César de Carvalho Scaglione and Rosely Costa de Carvalho Scaglione, who worked so hard to give me this opportunity and support me no matter what. To my brother, Luiz Fernando de Carvalho Scaglione, who has always been there for me at the hard times. To my grandmothers, Gessi Costa and Iracema Dolácio, who always demonstrated unconditional support and love for me. To my friends, who helped me relieve the stress and made me enjoy some good times. I sincerely thank all of you.

Bruno Scaglione

First and foremost, I would like to thank my whole family for their guidance and support throughout my whole life, especially my parents, Jonas Marzagão and Flávia Martelli Marzagão. Also, I would like to thank my friends, that despite any distance, were always there for me when I needed them the most. And lastly, I want to thank all the people that in any way helped me become better than I was the day before.

Pedro Marzagão

ABSTRACT

BRUNO, S.; PEDRO, M. **Integrating a high fidelity ship maneuvering simulator with ROS: a path-following case study**. 2021. 230 p. Monografia (Bacharelado em Engenharia – Engenharia Mecatrônica) – Escola Politécnica, Universidade de São Paulo, São Paulo – SP, 2021.

Ship maneuvering is a difficult task, especially in restricted areas such as port access canals. Non-robust design of canals can lead to huge accidents in the future. The stochastic nature of humans and the environment make the prediction of ship trajectories a probabilistic problem. The design of these access passages is done by analyzing the empirical distribution of ship trajectories. Sections with large variability of trajectories demand greater width and areas with lower variability can have smaller width to save money. The samples are taken both from computer run simulations, known as *fast-time*, and human pilot run simulations, known as *real-time*. This monograph tackles the problems with the current situation of the *fast-time* simulations. The current software that controls the craft is relatively legacy and does not take advantage of the vast capabilities of the state-of-the-art ship maneuvering simulator, named *pydyna*, and developed inside the Numerical Offshore Tank (TPN) of the University of São Paulo (USP). The *Robot Operating System 2* (*ROS2*) is presented as a solution to host these applications. The simulator is integrated within *ROS2* in the form of a *ROS2* package, and a proof-of-concept study case is developed to approximate real-world *fast-time* simulations. The study case is about a ship that follows a desired path defined by *waypoints* containing desired 2D localization coordinates and desired surge velocity. It is implemented as a Guidance-Navigation-Control (GNC) architecture, with a visualization module. The results show that the craft is able to follow smooth paths, at the environmental condition in which it was tuned and with a favourable initial yaw angle. The velocity control works for collinear *waypoints*. The maneuvering simulator and path-following packages are available to be used by TPN to kick-off their applications using *ROS2*, especially regarding *fast-time* simulations. Therefore, port canal design, and other applications, can be optimized by using the advantages and tools provided by the robotics distributed framework.

Keywords: Ship, Simulator, ROS2, GNC, Control.

RESUMO

BRUNO, S.; PEDRO, M. **Integrando um simulador de manobras de navios, de alta fidelidade, com ROS: estudo de caso de um seguidor de caminho**. 2021. 230 p. Monografia (Bacharelado em Engenharia – Engenharia Mecatrônica) – Escola Politécnica, Universidade de São Paulo, São Paulo – SP, 2021.

Manobrar navios é uma tarefa difícil, especialmente em áreas restritas como canais de acesso à portos. Projetos não-robustos de canais podem acarretar em grandes acidentes no futuro. A natureza estocástica de humanos e do ambiente fazem da predição de trajetória do navio um problema probabilístico. O design dessas passagens de acesso é feito analizando a distribuição empírica das trajetória do navio. Áreas com grande variabilidade de trajetórias demandam maior espessura e áreas com menor variabilidade podem ter menor espessura para economizar dinheiro. As amostras são tiradas tanto de simulações computadorizadas, conhecidas como *fast-time*, quanto de simulações feitas com pilotos humanos, conhecidas como *real-time*. Esta monografia ataca os problemas envolvendo a situação atual das simulações *fast-time*. O *software* atual que controla o navio é relativamente legado e não tira vantagem de várias capacidades do simulador de manobras estado-da-arte, chamado *pydyna*, e desenvolvido dentro do Tanque de Provas Numérico (TPN) da Universiade de São Paulo (USP). O *Robot Operating System 2 (ROS2)* é apresentado como uma solução para hospedar estas aplicações. O simulador é integrado dentro do *ROS2* como um pacote *ROS2*. Adicionalmente, uma prova de conceito, em forma de estudo de caso, é realizada com objetivo de aproximar uma aplicação real de simulações *fast-time*. O estudo de caso trata de um navio que precisa seguir um caminho especificado por *waypoints* com coordenadas 2D de localização e velocidade longitudinal desejadas. O estudo de caso é implementado como uma arquitetura Guidance-Navigation-Control (GNC), com um módulo de vizualização. Os resultados mostram que o navio é capaz de seguir caminhos suaves, na condição ambiental em que foi tunado e para ângulos de guinada favoráveis. O controle de velocidade funciona para casos com *waypoints* colineares. Os pacotes do simulador de manobras e do seguidor de caminho estão disponíveis para serem usados pelo TPN como ponto de partida para aplicações usando *ROS2*, especialmente para simulações *fast-time*. Portanto, o projeto de canais de acesso à portos, assim como outras aplicações, podem ser otimizados ao aproveitar as vantagens e ferramentas oferecidas pelo *framework* de robótica distribuído.

Palavras-chave: Navio, Simulador, ROS2, GNC, Controle.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ROS2	Robot Operating System 2
AI	Artificial Intelligence
ASVs	Autonomous Surface Vehicles
CG	Center of Gravity
COLREGS	International Regulations for Avoiding Collisions at Sea
DOF	Degrees of freedom
DP	Dynamic Positioning
GNSS	global Navigation Satellite System
GPS	Global Positioning System
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
LF	Low Frequency
LIDAR	Light Detection And Ranging
LOS	Line of Sight
MIMO	Multiple Input Multiple Output
MISO	Multiple Input Single Output
MOS	Meta-Operating System
OS	Operating System
PID	Proportial Integral Derivative
SIMO	Single Input Multiple Output
SISO	Single Output Single Input
SV	Surface Vehicle
TPN	Numerical Offshore Tank
USP	University of São Paulo
USVs	Unmanned Surface Vehicles
UUV	Unmanned Underwater Vehicle
WF	Wave Frequency

CONTENTS

CHAPTER

1

INTRODUCTION

1.1 Motivation

When it comes to ship maneuvering, it takes tremendous skill from the pilot to operate in difficult areas such as: restricted areas, areas with significant environmental disturbance or many obstacles. Not only does the operator have to be aware of the surroundings, he also needs to take into account wind, waves and current that influence the ship's movement. In a restricted environment, such as a canal, large deviation from the desired trajectory can lead to huge accidents.

Recently, in March 2021, the whole world witnessed a single ship stranded in the Suez Canal blocking 12 percent of the world's trade and costing nearly 10 billion dollars a day (STEIGRAD, 2021). An animation of the accident can be seen in Figure 1.

According to the European Maritime Safety Agency, in 2016 there were 3145 casualties and incidents. There were also 26 ship lost due to maritime accidents, regarding 3500 ships. By far, the biggest cause of the reported accidents were human erroneous actions, making up 60.5 percent of them. Not only that, the majority of ships involved in these situations are general cargo ships, making up 43 percent of the accidents. But what might be the most important data among all these, is the fact that 42 percent of the casualties occurred in port areas, and 28 percent in coastal areas (EUROPEAN MARITIME SAFETY AGENCY, 2017).

One of the solutions to avoid these problems would be autonomous surface vehicles (ASV). They would be able to fully grasp the environmental conditions *online* and act accordingly to the situation in a precise manner. The major issue with this solution is that this technology is still under development. The most recent tests involve smaller vessels, an AI captain, still needs an on-site crew and operates without any cargo or passengers. On example is the 50-foot-long Mayflower Autonomous Ship, seen in Figure 2.

Figure 1 – Animation of Ever Given stuck in Suez Canal

Source: Kofi (2021).

Considering the use of autonomous ships is a far cry from the current paradigm, solutions taking into account humans are used. Solving all human errors is not realistic, however, these errors can be minimized through training with realistic simulators and learning from successful autonomous algorithms decisions. In addition, sometimes the line between a situation that is considered an error by the pilot and accounting for human navigation variability (and error-prone nature), can be difficult to draw. With this is mind, much of the safety engineering can drift towards the design of canals.

With the advancement of simulation techniques, design of ports and canals by hand has become outdated. Not only the calculations take time, but different ship dynamics, environmental influences and pilot navigation variability[1] must be taken into account. The simulations are able to recreate the studied area, with different models of ships. Regions in the canal can be separated through the variability of trajectories they present after a set of simulation runs. High variability regions demand a greater width. It is essential to guarantee a secure gap between the border of the region and the canal coast.

Another pertinent point that must be taken into consideration is how costly building a canal is. For example, in nowadays currency, the Panama Canal would have cost approximately 14.3 billion dollars and its length is of around 82 kilometers, with a depth of 12m and width of 33.5 meters (WHAT IT COSTS, 2016). Just a meter less or more in construction would have had an absolutely great impact on the total cost. This means that saving money is a huge concern

[1] Although there is training and there are navigation norms to follow, innate human variability can produce different trajectories for the same conditions.

also, and needs to be handled together with safety efforts.

In order to spend only as much as required, the engineers in charge of the construction should always be aware that, due to the production of new and bigger ships, the canal should be able to withstand future modifications[2].

Therefore, by using simulators it is possible to analyze, under a myriad of conditions, ship trajectories. Based on the results found, the ones building or adapting the port canal are then able determine the required shape of the course that guarantees that specific types of ships can securely pass trough, while only spending as much as needed in the process.

Figure 2 – MayFlower autonomous ship

Source: Niller (2020).

1.2 Scope

Optimal port design is one of the main areas of work, having significant expertise, inside Numerical Offshore Tank (TPN)[3], known as “Tanque de Provas Numérico”, Laboratory of the University of São Paulo (USP) (NUMERICAL OFFSHORE TANK, 2020a). TPN is a “[...] computational and experimental hydrodynamic laboratory for the design and analysis of offshore systems, ports, platforms, ships and barges” (NUMERICAL OFFSHORE TANK, 2021).

Considering the costly, unsafe and large-scale unfeasibility nature of conducting experiments with real ships; ship maneuvering simulators are developed to model real world dynamics.

[2] Information obtained with Prof. Dr. Eduardo Aoun Tannuri.
[3] Details can be found in <https://tpn.usp.br/>.

These simulators can be used along with a statistical approach for the design of safe and efficient access canals to ports.

TPN has a state-of-the-art ship maneuvering simulator developed on-premise, by the Ship Maneuvering Simulation Center, and using proprietary knowledge and technology. There are two types of maneuvering simulations conducted: *real-time* and *fast-time*. *Real-time* simulations are the ones that aim to mimic real-world operation. The pilot controls the simulator, as if it were a real ship, to perform the trajectory (e.g. navigating through a port access canal). The *Full Mission Simulator 1*, one of TPN's real-time simulators, can be seen in Figure 3. Differently, *fast-time* simulations are done by computer controlled maneuvers, named path-following autopilot, that follows desired *waypoints*. These *waypoints* contain desired locations and may present desired surge velocities as well. An example of a *fast-time* simulation, of a ship entering a port, can be seen in Figure 4.

Figure 3 – *Real-time* simulator: *Full Mission Simulator 1*

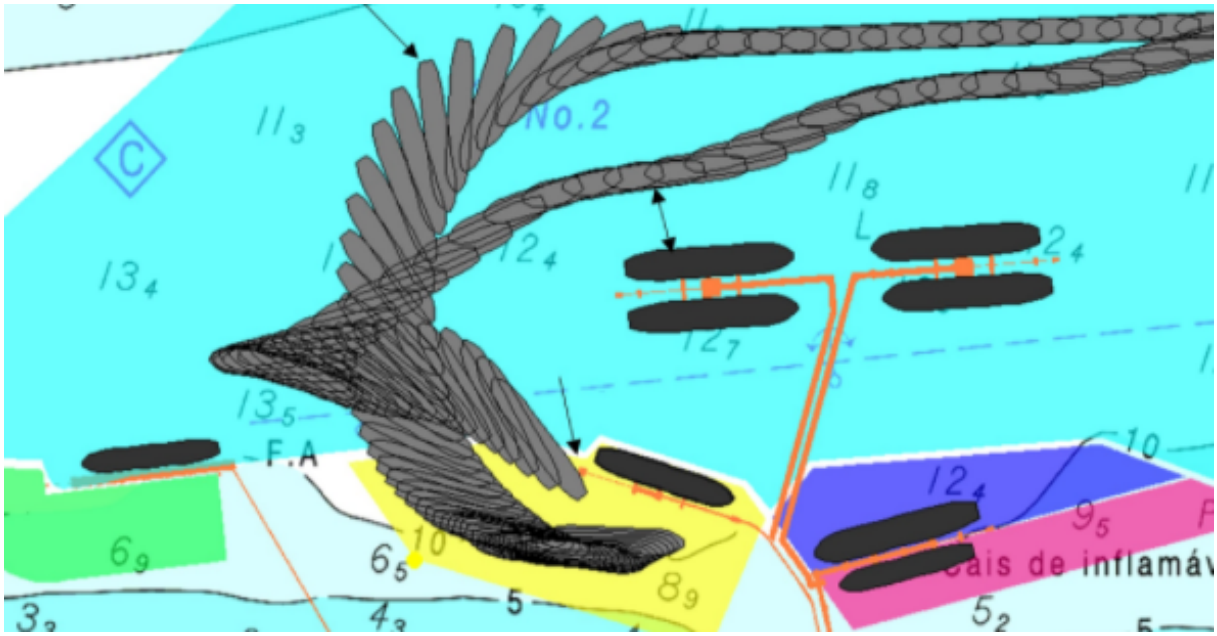
Source: Numerical Offshore Tank (2020c).

Although *real-time* simulations are more realistic in terms of real-world operations, *fast-time* simulations have a number of advantages over the former and are described below.

- **Number of runs**. Does not have the real-world time constraint imposed by mimicking real-world conditions, therefore, it can be executed much faster and consequently conduct much more runs in the same time span as *real-time* simulations (NUMERICAL OFFSHORE TANK, 2020d).

- **Less amount of resources**. Does not require scheduling of experts coming to the simulation center neither costly infrastructure setup.

Figure 4 – Example of a *fast-time* simulation



Source: Numerical Offshore Tank (2020d).

- **Standardization and reproducibility.**[4] Humans are systems subjected to a large amount of external factors which interact in a complex way, still far from current modelling capabilities. Therefore, treating them in a systematic way and assuring reproducible results is very difficult. Computer programs provide these qualities to a great extent.

Therefore *fast-time* can contribute a lot to "[...] optimize existing ports, increase ship size or draft and to assess the safety of new operations" (NUMERICAL OFFSHORE TANK, 2020a).

Currently TPN makes use of both types of simulations for analysis and design of port access canals. With a small amount of *real-time* (approximately 30) runs and a vast amount *fast-time* (aproximately 500) runs (verbal information[5]). This approach ensures robustness of the design to auto-pilot and pilot conditions, and also enables the comparison of auto-pilot *vs* pilot trajectories to identify potential improvements in *fast-time* software or pilot skills.

The *fast-time* software in TPN is composed of two parts: the core state-of-the-art maneuvering simulator implemented in *C++* with a *Python* Interface, distributed as a private *Python* library and named *pydyna*; and the surrounding path-following architecture implemented in *MatLab*.

However, this structure does not take advantage of the vast capabilities provided by the state-of-the-art maneuvering simulator. The control software presents some limitations, such as the ones described below.

[4] In a statistical sense, considering a sufficiently large sample size.
[5] Information obtained with Prof. Dr. Eduardo Aoun Tannuri.

- **Lack of native integration with the maneuvering simulator.** There is not a low-level layer that supports the communication between the systems. Consequently, development becomes cumbersome, unscalable and with low communication performance. Further, the *fast-time* software does not make use of additional resources provided by the simulator such as Dynamic Positioning (DP), hardware in the loop, mooring and cable control infrastructure capabilities; illustrated in Figure 5.

- **Lack of support for modularity.** The current software is, to a great extent, a monolith, and thus cannot have independent development, interchangeable parts and fault tolerance.

- **Does not enable easy aggregation of state-of-the-art robotics packages.** For example, to implement a new control algorithm, sensor model or visualization modules. These tasks would require the developers to learn how the module was coded in a third-party project and re-implement from scratch the modules and their communication, and at the same time, trying to make the newest version still compatible with the older ones.

- **Does not enable rapid prototyping and experimenting.** Since everything is done from scratch, the development cycle is very big, making it very costly to try new approaches and test hypotheses. Consequently experimenting alternatives *off-the-shelf* is not a possibility and quick evaluation of new modules cannot be done. Finally, since there is not a framework controlling the architecture (it is *hard-coded*), rapid experimenting of different architectures and configurations is unmanageable.

- **Requires experts to develop and maintain.** It is not a trivial task to deal with advanced control algorithms or complex modelling. Furthermore, dealing with the increasing complexity in a *hard-coded* communication system, as more components are added to the architecture, is not easy neither practical[6].

- **Does not have hardware in the loop.** The current implementation does not take into account the dynamics of the hardware components in real world non-ideal conditions.

- **Cannot scale easily to a complex robotic system.** Improvements in *fast-time* simulations requires incorporating more aspects of reality into the closed-loop system and having more intelligent algorithms. These generally come in the form of independently developed modules that communicate, with one or more other components, in synchronous or asynchronous manner. The increasingly complexity of dealing with: communication between services, compatibility issues and hardware drivers; while working on the modules themselves, requires a low-level abstraction layer[7]. With an abstraction layer, low level development and integration can be abstracted, and components can be re-utilized. This

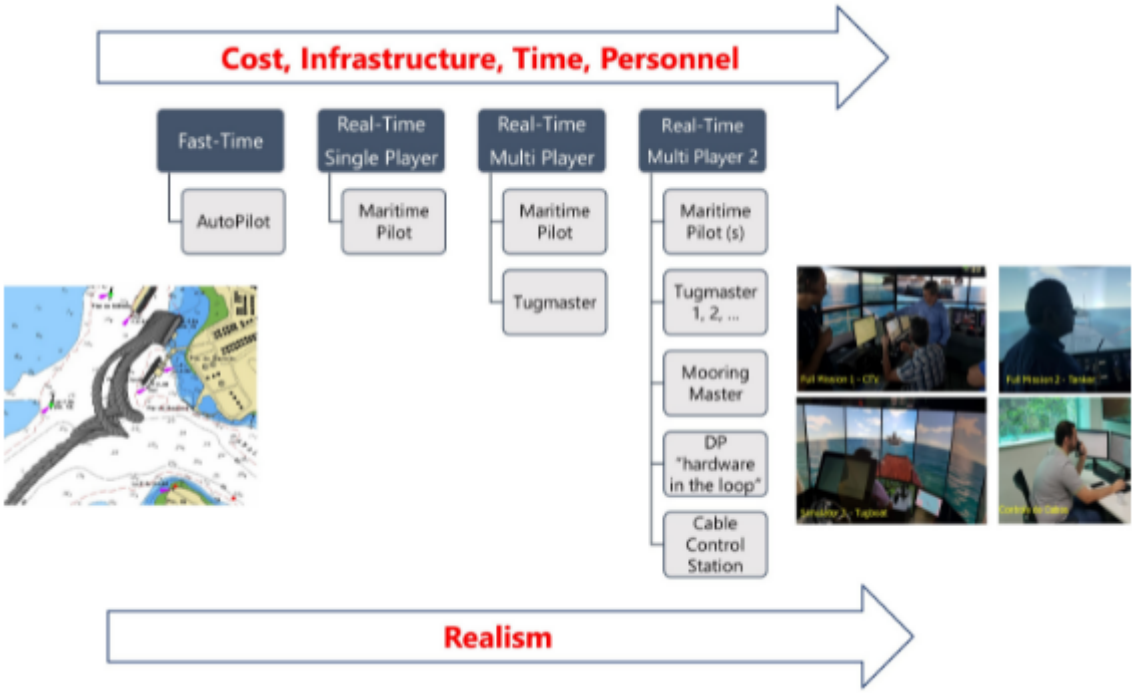
[6] Considering the evolution of the current architecture from only a path-following control algorithm and the core simulator to possibly many components.

[7] Not present in the current software.

enables research and development deep focus only on the area of interest, while still having state-of-the-art tools at disposal.

In addition, although current ship operations in access canals are conducted by pilots, more and more automation is being integrated into maritime applications, such as, the rise of Unmanned Surface Vehicles (USVs) and Autonomous Surface Vehicles (ASVs). Therefore, cargo ship operation in a near future might be subject to a shift to complex robotic systems. At the same time in which path-following control algorithms, which have limitations such as a non-systematic way of setting *waypoints* and not having future knowledge incorporated (for anticipation of dynamics), might give place to approaches such as Artificial Intelligence (AI) (verbal information[8]).

Figure 5 – TPN's simulations summary



Source: Numerical Offshore Tank (2020d).

The Robot Operating System 2 (*ROS2*) is a low-level framework, that operates on top of an Operating System (OS), and provides solutions to many of these problems; in addition to having a large community behind it. *ROS2* "[...] provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more [...]" (WIKI, 2020).

A great advantage also resides in the fact that the framework operates mainly in *Python* and *C++*. The *Python* language provides a wide range of useful open-source libraries and applications to be used right away. The *C++* library also provides a lot of libraries, but aggregates

[8] Information obtained with Prof. Dr. Eduardo Aoun Tannuri.

the advantage of code efficiency. With these two, quality software can be aggregated into the *ROS2* environment, without having any relation to *ROS2*.

This project aims develops the *pydyna-simple* package that wilencapsulate te maneuvering simulator in a ROS node and enable taking advantage of the core state-of-the-art maneuvering simulator in the *ROS2* environment. There it can be combined with multiple other packages natively and in modular fashion. In this structure, the project can also scale very well to a complex robotic system when more modules are added.

The package proposition is done in the context of a proof-of-concept case study, where the aim is to provide enough demonstration of the capabilities that come along with the *ROS2* environment. It is a path-following scenario, similar to the ones in *fast-time* simulations.

1.3 Methodology

The methodological framework adopted for the development of this monograph consists on a sequence of steps, which are directly related to the chapters. The steps, in sequence, are the following:

1. Overview of GNC systems, *pydyna*, *ROS2* and common approaches adopted in the literature for surface vehicles in *ROS2*. Furthermore, an "off-literature" search for similar implementations;
2. Study on GNC systems (with focus on the path-following scenario), *pydyna* and *ROS2*;
3. Formalization of the problem, by stating the objectives and requirements. The objectives are the fulfillment of the TPN team subjective needs. The requirements are objective constraints of the problem and proxies for the objectives.
4. Development of the *pydyna-ros* package;
5. Development of a prototype for the case study with a simple GNC architecture;
6. Improvement of the prototype, by adding more functionality;
7. Development of user documentation for the software;
8. Formalization of the problem (more specifically, the solution to the problem), by stating the specifications. The specifications should guarantee full description and reproducibility;
9. Exposure and discussion of the results obtained by the path-following simulation;
10. Exposure of the project's contributions;
11. Discussion of future work paths based on this project.

CHAPTER

2

THEORETICAL BACKGROUND

2.1 Maneuvering model

There are two types of dynamical models in this context: production and control design models. These are described below.

- **Production models**: these are models which intend to mirror as much as possible the real-world system. These are useful for simulating the system. *Pydyna's* core is a model of this kind, for craft maneuvering.
- **Control design models**: these models are simplified versions intended to design controllers. These are used do design the yaw angle and surge velocity controllers in the case study.

The maneuvering model analyzed in this section is ***pydyna's* production model**. The model is described by Equation 2.1, as presented by Amendola *et al.* (2020).

$$\begin{bmatrix} m-X_{\dot{u}} & 0 & 0 \\ 0 & m-Y_{\dot{\mathrm{v}}} & mx_G-Y_{\dot{r}} \\ 0 & mx_G-Y_{\dot{r}} & I_z-N_{\dot{r}} \end{bmatrix} \begin{bmatrix} \dot{u} \\ \dot{\mathrm{v}} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} 0 & -mr & -mx_G r + Y_{\dot{\mathrm{v}}}\mathrm{v} + Y_{\dot{r}} r \\ mr & 0 & -X_{\dot{u}} u \\ mx_G r + Y_{\dot{\mathrm{v}}}\mathrm{v} + Y_{\dot{r}} r & X_{\dot{u}} u & 0 \end{bmatrix} \times \begin{bmatrix} u \\ \mathrm{v} \\ r \end{bmatrix} + (X_{\dot{u}} - Y_{\dot{\mathrm{v}}}) \begin{bmatrix} 0 & r & 0 \\ r & 0 & 0 \\ -\mathrm{v} & u_c-u & 0 \end{bmatrix} \begin{bmatrix} u_c \\ \mathrm{v}_c \\ 0 \end{bmatrix} = F_{rudder} + F_{prop} + F_{tugs} + F_{curr} + F_{wind} + F_{wave} \tag{2.1}$$

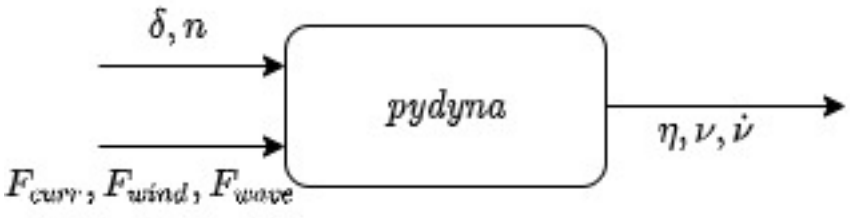
Respectively, the vectors $\nu = [u \ \mathrm{v} \ r]^T$ and $\dot{\nu} = [\dot{u} \ \dot{\mathrm{v}} \ \dot{r}]^T$ are the velocity and acceleration of the vessel. The generalized coordinates are explained on page 43. According to Amendola *et*

al. (2020), "This model assumes an ocean current velocity of slow and irrotational variation that can be decomposed in the LRF as $\nu_c = [u_c \;\; v_c \;\; 0]^T$"

As for the right side of the equation: F_{rudder}, F_{prop}, F_{tugs}, F_{curr}, F_{wind}, F_{wave} are, respectively, the forces of the rudder, propeller, tug boats, current, wind and waves. The force F_{tugs} is not considered in this project, since only single ship simulations are performed. The variable m is the mass of the ship. The other variables are coefficients; it is not interesting to expand on them for the present moment.

A block diagram representing a simplified version of the craft's model can be seen below:

Figure 6 – Block diagram of *pydyna's* craft model

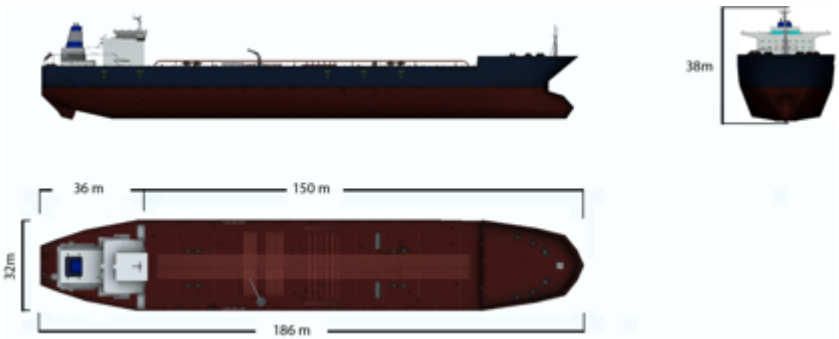


Source: Elaborated by the author.

In Figure 6, the inputs of F_{curr}, F_{wind} and F_{wave} represent the environmental disturbances due to current, wind and waves respectively; these are the most relevant disturbances. The other inputs, δ and n, are respectively rudder angle and the propeller rotation. The outputs, η, ν and $\dot{\nu}$, are, in order, the position, velocity and acceleration 3-Degrees of freedom (DOF) vectors of the vessel at a given time in the simulation.

The specific craft used in simulation is the *Tanker55000DWT*, which is illustrated[1] in Figure 7.

Figure 7 – Different views and parametrization of the *Tanker55000DWT*



Source: Numerical Offshore Tank (2020b).

[1] The illustration was provided by TPN-USP but is not currently public.

2.2 GNC systems

This section presents a brief overview of GNC systems, following a more detailed explanation of Guidance, Navigation and Control in the context of ASVs. The choices and simplifications adopted for the case study are highlighted along the way.

The GNC system is composed of three subsystems which interact with each other to successfully control the craft in a desired fashion. In order to do this, first it is necessary to determine what is the desired behaviour, this is handled by the Guidance subsystem. Secondly, inputs to the system need to be provided in a way that controls it to the desired behaviour, this is handled by the Control subsystem. These systems typically make their decisions based on information of the current state of the craft and on the environment, this is provided by the Navigation subsystem. The division in three independent modules is a common and efficient approach, since it provides modularity in a relatively loosely coupled manner (FOSSEN, 2011). Modular development enables more practical, scalable and fault-tolerant development. However, the boundaries of these systems are not always that clear with some approaches, especially between Guidance and Control.

In practical terms, these systems "[...] generally consist of commercial off-the-shelf (COTS) sensors coupled with on-board computers which carry out the path planning and generate signals to directly control the actuators" (CAMPBELL; NAEEM; IRWIN, 2012).

A formal description of each of the subsystems is presented by Liu *et al.* (2016):

- **Guidance:** "[...] responsible for continuously generating and updating smooth, feasible, and optimal trajectory commands to the control system according to the information provided by the navigation system, assigned missions, vehicle capability, and environmental conditions".
- **Navigation:** "[...] concentrates on identifying the USV's current and future states (such as position, orientation, velocity, and acceleration), and its surrounding environment based on the past and current states of the USV as well as environmental information including the ocean currents and wind velocity) obtained from its onboard sensors".
- **Control:** "[...] focuses on determining the proper control forces and moments to be generated in conjunction with instruction provided by the guidance and navigation systems, while at the same time satisfying desired control objectives".

A simplified diagram representing the GNC architecture for a **closed-loop guidance and control system (which is the most common approach and also the one used in the case study**) can be seen in Figure 8.

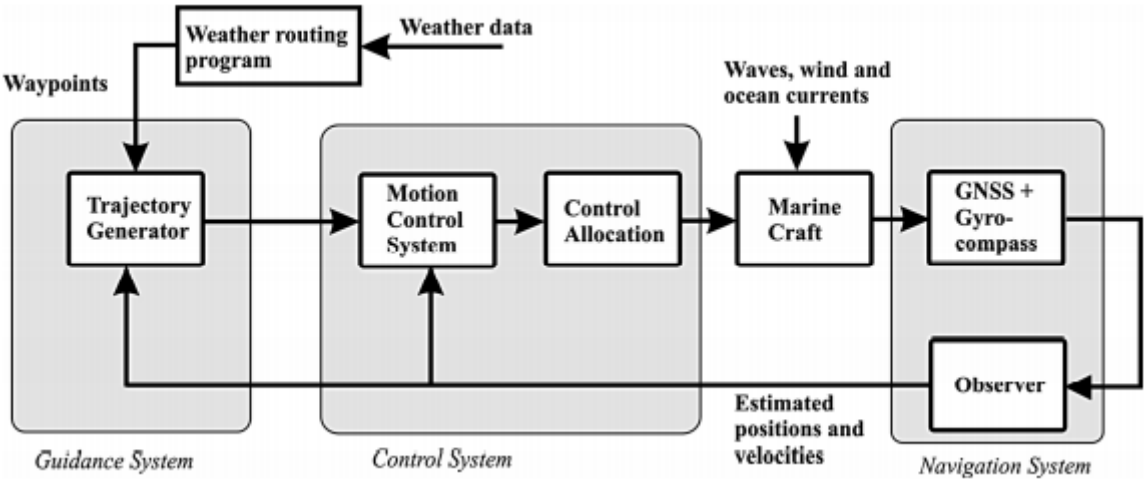
It is important to highlight that in this project the whole system is digital. The input and output of the plant is discrete, at approximately 10 Hz. Also, this frequency is less than

the sampling frequency of the sensors (GPS and IMU). Therefore, digital controllers and filters become native, making blocks such as Sampling and Zero-Order Hold not present.

To approximate the behaviour of the simulated discrete system to the real system, which uses a continuous plant, the controllers of the GNC architecture cannot wait for processing to occur to give their outputs. Processing must occur asynchronously to update the current output value. The most current control action value is given directly by the controller when it receives the filtered state. This procedure delays response, but approximates what would happen in a digital GNC architecture with a continuous plant. In the last mentioned case, a Zero-Order Hold would be present to always provide input to the dynamical system. Additionally, like in real systems, large amounts of computation are penalized, since it takes a longer time to run the program and update the outputs. This makes the craft act in delay for longer periods of time.

The Guidance block could have been implemented synchronously or asynchronously with respect to the controllers. In the synchronous way the controller only acts when receives a new set-point, this causes the craft to use "old" control action values, during roughly the processing time of Guidance, for up-to-date set-points. In the asynchronous way the craft does not wait for the set-points and uses the last set-point provided by Guidance, this causes the craft to use up-to-date action values for "not so old" (because Guidance has slower updates than Control) set-points. **The second approach is chosen, because the processing time of Guidance is relatively significant, making acting based on small delays for set-points more promising than large delays in control action.**

Figure 8 – Simplified schematic diagram of the GNC architecture for a closed-loop guidance and control system



Source: Fossen (2011).

Given the necessary introduction, it is possible to dive in each of the modules. Starting with Guidance, following with Navigation and then Control.

2.2.1 Guidance

Complete Guidance can be viewed as planning for the craft in three hierarchical stages: global, local and micro; given a goal, the craft and the available control and navigation resources. From global to micro, the stages consider a smaller absolute search space, however by adding more realistic modelling, they also consider a higher number of variables and constraints. This separation aims to make each stage a more feasible problem to solve. Stages act upon the output of the previous one (if there is one), refining and/or adapting the solution (ZHOU *et al.*, 2020). The micro stage is usually denoted as motion planning in the literature and it's output are setpoints to the control subsystem.

Global planning considers the craft as a particle and can take into account macro constraints, such as: the environment map, weather forecasts and high level protocols to follow; in order to calculate the best path. This stage uses search and/or optimization methods for this objective. The output is usually a set of discrete *waypoints*, but can also be a continuous path.

Local planning[2] provides capabilities such as local optimization and obstacle avoidance; considering partial and decoupled dynamic constraints of the craft and/or collision avoidance protocols, such as, the International Regulations for Avoiding Collisions at Sea (COLREGS). The output is usually a set of discrete *waypoints*, but can also be a continuous trajectory.

Micro or Motion Planning considers all the proposed constraints and focuses on generating smooth and feasible paths or trajectories, taking into account the actuators constraints and the dynamic system stability. Motion Planning methods aim to guarantee convergence of a path or trajectory related error metric to zero, as time goes to infinity. The output are state[3] setpoints for the control system to achieve.

Depending on the problem scope and the level of human intervention, complete guidance can be reduced. Global or even local planning may not be required if the initial search space is not so big, or a human is already providing the functionality. **In this monograph's case study, only motion planning is considered** due to the access canal being a reduced search space, the environment being static and for simplicity purposes (by having the *waypoints* set by a human).

A focus can now be given to the Micro stage. Motion Planning is usually control-based or sampling-based and can be divided according to the control objectives, as follows.

- **Setpoint Regulation:** "[...] is a special case where the desired position and attitude are chosen to be constant" (FOSSEN, 2011).
- **Path Following:** "[...] is following a predefined path independent of time" (FOSSEN, 2011).

[2] Motion planning can also be called Local Planning or Local Methods in the literature, however this is not the terminology used here.

[3] Often partial state setpoint, this is directly related to the number of actuators.

- **Path Maneuvering:** "[...] Relates vehicle motion to feasible path following, often with less importance placed on time in favour of spatial constraint" (CAMPBELL; NAEEM; IRWIN, 2012).

- **Trajectory Tracking:** "[...] where the objective is to force the system output $y(t) \in \mathbb{R}^m$ to track a desired output $y_d(t) \in \mathbb{R}^m$" (FOSSEN, 2011).

It is important to highlight that the control objectives need to be closely related to the actuation system of the craft. When the number of actuators of the craft is less than the number of degrees of freedom of the motion planning workspace, the problem is not trivial neither can be solved by linear theory, due to non-holonomic constraints. "It is trivial to control a fully actuated marine craft while underactuation puts limitations on what control objectives can be satisfied. More specifically, the control objective must be formulated such that the craft can satisfy all requirements even if it is equipped with actuators that purely produce forces in some directions. Unfortunately, most marine craft are underactuated since they cannot produce control forces and moments in all DOFs." (FOSSEN, 2011).

A typical marine craft has two actuators:

- **rudder**: controls the yaw angle;

- **propeller or thruster**: controls surge velocity.

Therefore, **the path-following control objective is chosen as the case study for simplicity**, since the chosen method works on a 2-D workspace, where the coordinates are the cross-track error $e(t)$ and along-track distance $s(t)$. Hence, it becomes a fully actuated control setting. The path following guidance, however, only focuses on converging the cross-track error to zero. Converging along-track distance to zero is typically seen in trajectory tracking. Along-track distance is also used for velocity guidance laws in path maneuvering scenarios.

A popular method (and also the chosen one for the case study) for path-following motion planning, is the *Line of Sight (LOS)* guidance law. This method receives as input discrete *waypoints* and guarantees the convergence of the cross-track error $e(t)$ to zero, as time goes to infinity, along the straight line connecting the *waypoints*. The method provides a steering law, in other words, it only controls the yaw angle of the craft and assumes positive absolute velocity. The yaw angle is controlled based on the knowledge of the current position of the craft, and a point between the craft and the next waypoint; where this point needs be inside the straight line connecting the *waypoints*. The surge velocity shouldn't be too high, in order to avoid potential problems related to high turning rates such as actuator wear and craft instability. A summary of the *LOS enclosure-based steering law* is presented next. A graphical representation can be seen in Figure 10.

Firstly, it is necessary to introduce the generalized coordinates and references frames adopted. The original 6-DOF configuration space of the craft can be reduced to 3-DOF for a marine craft operating in the horizontal plane (FOSSEN, 2011). An illustration of the 3-DOF configuration space is present in Figure 9, where $O_E X_E T_E$ is the earth-fixed reference frame and $O_B X_B Y_B$ is the body-fixed reference frame, that is positioned at the Center of Gravity (CG) of the craft.

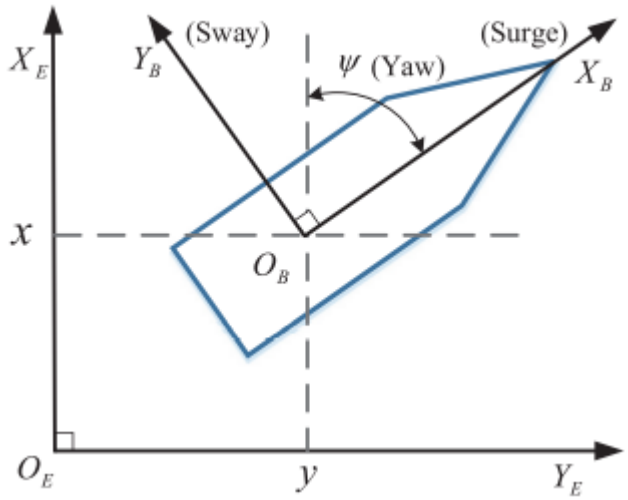
Based on Fossen (2011), the generalized positions and velocities (in order) in the 3-DOF configuration space are:

$$\eta = [x \;\; y \;\; \psi]^T \in \mathbb{R}^2 \times S \tag{2.2}$$

$$\nu = [u \;\; \mathrm{v} \;\; r]^T \in \mathbb{R}^3 \tag{2.3}$$

where $\mathbb{R}$ is the Euclidean space; S is a circumference; $[x, \; y, \; \psi]^T$ has coordinates in the earth-fixed reference frame; and $[u, \; v, \; r]^T$ has coordinates in the body-fixed reference frame.

Figure 9 – Schematic diagram of the 3-DOF configuration space of a marine craft



Source: Liu *et al.* (2016).

The following description of the *LOS enclosure-based steering law* is based on Fossen (2011) and Liu *et al.* (2016). As a note, given any real numbers a and b, $arctan2(a, \; b)$ denotes the four-quadrant version of arctan $\left(\frac{a}{b}\right) \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

$$U(t) := \sqrt{u^2 + \mathrm{v}^2} \tag{2.4}$$

where $U(t)$ is the absolute velocity of the craft and $U(t) > 0$.

$$\chi(t) := arctan2(\dot{x}(t), \dot{y}(t)) \in \mathbb{S} := [-\pi, \pi] \tag{2.5}$$

where $\chi(t)$ is the course angle (angle between X_E and the craft resultant velocity vector).

$$\beta = \arcsin\left(\frac{\mathrm{v}}{U}\right) \tag{2.6}$$

where β is the sideslip angle (angle between X_B and the craft resultant velocity vector).

Considering a straight line defined by two *waypoints* $p_k^n = [x_k, y_k]^T \in \mathbb{R}^2$ and $p_{k+1}^n = [x_{k+1}, y_{k+1}]^T \in \mathbb{R}^2$:

$$\alpha_k := arctan2(y_{k+1} - y_k, x_{k+1} - x_k) \in \mathbb{S} \tag{2.7}$$

where α_k the angle is the angle of rotation of the earth-fixed reference frame to the path-fixed reference frame, which has it's x axis aligned with the vector connecting the two *waypoints*.

The desired yaw angle is given by[4]:

$$\psi_d(t) = \chi_d(t) - \beta \tag{2.8}$$

where:

$$\chi_d(t) = arctan2(y_{los} - y(t), x_{los} - x(t)) \tag{2.9}$$

where $\chi_d(t)$ is the desired course angle and $p_{los}^n = [x_{los}, y_{los}]$ is found by solving Equation 2.10 and Equation 2.11 for x_{los} and y_{los}.

$$(x_{los} - x(t))^2 + (y_{los} - y(t))^2 = R_{los}^2 \tag{2.10}$$

where:

$$\tan(\alpha_k) = \frac{y_{los} - y_k}{x_{los} - x_k} = constant \tag{2.11}$$

for a chosen $R_{los} > 0$.

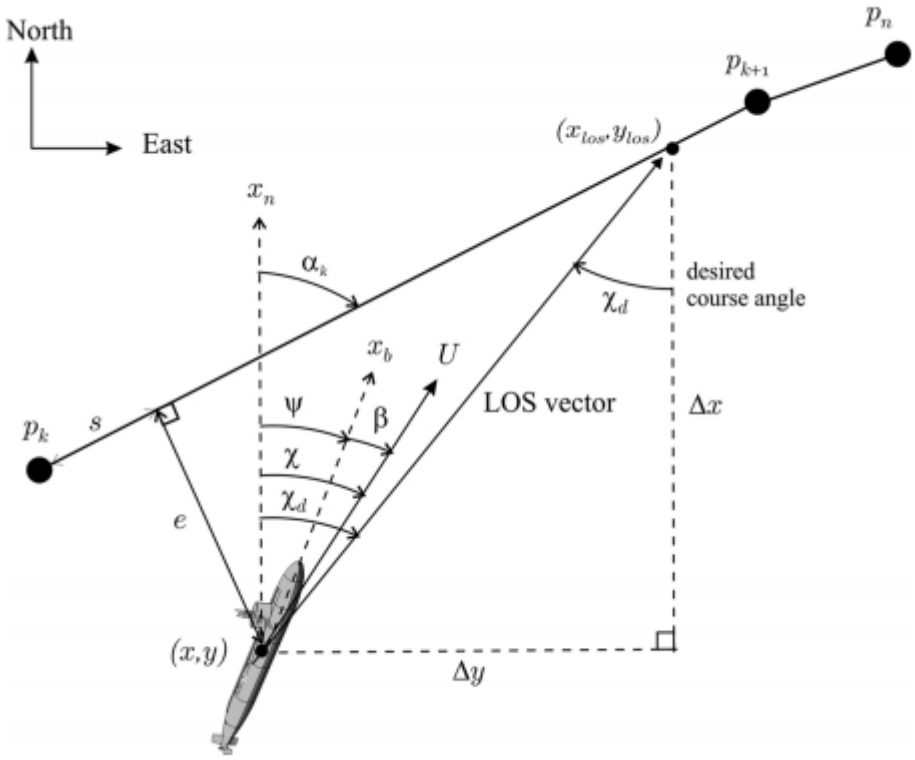
[4] Observation: the convention used here for β is assuming the sway velocity of the craft is positive to starboard (10), differently from 9, where it is positive to port.

The circle around the craft R_{los} is typically given by:

$$R_{los} = 2L_{pp} \tag{2.12}$$

where L_{pp} is the vessel length, more specifically, the length between perpendicular, illustrated in Figure 24.

Figure 10 – *LOS* guidance



Source: Fossen (2011).

2.2.2 Navigation

The Navigation subsystem is usually composed of physical sensors along with software-based data processing methods. It's job is to make available to the guidance and control systems the most useful state-related information, so that these systems can do their jobs.

The sensors should always provide information about the craft's current state. This is usually done by a global Navigation Satellite System (GNSS) such as the Global Positioning System (GPS) along with motion sensors such as the Inertial Measurement Unit (IMU). The GPS measures position, velocity and heading; while IMU measures angular velocity and acceleration. Additionally, the sensors can also gather information about the environment, with objectives such as modelling wind and waves; or avoiding obstacles. This can be done, for example, passively by the use of cameras or actively by the use of Light Detection And Ranging (LIDAR). **Following the requirements of this monograph, sensor models need to be present in the simulation to capture hardware-in-the-loop dynamics**.

However the data that comes out of these sensors are usually very noisy "[...] due to influences from (1) environmental noises; (2) accumulative errors resulting from inherent drift; (3) time-varying model uncertainties; and (4) sensor faults. Additional correction actions are hence required to improve navigation performance" (LIU *et al.*, 2016). Furthermore, the high frequency disturbances, related to the wave elevation[5], can cause unnecessary use of the actuators[6] (FOSSEN, 2011) and potentially severe damage to the aforementioned, if fed directly to the other systems (CAMPBELL; NAEEM; IRWIN, 2012).

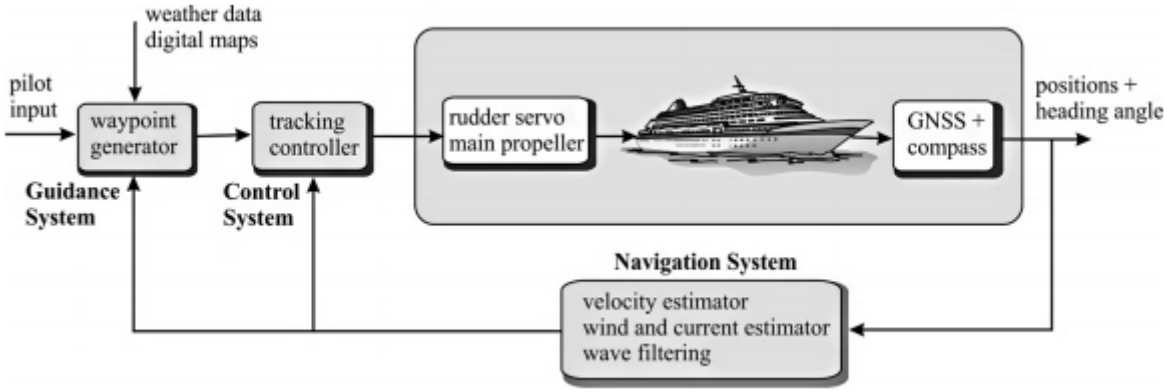
Considering the problems mentioned, some data processing modules are usually used in conjunction with the sensors. A wild point removal can remove outliers; a wave filter can remove high frequency components and a state estimator, such as a Kalman Filter, can estimate the state of the craft and/or environment based on noisy and possibly incomplete data from the sensors. A block diagram exposing some of these navigation capabilities can be seen in Figure 11, which is a heading autopilot system.

A special attention is given to frequency filters for their simplicity and effectiveness. They provide the capabilities of removing known unwanted frequencies and high frequency noise. Since waves are a major source of disturbances which operate at a given frequency and Gaussian noise is introduced by the sensors, these capabilities are fundamental to GNC system.

These methods are not explicitly in the project's requirements, however, are necessary due to the high-fidelity aspect of the simulation in conjunction with hardware-in-the-loop dynamics added by sensor error models.

[5] Which is described as a stochastic process.
[6] Reducing tracking performance and increasing fuel consumption (FOSSEN, 2011).

Figure 11 – Block diagram of a heading autopilot system



Source: Fossen (2011).

2.2.2.1 *Wave filter*

Fossen (2011) describes wave disturbances as two components. These components can be visualized as their effect on the craft control in Figure 12 and are described next.

- **First order wave forces:** produce high frequency oscillations around a mean wave force, usually denoted as Wave Frequency (WF). To remove this component it is necessary to perform *Wave Filtering*.
- **Second order wave forces:** produce the mean wave force (drift) which has Low Frequency (LF). These can be compensated by using integral action in the control law.

A common approach is to do *Notch Filtering* around the main frequency of the waves. The filtered state X_f is expressed, in the z domain, by:

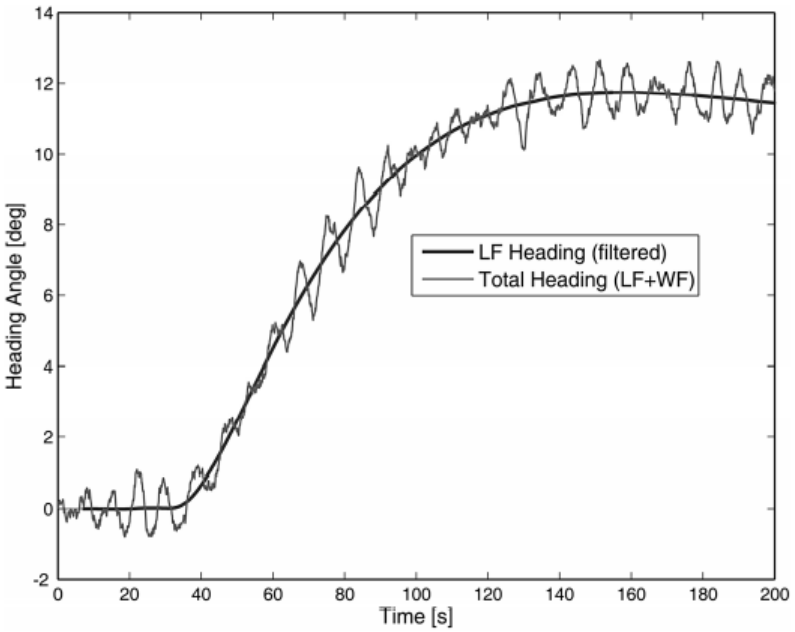
$$X_f(z) = h_n(z)X(z) \tag{2.13}$$

where h_n is a 6^{th} notch filter[7] which attenuates frequencies around the wave frequency $w_f = 0.083\ Hz$; for sampling period $T_s = 0.1\ s$[8]:

The filter is presented by Fossen (2011) in the s domain (continuous signals). Since it is a digital system, the continuos filter must be discretized. The equivalent discrete transfer function in the z domain is obtained trought the *bi-linear transform*. The steps are described below.

[7] A filter which attenuates frequencies within a specific range.
[8] Which is the time step of the simulator.

Figure 12 – Motion of a marine craft due to only LF (filtered) and WF+LF (not filtered)



Source: Fossen (2011).

First, the continuous-time transfer function of the filter is defined as:

$$h_n(s) = \prod_{i=1}^{3} \frac{s^2 + 2\zeta s + w_i^2}{(s + w_i)^2} \tag{2.14}$$

where: $\zeta = 0.7$; $w_1 = 0.29124\ rad/s$, $w_2 = 0.52124\ rad/s$ and $w_3 = 0.89124\ rad/s$ are values adapted from Fossen (2011) to be centered at the project's wave frequency.

The *bi-linear transform* is defined as:

$$s = C \cdot \frac{1 - z^{-1}}{1 + z^{-1}} \tag{2.15}$$

with $C = \frac{2}{T_s} > 0$ as a typical value.

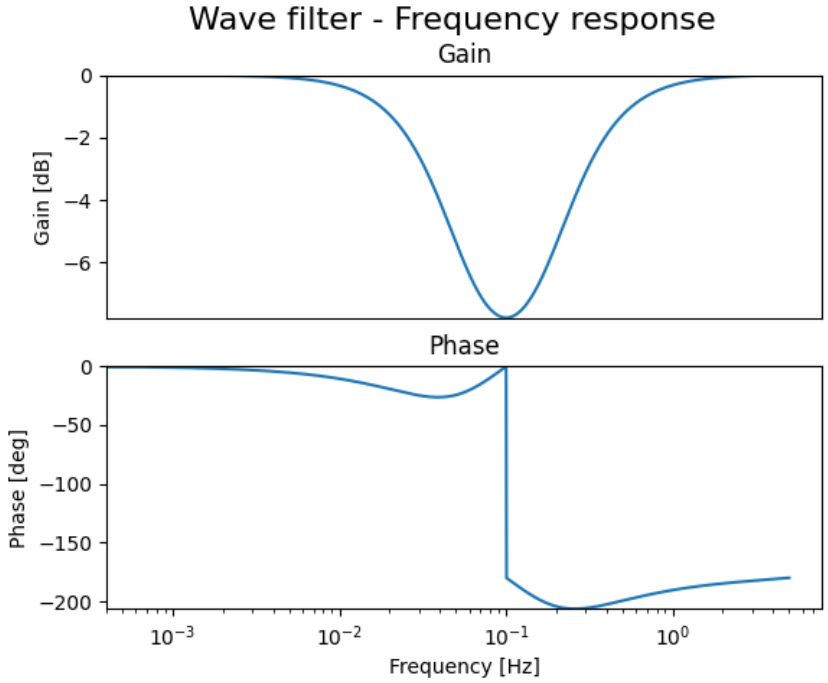
Applying 2.15 to 2.14 results in the following discrete transfer function:

$$h_n(z) = \frac{a_n + b_n \cdot z^{-1} + c_n \cdot z^{-2} + d_n \cdot z^{-3} + e_n \cdot z^{-4} + f_n \cdot z^{-5} + g_n \cdot z^{-6}}{h_n + i_n \cdot z^{-1} + j_n \cdot z^{-2} + k_n \cdot z^{-3} + l_n \cdot z^{-4} + m_n \cdot z^{-5} + n_n \cdot z^{-6}} \tag{2.16}$$

where the coefficients are described until the second decimal place and are the following: $a_n = 0.75$; $b_n = -4.68$; $c_n = 12.17$; $d_n = -16.89$; $e_n = 13.18$; $f_n = -5.48$; $g_n = 0.95$; $h_n = 0.71$; $i_n = -4.51$; $j_n = 11.959$; $k_n = -16.87$; $l_n = 13.39$; $m_n = -5.67$; $n_n = 1$.

The bode plot of the notch filter can be seen in Figure 13.

Figure 13 – Bode plot for the notch wave filter



Source: Elaborated by the author.

2.2.2.2 Noise filter

To bypass the noise added by the sensors, a number of techniques can be used. These include: kalman filters, time-frequency filters and frequency filters. **The approach chosen is frequency filters, more specifically, a low-pass filter**. Although the low-pass filter may not be the best option between the mentioned techniques (because it does not filter out low frequency noise), when the cut-off frequency is tuned to give the best results, it can work well, by finding the right trade-off between filtering high frequency noise and not removing real signal. This is the technique chosen mainly to avoid adding more complexity to the system, considering the purpose of the monograph is not to construct a high-performing GNC system, but a sufficiently representative scenario.

The filtered state X_f is expressed, in the z domain, by:

$$X_f(z) = h_{lp}(z)X(z) \tag{2.17}$$

where h_{lp} is a 6^{th} order dgital *Butterworth* low-pass filter[9] with cut-off frequency $w_f = 0.10625\ Hz$ and sampling period T_s:

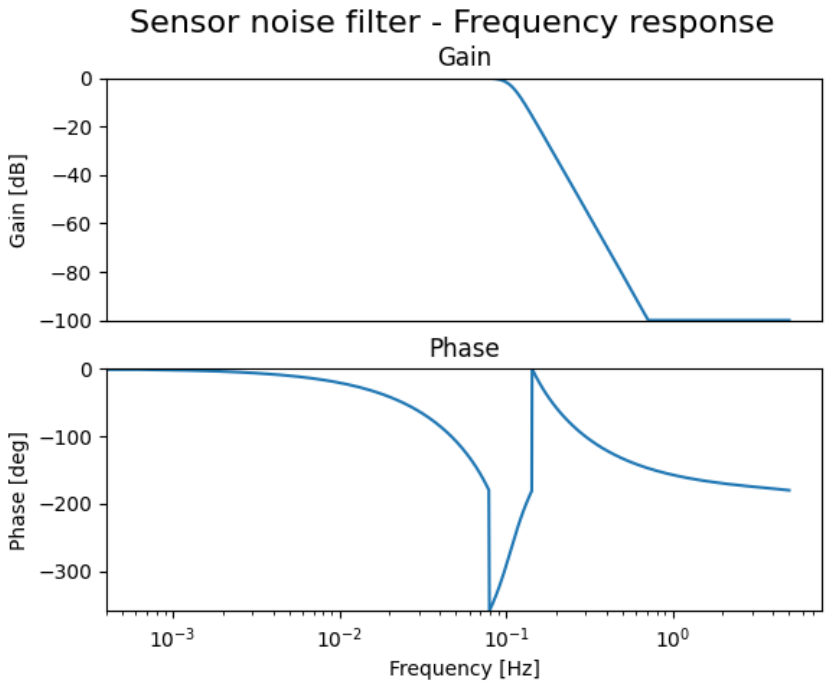
[9] A filter which attenuates frequencies above a specific threshold.

$$h_{lp}(z) = \frac{1}{h_{lp} + i_{lp} \cdot z^{-1} + j_{lp} \cdot z^{-2} + k_{lp} \cdot z^{-3} + l_{lp} \cdot z^{-4} + m_{lp} \cdot z^{-5} + n_{lp} \cdot z^{-6}} \quad (2.18)$$

where the coefficients are described until the second decimal place and are the following: $h_{lp} = 0.77$; $i_{lp} = -4.83$; $j_{lp} = 12.61$; $k_{lp} = -17.55$; $l_{lp} = 13.74$; $m_{lp} = -5.74$; $n_{lp} = 1$.

The bode plot of the low-pass filter can be seen in Figure 14.

Figure 14 – Bode plot for the low-pass noise filter



Source: Elaborated by the author.

2.2.2.3 Filters in cascade

When the Wave Filter is used in combination with the Noise Filter, they act in cascade. The input to the Noise Filter is the output of the Wave Filter.

The filtered state X_f is expressed, in the z domain, by:

$$X_f(z) = h_{lp}(z) h_n(z) X(z) \quad (2.19)$$

2.2.3 Control

The job of the Control system, in the GNC setting, is to make the controlled variables reach the set-point given by the Guidance system, as fast and stable as possible. Firstly, it is necessary to model the system as a dynamical system; then identify the parameters by experimenting with the real system[10]; and finally, establish a control law to control state variables to the desired value. **In this case study (and generally) it is used a closed-loop control system**. The control system gets the current state information from the Navigation system, and based on the difference between the desired state and the current state (the error), it calculates the control signal that is input to the dynamical system.

2.2.3.1 Dynamical system

"It is well known that coupled nonlinear differential equations are needed to fully represent the complicated ship maneuvering dynamics" (TZENG; CHEN, 2002). However simplifications are usually implemented by linearization and decoupling of the system. The following description, of the steps and assumptions required to get to the simplified[11] models used in the case study, is based on Fossen (2011).

The Nonlinear Maneuvering Equations are expressed, in matrix form, as:

$$\underbrace{M_{RB}\dot{\nu} + C_{RB}(\nu)\nu}_{\text{rigid-body forces}} + \underbrace{M_A(\nu_r)\nu_r + C_A(\nu_r)\nu_r + D(\nu_r)\nu_r}_{\text{hydrodynamic forces}} + \underbrace{g(\eta) + g_o}_{\text{hydrostatic forces}} = \tau_{act} + \tau_{ext} \quad (2.20)$$

where: $\tau_{act} = \left[\tau_{1,act},\ \tau_{2,act},\ \tau_{6,act}\right]^T \in \mathbb{R}^3$ is the vector of forces and moments acting on the craft, generated by the actuators, expressed by the craft's coordinate system and $\tau_{ext} = \tau_{wind} + \tau_{wave} = \left[\tau_{1,ext},\ \tau_{2,ext},\ \tau_{6,ext}\right]^T \in \mathbb{R}^3$ is the same as the aforementioned, with the difference that it's generated by external agents, more specifically *wind* and *waves*; $\nu_r = \nu - \nu_c$ is the relative velocity vector, where ν_c is the *current* velocity vector; M_{RB}, C_{RB}, M_A, C_A and D are matrices.

With the following **simplifying assumptions:**

- restoring forces (hydrostatic forces) negligible;
- coriolis and centripetal forces linear around fixed cruise speed U;
- linear damping (the non-linear component of the damping matrix $D(\nu_r)$ is zero) which implies ν being small.
- ocean current is negligible;

[10] In the case of this monograph, the real system is the *pydyna* simulator.
[11] These models are also very popular.

- starboard-port symmetry (which is symmetry between both sides of the ship, illustrated in Figure 15).

Figure 15 – Marine positions terminology diagram

Source: Specialist ().

Equation 2.20 reduces to two decoupled equations, Equation 2.21 (in component form) and Equation 2.25 (in matrix form).

2.2.3.1.1 Surge model

Consequently, the surge model is given by:

$$(m - X_{\dot{u}})\dot{u} - X_u u = \tau_{1_{act}} + \tau_{1ext} + w_u \tag{2.21}$$

Where: $\tau_1\ [kN] = \tau_{1,act} + \tau_{1,ext}$ is the x axis force, in the craft's reference frame, where $\tau_{1_{act}}$ is the force caused by the actuators and τ_{1ext} the force caused by external disturbances; $X_{\dot{u}} = -3375\ t$ and $X_u = -86\ \frac{kN}{(m/s)}$ are the added mass and linear damping coefficients respectively; $m = 40415\ t$ is the mass of the craft; $u\ \left[\frac{m}{s}\right]$ is the surge velocity of the craft used in simulation (craft is illustrated in Figure 7); and w_u is the process noise that accounts for unmodelled dynamics.

However, the most common approach is to consider the non-linear damping component $-X_{|u|u}u|u|$ where $X_{|u|u} = -8.36\ \frac{kN}{(m/s)^2}$ [12] for the surge velocity equation, because assuming u being small ($|u| < 2\ m/s$) deviates substantially from real applications. Additionally, τ_{1ext} is not directly

[12] Assuming sideslip angle $\beta = constant = 0$ as a simplification.

modelled due to the complexity involved, therefore this term introduces relevant process noise. This noise merges with w_u to form the resultant process noise w_{τ_1}, as follows:

$$(m - X_{\dot{u}})\dot{u} - X_u u - X_{|u|u} u|u| = \tau_1 + w_{\tau_1} \tag{2.22}$$

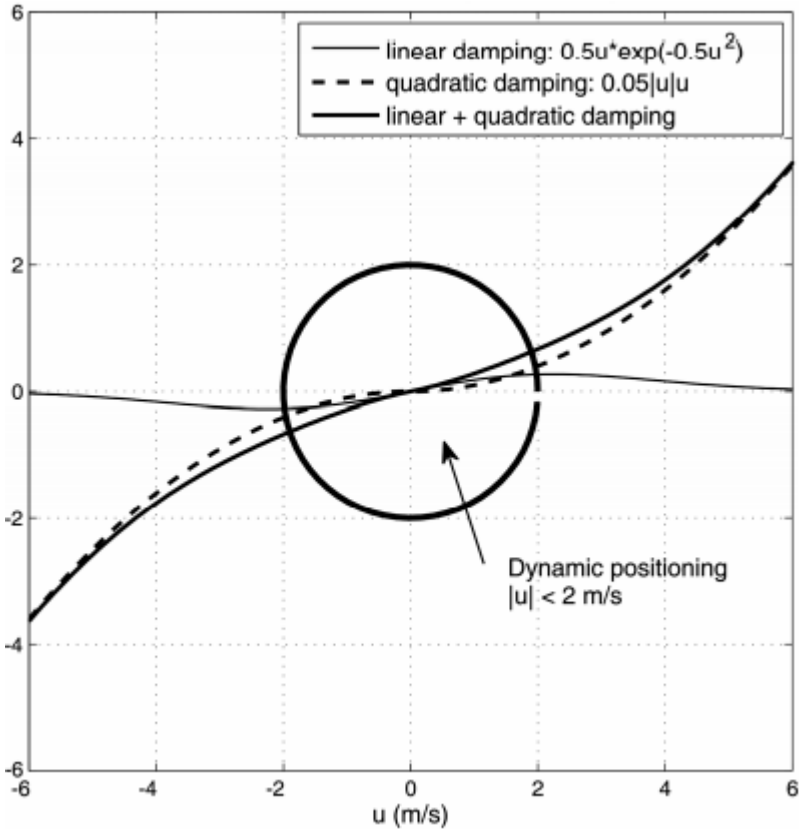
The linear damping coefficient is given by (FOSSEN, 2011):

$$X_u = ce^{\left(-\frac{u^2}{2}\right)} \tag{2.23}$$

where c is constant.

As illustrated in Figure 16, for $|u| > 2\ m/s$, $X_u \approx 0$. Since the path-following task is imposed to work mostly on the $|u| > 2\ m/s$[13] regime, the term X_u can be discarded.

Figure 16 – Linear and quadratic damping and their regimes



Source: Fossen (2011).

[13] All *waypoints have velocity above* $2\ m/s$.

Therefore, the final model used for surge velocity control in the case study is:

$$(m - X_{\dot{u}})\dot{u} - X_{|u|u}u|u| = \tau_1 + w_{\tau_1} \tag{2.24}$$

The surge model, Equation 2.24, is based on two coefficients: $X_{\dot{u}}$ and $X_{|u|u}$. The first one is the added mass, whereas the second is the drag coefficient. Both of them are assumed given[14].

2.2.3.1.2 Yaw model

The yaw model is given by:

$$M\dot{\nu}_{sway-yaw} + N(u_o)\nu_{sway-yaw} = b\delta \tag{2.25}$$

where: $\nu_{sway-yaw} = [\mathrm{v}, r]^T \in \mathbb{R}^2$ is the velocity vector for sway and yaw only; δ is the rudder angle and $b\delta = \tau^T_{sway-yaw} = [\tau_2, \tau_6]^T$, where b is a vector. The equation is valid when: $\nu_{sway-yaw}$ is small; and $u \approx u_o = constant$. Here, the process noise ends up hidden inside the coefficients, because they are identified via empirical data (2.2.3.2)[15] .

Equation 2.25 has 2 DOF which makes it inherently more complex. With the earlier assumption that r (1) and v (2) are small, this equation can be transformed into two second order decoupled equations for sway and yaw. By choosing to use the assumption (1) while freeing assumption (2), and choosing to use the assumption (2) while freeing assumption (1); for sway and yaw respectively.

The transfer function of the second order decoupled yaw equation, derived from Equation 2.25, considering δ as the input with r as the output, can expressed as:

$$\frac{r}{\delta}(s) = \frac{K(1 + T_3 s)}{(1 + T_1 s)(1 + T_2 s)} \tag{2.26}$$

where: T_1, T_2 and T_3 are time constants; and $K = 1.6e-5$ $[dimensionless]$ is the static gain.

Finally, since in practice $T_3 \approx T_2$, the zero nearly cancels with the pole (TZENG; CHEN, 2002), and the *First Order Nomoto Model* can be expressed as:

$$\frac{r}{\delta}(s) = \frac{K}{1 + Ts} \tag{2.27}$$

[14] The values of the coefficients are extracted from the vessel configuration file
[15] This identification was done without the influence of waves.

and therefore:

$$\frac{\psi}{\delta}(s) = \frac{K}{s(1 + Ts)} \tag{2.28}$$

and in the time domain:

$$T\ddot{\psi} + \dot{\psi} = K\delta \tag{2.29}$$

where $T = T_1 + T_2 - T_3 = 133\ s$. The parameters K and T are identified by analyzing the step response.**The *First Order Nomoto Model* is used in the study case for yaw control**.

2.2.3.2 Model Identification

Calculating the coefficients of Equation 2.29 with precision is neither practical nor trivial, due to difficulties in micro-scale theoretical modelling, a range of complicated experiments needed[16] and the uncertainties involved. A popular approach is to treat the coefficients as unknown parameters, and estimate them based on empirical data related to the system response, using known predefined inputs. This is one type of *Model Identification* that places more focus on theoretical analysis and gives rise to a *Light-Gray-Box Model*, illustrated in Figure 17 (ISERMANN; MüNCHHOF, 2011). **In the aforementioned case, a previous structure for the differential equations is given from theoretical modelling; and is the approach used in the case study**.

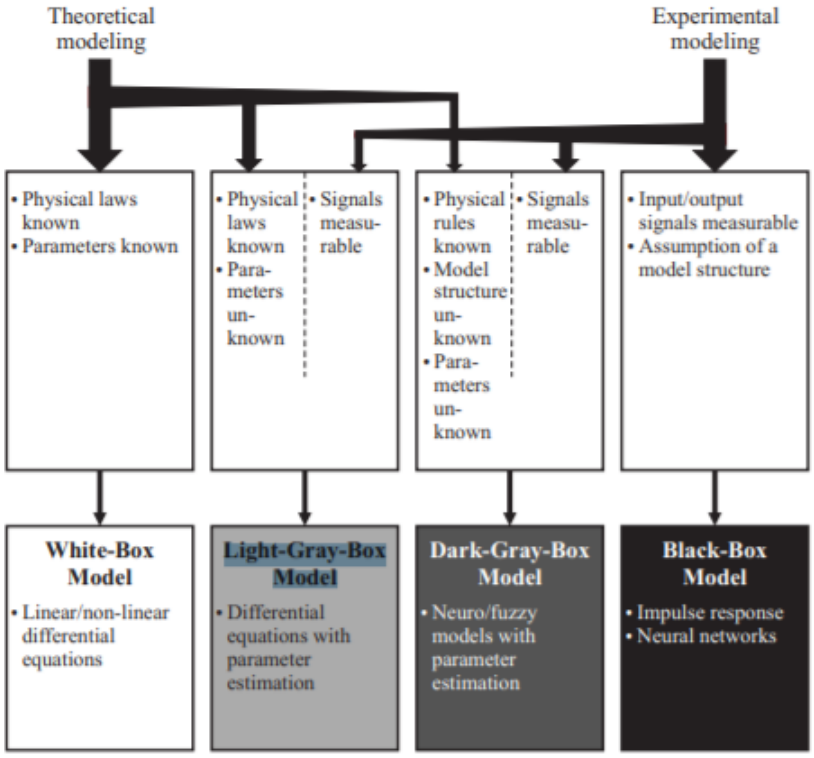
Model Identification methods can be divided broadly into two categories according to the setting in which they are performed: *off-line* and *on-line* methods. *Off-line* approaches estimate the system, through experimentation, before it is deployed for control; and *on-line* approaches estimate the system in real-time, at the same time the system is being controlled (LIU *et al.*, 2016). **In this case study, it is used the *off-line* approach, by assuming that the simplified model is sufficiently robust to model all situations of the craft[17]; and by the fact that there is easy access to experimentation, since it is all done in simulation**.

Other useful separations can be drawn between methods based on the working domain, parametrization, linearity, time modelling and number of controlled variables. These are described below:

- **working domain:** *time-domain* and *frequency-domain*;
- **parametrization:** *parametric* and *non-parametric*;
- **linearity:** *linear* and *non-linear*.

[16] Experiments to get general coefficients, not related to the specific system as a whole.
[17] Therefore, e.g., does not need to keep changing the linearized model for each operating point, during the operation of the craft.

Figure 17 – Different kinds of mathematical models as a result of the amount theoretical and experimental influence



Source: Isermann and Münchhof (2011).

- **time modelling:** *continuous-time* and *discrete-time*.
- **number of controlled variables:** *Single Output Single Input (SISO)*, *Single Input Multiple Output (SIMO)*, *Multiple Input Single Output (MISO)* and *Multiple Input Multiple Output (MIMO)*.

Based on the established categories, the requirements present for the identification of Equation 2.29 in case study are: the use of a *linear* method, since the model is linear; and the use of SISO methods, since surge and yaw motions are decoupled.

The coefficients of Equation 2.24 are not identified in the case study. Instead, they are extracted from the vessel configuration file[18]. This is because the equation is non-linear and identifying the coefficients is a much harder task, judged beyond the scope of this monograph.

[18] This file is mandatory for performing the simulation.

2.2.3.3 Controllers

There are a number of methods for Controller design, but the most popular one, for it's simplicity and good results, is the *Proportial Integral Derivative (PID)* method. This method is intended for linear systems, thus it can be applied directly to the heading model derived in 2.2.3.1, for yaw control. In this case, since the system can be represented as a transfer function, the controller design is usually done in the complex domain, by satisfying predefined control criteria which are based on the natural frequency w_n and the damping ratio ζ of the closed-loop or resultant system.

However, the surge velocity model is non-linear, thus this method cannot be applied directly. **The method chosen for surge velocity is *Sliding Mode Control***. This technique basically cancels the nonlinear terms and makes the sliding surface variable s converge to zero. The sliding surface variable is related to the variable of interest (velocity u) such that when s goes to zero, u goes to the set-point. In this case, since the system cannot be represented as a transfer function, the controller design must be done initially in the time domain.

After the controllers are theoretically designed, they are further tuned two times to obtain the best response: first with a closed-loop surge control, using *pydyna* as the dynamical system; and second in the production path following architecture, where the other modules are present. The values for the parameters, in the following sections, are the final values obtained after these procedures.

2.2.3.3.1 Yaw Controller

The continuous-time **yaw controller** is of the form:

$$\delta(\tilde{\theta}) = -K_p \cdot \tilde{\theta} - K_d \cdot \dot{\tilde{\theta}} - enable(\tilde{\theta}) \cdot \left(K_i \cdot \int_0^t \tilde{\theta}(\tau) d\tau \right) \tag{2.30}$$

But, since it is a digital implementation, becomes:

$$\delta(\tilde{\theta}) = -K_p \cdot \tilde{\theta}_k - K_d \cdot \frac{\tilde{\theta}_k - \tilde{\theta}_{k-n}}{t_k - t_{k-n}} - enable(\tilde{\theta}_k) \cdot \left(K_i \cdot \sum_{i=0}^{k-1} \tilde{\theta}_i \cdot \Delta_t \right) \tag{2.31}$$

where a bigger n makes the derivative approximation less subjective to noise but increases delay. The value used is $n = 1$ due to the presence of noise removing filters.

with:

$$0 \leq |\delta| \leq SM \cdot radians(35) \tag{2.32}$$

and:

$$enable(\tilde{\theta}) = \begin{cases} 1 & \tilde{\theta} \leq C \\ 0 & \tilde{\theta} > C \end{cases} \tag{2.33}$$

with the error being:

$$\tilde{\theta} = \begin{cases} \theta - \theta_d & |\theta - \theta_d| \leq \pi \\ -(\pi - (\theta \,\%\, \pi - \theta_d \,\%\, \pi)) & \theta - \theta_d > \pi \\ \pi + (\theta \,\%\, \pi - \theta_d \,\%\, \pi) & \theta - \theta_d < -\pi \end{cases} \tag{2.34}$$

where: $\theta = \frac{\pi}{2} - \psi$ if $\frac{\pi}{2} - \psi > 0$ else $\theta = 2 \cdot \pi - (\frac{\pi}{2} - \psi)$[19] is the yaw angle measured from east counterclockwise; θ_d is the desired yaw angle or yaw set-point; $K_p = 1.6 > 0$ $[dimensionless]$, $K_d = 65 > 0$ $[s]$ and $K_i = 0.00075 > 0$ $[1/s]$ are the controller gains; $enable(\tilde{\theta})$ is the anti-windup strategy with $C = 0.1$ as the integration threshold; δ is in radians and $0 \leq SM = 0.99 \leq 1$ is the safety margin coefficient used to avoid actuator wear; $\Delta_t = 0.1$ is the time step of the simulator; and $radians(x)$ is a function that converts degrees to radians. The values for the parameters where found after these procedures.

An illustration of the mentioned controller, with an additional *wind feed-forward* term (which is not present in the case study), and in conjunction with a LOS-based guidance can be seen in Figure 18. The *wind feed-forward* term is the torque to compensate for wind disturbances, which is calculated by an added estimation module.

An algorithm for computing the gains is also described by Fossen (2011):

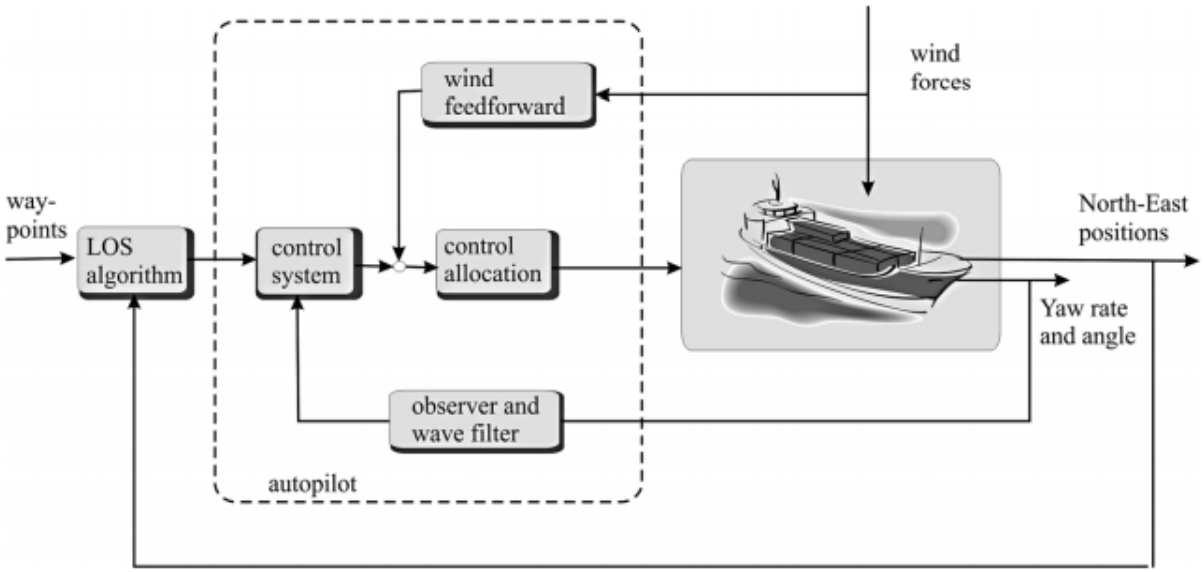
Algorithm 1 – *PID* gains for a yaw controller

procedure YAWCONTROLLERPIDGAINS(w_b, ζ) ▷ $w_b > 0$ and $\zeta > 0$
  $w_n \leftarrow \frac{1}{\sqrt{1 - 2\zeta^2 + \sqrt{4\zeta^4 - \zeta^2 + 2}}} w_b$
  $K_p \leftarrow m w_n^2 - k$ ▷ Proportional Gain
  $K_d \leftarrow 2\zeta w_n m - d$ ▷ Derivative Gain
  $K_i \leftarrow K_p \frac{w_n}{10}$ ▷ Integral Gain
  return (K_p, K_d, K_i)
end procedure

where: w_b is the desired bandwidth[20] of the closed-loop transfer function; ζ is the desired damping ratio of the closed-loop transfer function, which commonly follows the rule of thumb $\zeta = 0.8$; $m = \frac{T}{K}$ is the mass; $d = \frac{1}{K}$ is the damping coefficient; and $k = 0$ is the spring coefficient; for the *First Order Nomoto Model* (Equation 2.29).

[19] *pydyna gives the yaw angle of the craft as θ, not ψ*
[20] Frequency at which the magnitude of the transfer function is equal to -3dB.

Figure 18 – GNC System with *LOS* guidance and *wind feed-forward*



Source: Fossen (2011).

2.2.3.3.2 Surge Controller

The **surge controller** is designed using *Sliding Mode Control*. First, some definitions: the controlled variable x receives $x := u\ (velocity)$ and u is free to be used as the system's input. Next, it is important to explain that the form of the resultant controller will be the same for any point located between the same *waypoints*. This means that the controller changes every time the craft reaches the next *waypoint* and remains constant until reaches the next one.

The sliding surface $s(x)$ is $s(x) = \left(\frac{d}{dt} + \lambda\right)^{n-1} \cdot \tilde{x}$ where λ is a parameter and $\tilde{x} = x - x_{des}$ is the error, where x_{des} is the desired velocity of the next *waypoint*. Since the order n of the system is $n = 1$, the surface reduces to the error, so then:

$$s(x) \ = \ \tilde{x} \tag{2.35}$$

Deriving the surface with respect to time gets $\dot{s} = \dot{\tilde{x}} = \dot{x}$ since x_{des} is a constant. Therefore, $\dot{s} = f(x) + u$ where $f(x) = \frac{X_{|x|x|x|}}{(m - X_{\dot{x}})} + w_{\tau_1}$ and u is the input with:

$$\tau_1 \ = \ u \cdot (m \ - \ X_{\dot{x}}) \tag{2.36}$$

The continuous-time controller is defined as:

$$u \ = \ - \hat{f}_p \ - \ k \cdot sat\left(\frac{s(x)}{\phi}\right) \tag{2.37}$$

where: $\hat{f}$ is the estimate of f and $\hat{f}_p$ is the point estimate of f. With the minimum value that guarantees slipping in Equation 2.38. The digital version of the controller does not present significant notation changes, besides changing from a continuous signal $x(t)$ to a discrete signal $x[k]$. Therefore, although the digital version is the one that is implemented, the notation of the continuous version can be maintained for simplicity.

$$k = |\eta| + F \tag{2.38}$$

where: F is the uncertainty bound; η can be thought as the imposed acceleration[21]; and $-k \cdot sat\left(\frac{s(x)}{\phi}\right)$ is used in place of the theoretical best controller $-k \cdot sign(s(x))$ to avoid chattering of the input. Developing $\hat{f}_p$:

$$\hat{f}_p = [f] = \frac{[X_{|x|x}]}{[m] - [X_{\dot{x}}]} \cdot x|x| = \frac{-8.36}{40415 - (-3375)} \cdot x|x|$$

$$\therefore \hat{f}_p = -1.9091 \cdot 10^{-4} \cdot x|x| \tag{2.39}$$

The *sat* regards the saturation function and ϕ is a parameter that balances the smoothness of the input close to the sliding surface and the control guarantees (by diminishing the time it takes to reach the sliding surface and imposing a steady-state error bounded by $|\phi|$). The value chosen is $\phi = 0.19$ after the tuning procedures explained here, which is just enough to avoid chattering and maintains to great extent the method's initial control guarantees.

The parameter η is obtained as follows: $\eta = \frac{s(0)}{t_r} = \frac{x - x_{des}}{t_r} = \frac{x - x_{des}}{t_{est}}$, where t_r is the time it takes for $s = 0$, which is relaxed to $s > -\phi$ due to the boundary layer imposed by the *sat* function[22], and substituted by t_{est} which is the estimated time from one *waypoint* to the next. This happens since the problem is not time constrained, it is actually spatially constrained. Therefore, the time between *waypoints* can be estimated knowing roughly the velocity function.

[21] Assuming the theoretical best controller.
[22] The sat function is $sat(x) = max(-1, min(1, x))$.

Using the theoretical best controller $-k \cdot sign(s(x))$ the velocity should have an linear behaviour between the *waypoints*. But by using $-k \cdot sat\left(\frac{s(x)}{\phi}\right)$ the response get smoother and goes towards an exponential behaviour at the end. With this in mind, the estimated velocity is a weighted average between the estimated time with linear t_{estLin} response and estimated time with exponential response t_{estExp} dictated by ϕ:

$$t_{est} = \frac{\phi \cdot t_{estExp} + (\Delta_x - \phi) \cdot t_{estLin}}{\Delta_x} \tag{2.40}$$

where $\Delta_x = x_{des} - x_{wayOld}$ with x_{wayOld} being the surge velocity of the craft when it crossed the last *waypoint*.

The caculation of t_{estLin} is simply:

$$t_{estLin} = \frac{d}{\left(\frac{x_{wayOld} + x_{des}}{2}\right)} \tag{2.41}$$

where d is the euclidean distance between *waypoints*.

The caculation of t_{estExp} is done by integrating the velocity function between *waypoints*[23] to get the distance function; and solving $d = distance(t_{estExp})$ for t_{estExp}. Developing the last expression:

$$distance(t) = \int velocity(t)\,dt + C = \int [x_{wayOld} + (x_{des} - x_{wayOld}) \cdot (1 - exp(-t \cdot k))]\,dt + C \tag{2.42}$$

with the initial condition $distance(0) = 0$ which leads to $C = -\left(\frac{1}{k}\right) \cdot (x_{des} - x_{wayOld})$

$$d = distance(t_{estExp}) = (x_{wayOld} \cdot t_{estExp} + (x_{des} - x_{wayOld}) \cdot (t_{estExp} + \left(\frac{1}{k}\right) \cdot exp(-t_{estExp} \cdot k)) + C \tag{2.43}$$

is solved for t_{estExp} using a numerical solver.

However, there is another problem. The aforementioned approach assumes the craft has zero angle relative to the line connecting the *waypoints*, which is generally not the case. To solve this, an empirical approach was adopted. A linear model for correcting the estimated time was derived based on the craft's behaviour. When the craft reaches a *waypoint* and starts following another line, the more angled it is relative to the steady state angle of the craft when following the aforementioned line, larger the time.

[23] This step is performed assuming the model is linear for simplicity, the quadratic term $x|x|$ is for this step just x.

The linear correction model is the following:

$$t_{estCorr} = C_{corr} \cdot \left(\frac{|\theta_{dif}|}{2 \cdot \pi} + 1 \right) \cdot \left(\frac{1}{\cos(\theta_{ss})} \right) \cdot t_{est} \tag{2.44}$$

with:

$$\theta_{dif} = \begin{cases} \theta_{wayOld} - \theta_{ss} & |\theta_{wayOld} - \theta_{ss}| \leq \pi \\ -(\pi - (\theta_{wayOld} \,\%\, \pi - \theta_{ss} \,\%\, \pi)) & \theta_{wayOld} - \theta_{ss} > \pi \\ \pi + (\theta_{wayOld} \,\%\, \pi - \theta_{ss} \,\%\, \pi) & \theta_{wayOld} - \theta_{ss} < -\pi \end{cases} \tag{2.45}$$

where: $C_{corr} = 0.7$ is a constant; θ_{ss} is the steady state yaw angle the craft in path line (measured counterclockwise from east); θ_{wayOld} is the yaw angle of the craft (measured counterclockwise from east) when it crossed the last *waypoint*.

The value of θ_{ss} is calculated based on the angle of the path line ζ, (measured counterclockwise from east) and the sway velocity when it crossed the last waypoint v_{wayOld}. The steady state surge velocity x_{ss} of the craft is approximated to be the desired velocity of the next *waypoint* x_{des} and the steady state sway velocity v_{ss} is approximated to be the projection of x_{wayOld} and v_{wayOld} onto the unit vector which draws 90 degress counterclockwise from the line path. With the steady state velocities, it is possible to calculate the steady state yaw angle.

First, the steady state surge velocity:

$$x_{ss} = x_{des} \tag{2.46}$$

the steady state sway velocity:

$$v_{ss} = cos\left(\frac{\pi}{2} - \Delta_\theta\right) \cdot x_{wayOld} + cos(\Delta_\theta) \cdot v_{wayOld} \tag{2.47}$$

where $\Delta_\theta = \zeta - \theta_{wayOld}$.

The steady state beta angle:

$$\beta_{ss} = \arcsin\left(\frac{v_{ss}}{\sqrt{v_{ss}^2 + x_{ss}^2}} \right) \tag{2.48}$$

and finally the steady state yaw angle:

$$\theta_{ss} = \zeta - \beta_{ss} \tag{2.49}$$

The parameter F accounts for the uncertainty in $\hat{f}$, expressed in the term w_{τ_1}. This can refer to uncertainty in the parameters or unmodelled dynamics. Since the parameters for the surge model were obtained from ground truth, this type of uncertainty is not the major concern in this context. The major concern is about unmodelled dynamics, since the real model is coupled, stochastic and full of environmental disturbances among other complexities involved. The model used for control is a strong simplification, that does not account for the aforementioned effects. Therefore, the objective of F is to counteract these unmodelled forces. The final value $F = 0.08$ was found after the tuning procedures explained here.

By substituting Equation 2.39, Equation 2.38 and Equation 2.35 in Equation 2.37 with the values for F and ϕ, and defining $params = (x_{wayOld},\ \theta_{wayOld},\ \zeta,\ x_{des})$ the final control law obtained in Equation 2.50.

$$u(params) = 1.9091 \cdot 10^{-4} \cdot x|x| - \left(\left| \frac{\tilde{x}}{t_{estCorr}(params)} \right| + 0.08 \right) \cdot sat\left(\frac{\tilde{x}}{0.19} \right) \tag{2.50}$$

2.2.3.4 Control allocation

Control Allocation is the subsystem responsible for mapping the generalized forces, obtained from the Controllers, to the inputs of the dynamical system (FOSSEN, 2011). In this case, the yaw control, obtained from the *First Order Nomoto Model* (Equation 2.28), has already it's output equal to the input of the dynamical system. However, for surge control, control allocation is needed.

Pydyna implements the common constrained (due to saturation of the actuator) Main Propeller, that is, a propeller that produces forces only in the x direction in the craft's coordinate frame. Therefore, the dynamical system expects a control model adequate for this purpose.

There are several actuator models for thrusters and propellers, depending on the craft actuator implementation. **A quadratic actuator model, given by Equation 2.51, is chosen for the study case, since it is the model used by TPN (TANNURI, 2002).**

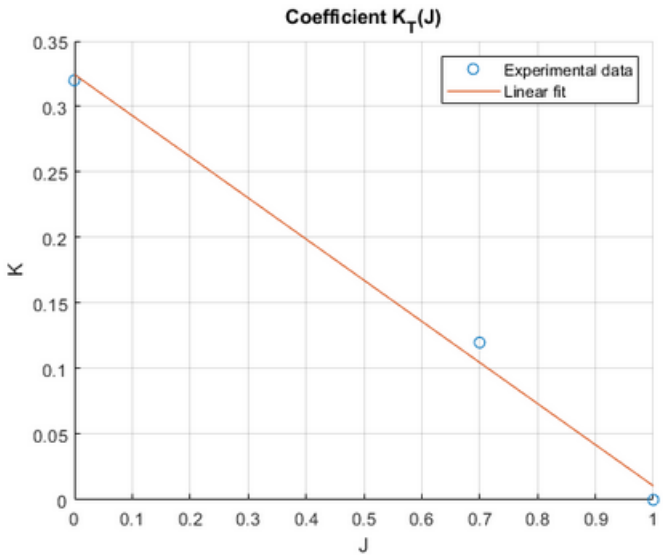
$$\tau_1 = \rho D_p^4 K_T(J) n_p |n_p| \tag{2.51}$$

where: τ_1 $[kN]$ is the thrust force; $\rho = 1.0245\ \frac{t}{m^3}$ is the specific mass of the water; $D_p = 7\ m$ is the diameter of the propeller; $K_T(J)$ is a coefficient dependent on u where $J = \frac{u}{n_p D}$; and n_p $[Hz]$ is the propeller rotation.

The values of $K_T(J)$ are found by a linear fit using three known[24] data points: (0; 0.32), (0.7; 0.12) and (1; 0). The plot of $K_T(J)$ is shown in Figure 19 and the formula is described in Equation 2.52.

[24] Experimental results found by TPN-USP.

Figure 19 – Coefficient K_T



Source: Elaborated by the author.

$$K_T = 0.3246 - 0.3139 \cdot J \tag{2.52}$$

By isolating n_p in Equation 2.51 and substituting the values of coefficients, n_p is obtained as a function of τ_1 in Equation 2.53.

$$n_p = \begin{cases} c_1 \cdot \sqrt{(c_2 \cdot u^2 + \tau_1)} + c3 \cdot u & \tau \geq 0 \\ -c_1 \cdot \sqrt{(c_2 \cdot u^2 - \tau_1)} - c_3 \cdot u & \tau < 0 \end{cases} \tag{2.53}$$

where $c_1 = 0.036$, $c_2 = 3.53$, $c_3 = 0.067$ and u is the surge velocity.

With:

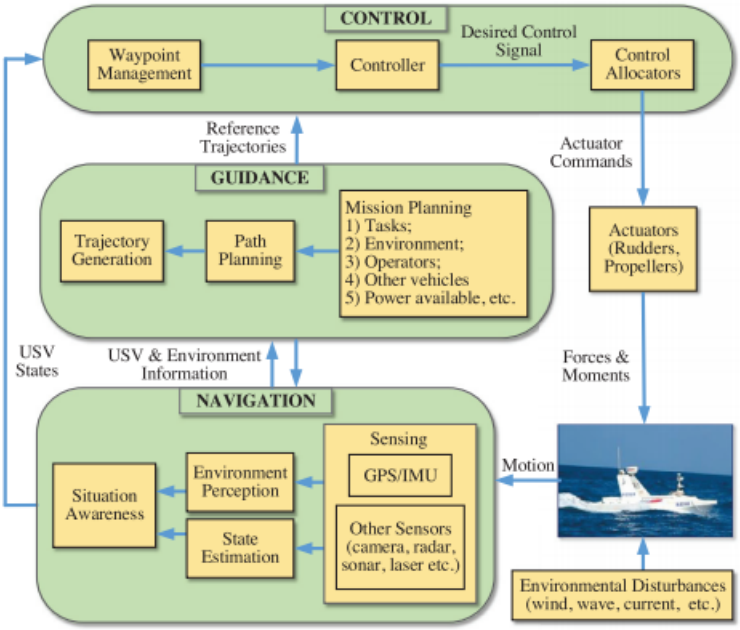
$$0 \leq |n_p| \leq SM \cdot 1.75 \tag{2.54}$$

due to propeller saturation. Where $0 \leq SM = 0.99 \leq 1$ is the safety margin coefficient used to avoid actuator wear.

2.2.4 Summary

A summary of a typical GNC system structure can be found in Figure 20.

Figure 20 – typical GNC system structure



Source: Liu *et al.* (2016).

2.3 Computational tools

Two important computational tools are used in this project: *pydyna* and *ROS*. These are explained briefly in the following sections, along with the most important aspects of each.

2.3.1 pydyna

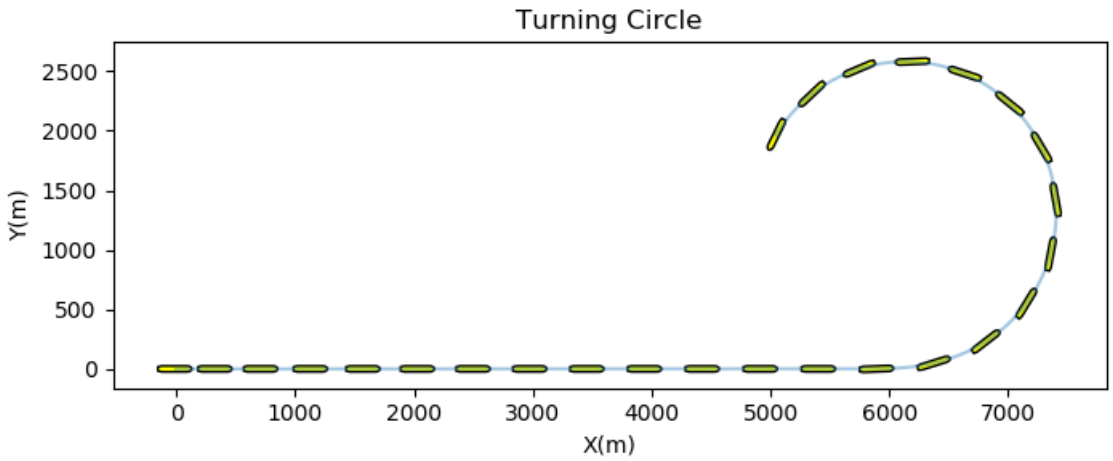
As already stated in the scope of the project, *pydyna* is a maneuvering simulator developed by TPN and found as a *Python* library, but it's core is written in *C++* and goes by the name *dynasim*.

So far the library is still under development, therefore it's use is quite restricted at the moment. The only people allowed access are TPN members or researchers with granted permission.

The first releases of *pydyna* were only for *Windows* OS. Due to more and more fronts of research involving *pydyna*, there are also newer versions for *Linux* OS, however they are still considered to be experimental.

For every step of the simulation *dynasim* is able, through a fourth order *Runge-Kutta* numerical integration, to output the position of the vessel . These outputs can be displayed either as an image (illustrated in Figure 21) or as a `.txt` file.

Figure 21 – Example of *pydyna's* visualisation for a given simulation



Source: TPNSHIP (2021).

The simulation must be created alongside a configuration file that contains vessel parameters. This other file may contain a variety of vessels, and as soon as the simulation is created, any of them can be used.

For the simulation, the system requires an initial state vector (initial positions and velocities) and some configuration parameters. Such parameters include the time interval between every step, and the maximum number of steps.

The trajectory of the vessel is described according to the inputs given, which can be set before the simulation or during it. E.g., all the inputs to the propellers and rudders could be given beforehand, in a static file; or these might be calculated at run-time by a controller.

2.3.2 ROS

ROS is defined as an *Meta-Operating System (MOS)* for robotics software development. This means that it runs on top of an OS, however, provides functionalities of a typical OS and also of a typical application framework (HASAN, 2019). In *ROS2* all main OSs are supported, and it is developed in *Python* and *C++*.

The Robotics Back-End (2021) breaks down *ROS* as two main components, presented next.

- **Core *middle-ware* with communication tools:** infrastructure for distributed processes;
- **Set of *plug-and-play* libraries:** reusable tested software for common usage. E.g., these can be drivers, algorithms, visualizations tools or simulators.

The goals of the MOS are to solve problems related to the complexity of developing and maintaining complex robotic systems infrastructure and applications. Three of these goals are stated, as described in Wiki (2020) and The Robotics Back-End (2021):

- standardize robotics software development;
- provide code reuse;
- provide an infrastructure that supports distributed processes and abstracts low-level device control.

Wiki (2020) describes some important *ROS* features that make robotics software development in the platform much more practical, these are described below.

- **ROS agnostic libraries:** developing ROS libraries is totally independent of ROS[25];
- **Language agnostic processes:** processes running within *ROS* can be written in any language, although *Python* and *C++* are the main interfaces;
- **Scaling:** due to it's distributed nature, it can scale well to a complex robotic system setting.

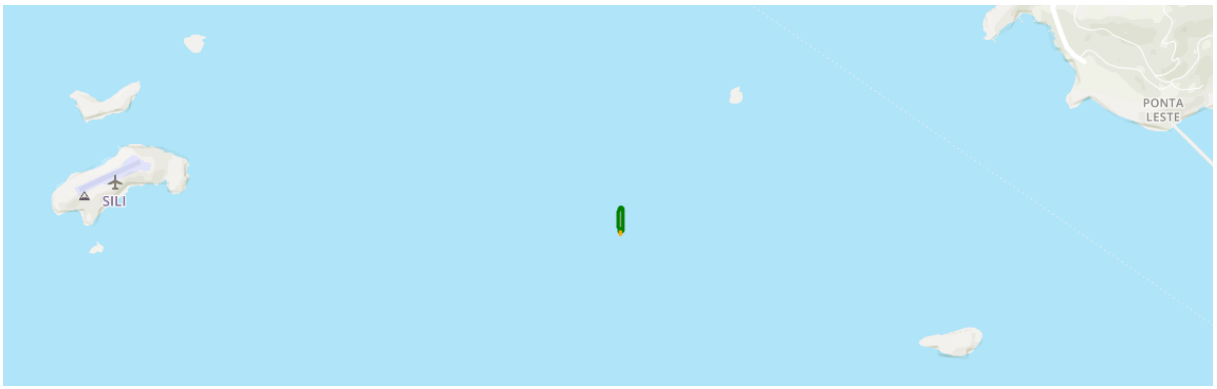
[25] In the sense that the core software of the library is independent, of course it depends on *ROS* to be integrated as a *ROS* package.

Some of the best use-cases are for research and prototyping purposes. In research, the focus is usally in some area of the robotic system, but requires that a whole infrastructure is set-up for testing the solution. *ROS2* let's the researcher focus on the area of interest, while providing ready-to-use software for the other parts. In prototyping, the goal to to build the solution fast. With *ROS2* knowledge this can be done without being an expert in all areas (THE ROBOTICS BACK-END, 2021). **These use-cases relate a lot, not only with this monograph, but especially with the research environment in which *pydyna* is already used**.

2.3.3 Venus

Venus is a web-based visualization tool, developed by the TPN team. It is a python library specialized for sea-related objects. It uses *MapQuest* as a third-party service, which provides the map images. **The tool is used in the case study to visualize the craft in real-time.**. An example of the craft visualized with *Venus* can be seen in 22.

Figure 22 – *Craft visualized with Venus*



Source: Elaborated by the author.

CHAPTER

3

STATE OF THE ART

3.1 Related Work

As the main objective of this work is to develop an integration between *pydyna* and *ROS2*, it is of utmost importance to verify whether similar integrations have been done before. Considering *pydyna* is not an open-source software, it is safe to say that this integration was never made before; however the general concept is to find maritime simulators implemented in the platform. Therefore, the bibliographic review is focused on USV's and AUV's in combination with *ROS2* implementations over the past 5 years.

As was established in conjunction with the professor advisor, the first 17 articles (ordered by relevance) were selected in *Scopus*. The query used was the following:

"TITLE-ABS-KEY (usv OR "unmanned surface vehicle" OR asv OR "autonomous surface vehicle" OR "Maritime Autonomous Surface Ships") AND (ros OR "Robot Operating System") AND (LIMIT-TO (SUBJAREA , "COMP") OR LIMIT-TO (SUBJAREA , "ENGI")) AND (LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019) OR LIMIT-TO (PUBYEAR , 2018) OR LIMIT-TO (PUBYEAR , 2017) OR LIMIT-TO (PUBYEAR , 2016))"

3.1.1 Findings

The table constructed can be found in Appendix A. The most important findings, relative to this project, will be presented below:

- *Gazebo* and *Qt ROS* integrations are used tools that provide Graphical User Interface (GUI) (VELAMALA; PATIL; MING, 2017) (SMITH; DUNBABIN, 2019);
- if necessary to use a third-party service, which is not wrapped as a *ROS* package a *ROS* bridge can be used (VELAMALA; PATIL; MING, 2017);
- one of the most common guidance techniques, implemented in *ROS* for USVs, is the line-of-sight algorithm (VILLA; AALTONEN; KOSKINEN, 2020);
- the main sensor used for USVs and ASVs is LiDAR, often in an obstacle avoidance setting. There are already simulator implementations of the aforementioned (VILLA; AALTONEN; KOSKINEN, 2020); (KRUPINSKI; MAURELLI, 2018).
- the GNC system can be composed by the following modules: path planner, path manager, path following, autopilot; which then control the USV (IOVINO; SAVVARIS; TSOURDOS, 2018);
- being an application agnostic framework, *ROS* can easily incorporate existing software modules by simply implementing them in a node or wrapping them in a package that can contain one or more nodes (VILLA; AALTONEN; KOSKINEN, 2020)(VELAMALA; PATIL; MING, 2017);
- existing other ASV and USV simulators software have been integrated in *ROS* (FEDORENKO; GURENKO, 2016) (SMITH; DUNBABIN, 2019);
- sensor high-fidelity modelling has been created and implemented using *ROS*, for *hardware-in-the-loop* simulations. (SMITH; DUNBABIN, 2019).

3.1.2 Discussion

The most blatant output that came from this first research would be that existing projects can be integrated into *ROS* without great difficulty. Software can be implemented as a node, or being wrapped in packages. Specifically, as integrations of other USV and ASV simulators have been made before, there should not be to many inherent obstacles for this project.

Also, it is important to highlight the possibilities of using *hardware-in-the-loop* simulations. Through already implemented *ROS'* applications, a user can emulate sensors, allowing realistic data gathering, which is likely to get results closer to the ones found with the physical sensor hardware.

As for the GNC architecture, some concepts were clarified. The path planner would be based on the inputs of destination, obstacles and map, that can be given to the system both from sensors and a database. Based on this planner, the waypoints generated are then used by the path manager, creating a path definition. In turn, this input is then used in the path following, that defines the speed and heading of the USV, leading to the autopilot. Finally, the autopilot gives the servo-commands to the USV.

Currently, *pydyna's* interface is not user-friendly to those that have little knowledge of programming or system modelling, such as ship operators or newcomers; nor does it provide a practical and insightful way of analyzing the ship and its surroundings [1]. As mentioned earlier, tools such as *Gazebo* and *Qt* can provide GUI's, making *TPN's* software's usability better, for a wide range of people.

3.2 *ROS* Surface Vehicle packages

A search for *ROS* packages involving maritime applications is necessary for understanding the current situation of maritime simulators inside *ROS* environment. It is important to examine the type of package, the amount of attention it has from the community, information regarding it's functioning and/or use and relevant features regarding Surface Vehicle (SV) simulation.

This search helps to gain insight on the relevance of integrating a new state-of-the-art ship maneuvering simulator in *ROS*. Furthermore, these existing packages can be essential learning material for developing the current package, theme of this monograph.

The search was done in *ROS Index*, a recent *ROS* project[2], as well as in *GitHub*. For *ROS Index* the following search entries were used separately: maritime, boat, ship, usv, asv. For *GitHub* the aforementioned entries were used in combination with *ROS*.

The project type can be obtained by the *GitHub* repository *about* section (if there is one for the repository); number of *stars* and *forks* can be used as a proxy for community adoption; information about the package is a boolean: if the repository has a reasonable *README* file or points to some other informative resource it is true, else it is not[3]; and relevant features are obtained scanning the information provided.

[1] Information obtained with Humberto Makiyama, one of *pydyna's* coordinators.
[2] Still in beta fase.
[3] Reasonable is subjective, however it's in the sense that installation and a basic use of the package are minimum requirements.

3.2.1 Findings

The table constructed can be found in Appendix B. The most important findings, relative to this project, will be presented below:

- all packages are related to small vessel USV's, not large boats or ships;
- communication with non *ROS2* resources is not necessary;
- some packages are self-contained, meaning they provided a complete set of components such as motion simulation, control, sensing and visualization;
- many packages are based on *Gazebo usv plugins* and *Virtual RobotX Maritime Challenge* simulator;
- the most relevant resources are related to *Virtual RobotX Maritime Challenge* (offered by the organization or by participating teams);
- most packages don't present large use from the community, neither have decent documentation.

3.2.2 Discussion

The fact that none of the packages are related to ship simulation shows the current state of autonomous SVs efforts. These efforts are mainly restricted to Maritime Robotics Competitions, especially *RobotX* virtual competition. The advantage is that teams use the base simulator resources from the organization and build on top of it. Control, sensing, visualization and other capabilities are then added to the simulation. However, development does not reach larger scales, such as large boats or ships. Considering this is a promising area of development, having a state-of-the-art ship simulator at disposal for the community could open a new door for future work and research. In consequence, this work could then help TPN's fast-simulation and autonomous ship efforts.

About the construction of the package itself, it is clear that most of the necessary tools can be obtained within the *ROS2* environment, even further, the package can not only encapsulate the simulator but also other modules such as control, sensing and visualization; for a more robust simulation.

The use of *Gazebo usv plugins* and *Virtual RobotX Maritime Challenge* resources may be a path to follow in this project, however all the *physics* of *pydyna* core simulator would need to be provided to the simulator and it would increase the complexity of the project. Since the exact dynamical model that governs *pydyna* is not available to the authors and unnecessary complexity may damage the project, using a purely visualization tool appears to be the best approach.

Due to most relevant resources being related to the *Virtual RobotX Maritime Challenge*, exploring the competition, as well as, the participating teams' projects is worthwhile.

Finally, considering most of the packages offer little to no documentation, they can be used as a secondary resource option, mainly for advanced development efforts, when *ROS2* knowledge and experience is sufficient. Furthermore, presenting a package with good documentation for newcomers can be very powerful in this setting.

CHAPTER

4

PROBLEM FORMALIZATION

4.1 Objectives

The objectives of this project are divided into two categories: primary objectives and extensions. Primary objectives are the main goals of the project and need to be reached. Extensions are bonus achievements, on top of the main goals of the project.

4.1.1 Primary objectives

The primary objectives can be listed as follows.

1. Make available a *ROS2* package that enables usage of *pydyna* in a modular manner, within the whole system. So that the high-fidelity maneuvering simulator could be used in combination with other *ROS2* packages, in a practical and scalable way.
2. Demonstrate a native integration of the maneuvering simulator with *ROS2*, together with a GNC system and visualization capabilities; through a path-following task.
3. Describe how the software should be used.

4.1.2 Extensions

The extensions can be listed as follows.

1. Make available a *ROS2 path-following* package. The package would include the TPN maneuvering simulator, controllers and one or more of the following: sensor emulation and visualization tool.
2. Private maintenance documentation: besides a documentation for general usage, a documentation for maintenance.
3. Distribute the package to the scientific community[1]

4.2 Requirements

The following are requirements of the project.

- **User documentation:** the project must have a documentation for any potential user. This is mainly how to interact with *ROS2* interfaces to run the simulation;
- **OS support:** the project must be developed and tested on *Ubuntu 18.04* or *Windows 10*, to avoid potential compatibility issues, since *pydyna* was built and tested in these settings;
- **Case study requirements:** the case study needs to be sufficiently representative of a real TPN maritime application. The use of hardware-in-the-loop for navigation and a visualization tool represent important contributions to the existing software and should be present. A metric, along with a pre-defined minimum value for it[2], is required for the evaluation of the path-following task. Important to state that this metric should not depend on time, since the path-following motion objective consists solely on the defined path spatial constraints.

[1] Depends also on TPN's decision to make the package public or not.
[2] Assuming a metric that is subject to maximization.

4.3 Technical specifications

The following list states specifies software requirements of the project that should be installed in the machine:

- *Windows 10*;
- *Python 3.6.8*;
- *pydyna* [3];
- *venus* [4];
- *ROS2 galactic*;
- *Chocolatey*;
- *vcredist2013*;
- *vcredist140*;
- *OpenSSL*;
- *Visual Studio 2019*;
- *DDS implementations*;
- *OpenCV*;
- *CMake*;
- *Qt5*;
- *Graphviz*;
- *xmllint*;
- *Python* public libraries listed in Appendix C.

[3] *pydyna* is private to TPN
[4] *venus* is private to TPN

CHAPTER

5

METHODS

5.1 *ROS2* terminology

In order to understand the packages, some of *ROS2* terms must be explained:

- ***node***: independent processes that are able to send and receive data from each other;
- ***topic***: one of the means in which nodes can send (by publishing to a *topic*) and receive (subscribing to a *topic*) data;
- ***service***: another mean of communication between *nodes* with a request and response pattern. Can be asynchronous or synchronous. In our application only asynchronous services are used;
- ***msg* and *srv* files**: *msg* files define the data structure for *topics* and *srv* files for services. There are standard libraries with basic data types and structures available for use, but custom implementations are also needed in the application;
- ***rosbag***: tool that subscribes to topics and writes a bag file with the contents of all messages published on those specified topics. A bag file is essentially a *sqlite* database.

5.2 The *pydyna_ros* package

This section is the user documentation of the *pydyna_ros* package package.

5.2.1 Overview

The *pydyna_ros* package[1] has the objective of integrating *pydyna* to *ROS2*. This package contains the *pydyna_ros node*[2], which runs the simulation. The package also includes a *launch* file, that should be used to launch the *node* and consequently the simulation. A *config* folder contains the necessary archives to support *pydyna's* functioning (the main files ares *.p3d* file of the vessel, and the *.whl* file of the *pydyna* library). The code used in the package can be found in Appendix D.

5.2.2 Requirements

The requirements can be found in the following list:

- *Python 3.6.8*;
- *ROS2 Galactic*;
- *pydyna*[3];
- *venus*[4];
- *Python* public libraries listed in Appendix C.

5.2.3 Features

With this package, the user is able to start the simulation with a *request* using a service containing the initial state of the vessel, propeller rotation and yaw angle. After the simulation is initialized, the user can give two inputs to *pydyna*: propeller rotation and the rudder angle. The *pydyna* node subscribes to these two inputs as *topics*, runs one step of the simulation only when it has received both of these inputs, and publishes the next state of the vessel to the *state topic*. Ending the simulation is also an option and can be done with one of two *topics*: *end* or *shutdown*. The last relates to the *path_following* package that will presented afterwards.

To start a simulation, a request has to be sent using the service `InitValues.srv`, as shown in Source code 1. This service is a custom *srv* file. This file contains a request and response in *yaml* format. Request and response are separated by a dashed line. The request contains

[1] So far the package is still under development, and is called *pydyna_simple*
[2] Just like the package, the Node is also under development and called *pydyna_simple node*
[3] *pydyna* is private to TPN
[4] *venus* is private to TPN

four properties: `initial_state`, `waypoints`, `surge` and `yaw`. The `initial_state` property contains the initial state of the vessel (in the same format that will be explained for the *state topic* in the next section). The `waypoints` property is not used in tis case. The `surge` and `yaw` properties are the inital values for propeller rotation and rudder angle, respectively. The values to the right are the default values for the properties.

The three, mentioned earlier ,*topics* are: `propeller_rotation`, `rudder_angle` and `state`; and are defined by their *msg* files. Propeller rotation and rudder angle use the standard library *Float32 msg* file. The state, seen in Source code 2, uses a custom *yaml-styled msg* file that contains `time`, `position` and `velocity` properties, which are *msg* files by themselves. Position is a set of three *Float32* properties: `x`, `y` and `theta`, seen in Source code 3. Velocity, in the same way, is defined by having `u`, `v` and `r`, seen in Source code 4. The values to the right are the default values for the properties.

All custom *msg* and srv files are defined and in a separate package for flexibility. This package is called *path_following_interfaces*. After building the aforementioned package, these data structures can be imported in nodes as *Python* objects. The code for this package can found in Appendix F.

Source code 1 – InitValues.srv

```
1: #request
2: State initial_state
3: Waypoints waypoints
4: float32 surge 0.0
5: float32 yaw 0.0
6: ---
7: #response
8: float32 surge 0.0
9: float32 yaw 0.0
```

Source code 2 – State.msg

```
1: # 3DOF state of the craft
2: Position position
3: Velocity velocity
4: float32 time 0.0
```

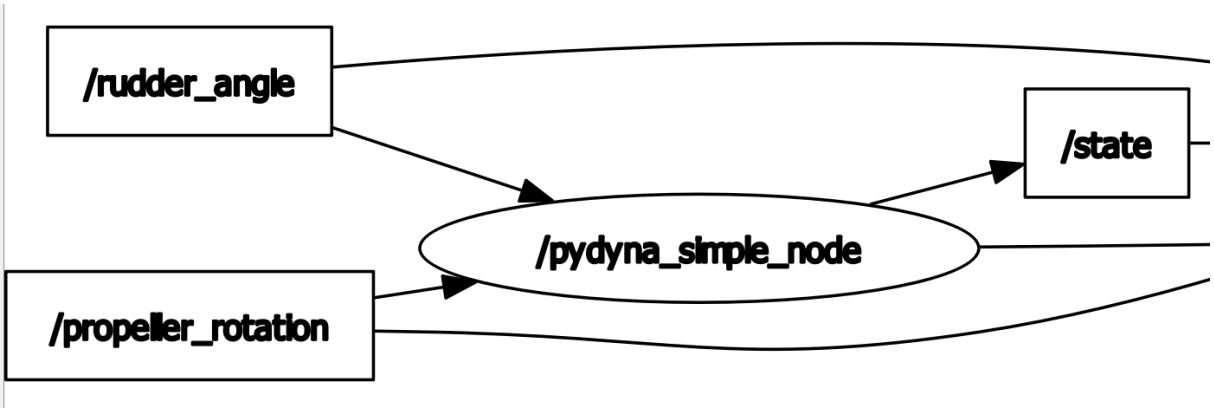
Source code 3 – Position.msg

```
1: # positions (earth-fixed reference frame)
2: float32 x 0.0
3: float32 y 0.0
4: # 1.57079632679 radians = 90 degrees
5: float32 theta 1.57079632679
```

Source code 4 – Velocity.msg

```
1: # velocities (craft-fixed reference frame)
2: float32 u 1.0
3: float32 v 0.0
4: float32 r 0.0
```

Figure 23 – *pydyna_simple topics* in graph



Source: Elaborated by the author.

5.2.4 Getting Started

For the setup of the *ROS2* and *pydyna* environments, it's necessary for the user to follow the steps regarded in the *ROS2 Galactic* and *TPN*[5] pages respectively. Then, the user must clone this repository.

1. To build packages, run the following command in `main_ws/src` with the x64 Prompt for Visual Studio 2019 terminal as admin:

```
          ~/tcc-autonomous-ship/src/main_ws/src>colcon build --merge
   -install
```

[5] TPN page is private

2. On the newly created install directory, run[6]:

```
~/tcc-autonomous-ship/src/main_ws/install>call setup.bat
```

3. Next, run the launch file. This will create *rosbags* in the same directory. Therefore, the recommendation is to run it in `main_ws/install/share/pydyna_simple/db` which is intended to store *rosbags*.

```
~/tcc-autonomous-ship/src/main_ws/install/share/
pydyna_simple/db>ros2 launch pydyna_simple
```

4. In order to verify active nodes and topics, the user may run:

```
~/>ros2 topic list -t
```

```
~/>rqt_graph
```

5. With the *pydyna_simple node* active, the user can, from the command line, start the simulation with the `init_simul` *service*, send *topics* for `rudder_angle` and `proppeller_rotation` and listen to `state`; each one of them in a separate terminal[7].

   a) start simulation by making *request* to `start_end_simul`:

```
~/> ros2 service call /init_simul
path_following_interfaces/srv/InitValues "{}"
```

   b) publish to `rudder_angle`:

```
~/> ros2 topic pub --once /rudder_angle std_msgs/
msg/Float32 "{data: 0}"
```

   c) publish to `proppeller_rotation`:

```
~/> ros2 topic pub --once /propeller_rotation
std_msgs/msg/Float32 "{data: 1}"
```

   d) subscribe to `state`:

```
~/> ros2 topic echo /state
```

[6] For every new terminal, this command must be run to setup the workspace environment
[7] The values used here are simply for the sake of testing, and can be changed

5.3 Case study: path-following ship

This section exposes the methods used in the case study as well as contains user documentation for the path-following package. First, the evaluation metrics are discussed, then the software architecture is exposed, followed by an explanation of the sensor emulation block and the documentation for the path-following package.

5.3.1 Evaluation

The necessity of reaching exactly the defined *waypoints* is not a requirement in this setting. Therefore, the notion of reaching the next *waypoint* can be relaxed to touching a circle of acceptance around it. When the craft enters the circle, the LOS algorithm considers it has already reached the *waypoint* and starts guiding to the next *waypoint*. This configuration is beneficial to the actuators since it smoothens the path.

The main evaluation matric is the *cross-track* error cte of the path followed by the craft relative to the desired desired LOS path (lines connecting the *waypoints*). Also, it is necessary to assure that the craft reaches the final *waypoint*, therefore the along-track error ate must be smaller than the final radius of acceptance, at the end of the simulation. In addition, the *width* error we is introduced. The aforementioned is the *cross-track* error measured at the end-point of the ship further from the line it has to follow. It measures the maximum distance from any point of the ship to the line connecting the *waypoints*. Both of the mentioned errors are positive when the craft is above the current path line, and negative when the craft is below the current path line.

Although the main objective is the calculation of the mean *cross-track* error, there are situations in which there are problems. Some of these are: the vessel reaches a circle of acceptance and then for some reason turns and starts going back; or if the vessel passes the next *waypoint* without reaching it. To adress these situations, the craft automatically changes it's current desired waypoint to match the position where the craft is. This means that, for example, if the craft passes a waypoint without reaching it, it does not keep trying to reach it at the costs of ruining the path following task, instead, it simply changes the desired waypoint to the next one.

The general scenario can be visualized in Figure 25, however, it is important to point out that the schematic is not drawn to scale. Where *inside_area* is the place the craft will generally be, and the *cross-track* error is measured normally. If the craft is in *inside_area* of the next *waypoint*, it will change it's desired *waypoint* to the following one, however, the *cross-track* will remain being measured relative to the line inside *inside_area*. If the craft is in *after_area* it will change it's desired *waypoint* to the next one, and will measure *cross-track* relative to the line inside *after_area*. The same procedure applies when the craft is in *inside_area* of the last *waypoint* or in *before_area*, but in practice these situations don't occur, therefore are not coded.

The following conditions must be satisfied in order to judge the path-following algorithm as successful:

$$ate \leq R_{wFinal} \tag{5.1}$$

where R_{wFinal} is explained right before Equation 5.4. The simulation shuts down automatically if the craft has $ate \leq R_{wFinal}$ at the last *waypoint*.

$$cte_m < \frac{beam}{2} \tag{5.2}$$

Where: cte_m is the mean *cross-track* error; and $beam = 32.2\ m$ is the width of the vessel used in this project, at it's widest point. Also illustrated in Figure 24.

The radius of acceptance R_w is typically in the range:

$$0 \leq R_w \leq R_{los} \tag{5.3}$$

The value chosen for R_w provides a trade-off between reaching the *waypoints* precisely and having a smooth path. If it is small, the craft will be concerned in reaching almost exactly the waypoints, however, this causes the craft to start adjusting to the next path line too late. Since the craft has a very low response time, it drifts a lot away from the path[8]. If it is high, the craft will exhibit a very smooth path (interpolation between path lines) which can give good results concerning path error and give smoother rudder commands, however, the craft will not reach the *waypoints* by a decent amount of distance[9].

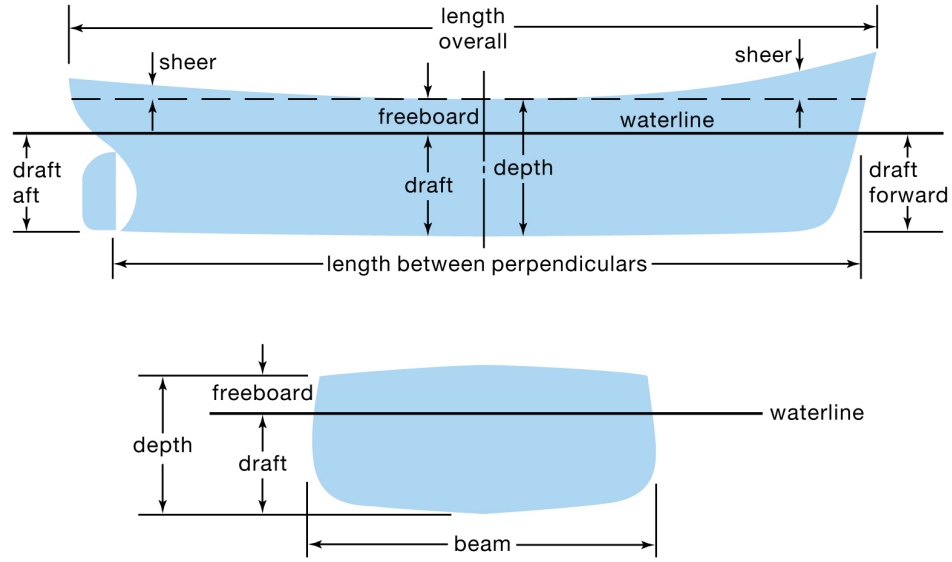
One exception is for the final *waypoint*. After reaching the final *waypoint's* radius of acceptance, the craft would end it's path, since there are no more *waypoints* to change to. For large values of R_w, this means that the craft would end it's path far away from the final *waypoint*. This is not a desired behaviour, therefore, a circle of acceptance exclusive to the final *waypoint* R_{wFinal} is introduced:

$$0 \leq R_{wFinal} \leq R_w \tag{5.4}$$

[8] When the path lines are not co-linear

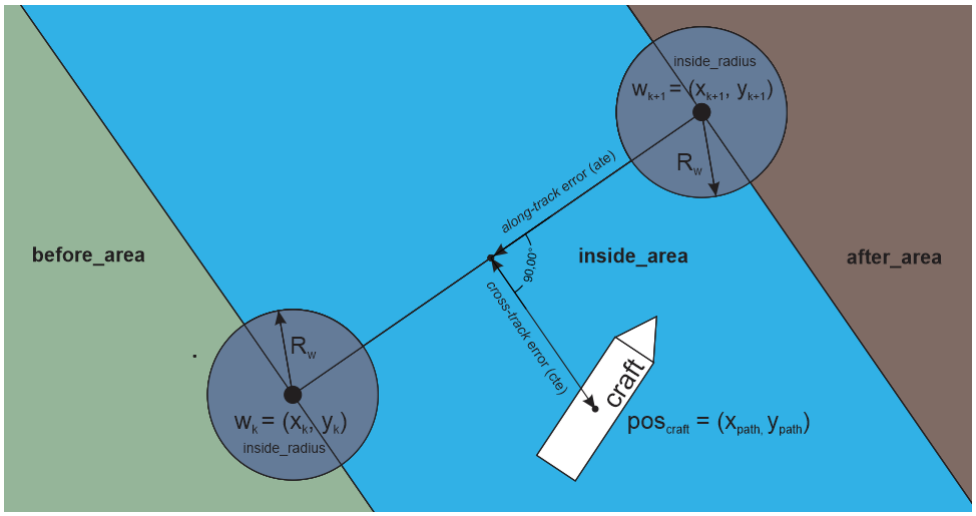
[9] Here, distance is used in a generalized manner. Not only in the xy plane, but also for velocity.

Figure 24 – Vessel dimensions



Source: Davies (2012).

Figure 25 – Schematic regarding the general scenario of the path-following task

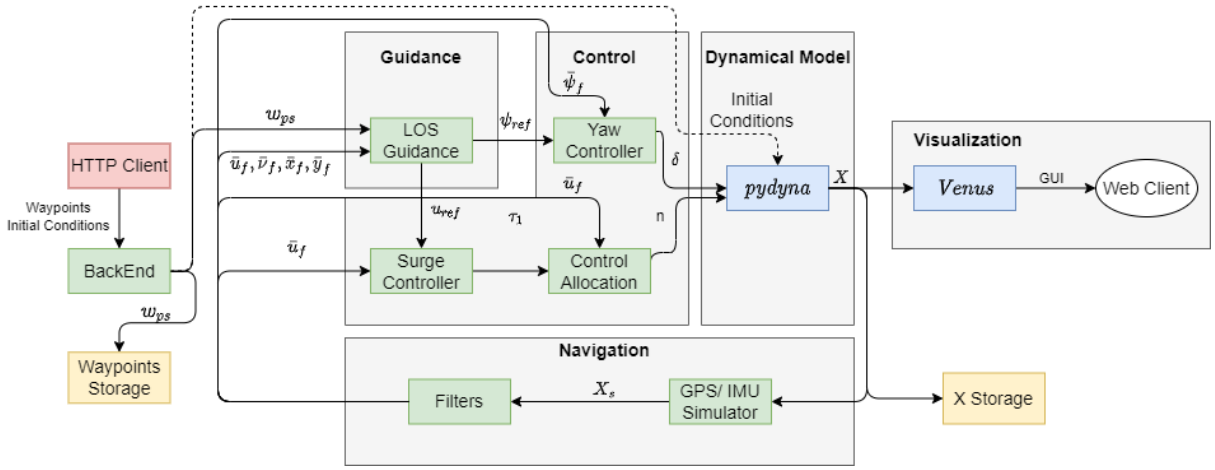


Source: Elaborated by the author.

5.3.2 Architecture

The project's main architecture is composed of five major groups: **Dynamical Model**, **Control**, **Guidance**, **Navigation** and **Visualization**; as illustrated in Figure 26. They are all explained in Chapter 2. In this environment, the controllers and filters are subject to a final tuning, guided by the error metrics, to get a better path-following behaviour. Figure 26 portrays the main interactions[10] between the blocks that compose the system:

Figure 26 – Architecture of the production environment, of the path-following system. Where: nodes in green are implemented with the use of public libraries; node in blue is implemented with the use of private libraries and public libraries; node in red is a third-party tool; nodes in yellow are files; node with no color is a web browser.



Source: Elaborated by the author.

where: w_{ps} are the *waypoints*; ψ_{ref} is the reference yaw angle; n is the propeller angular velocity; δ is the rudder angle; X is the state; X_s is the simulated state, that is, considering *hardware-in-the-loop* dynamics; $\bar{X}_s$ is the filtered simulated state; initial conditions are position η and velocity ν.

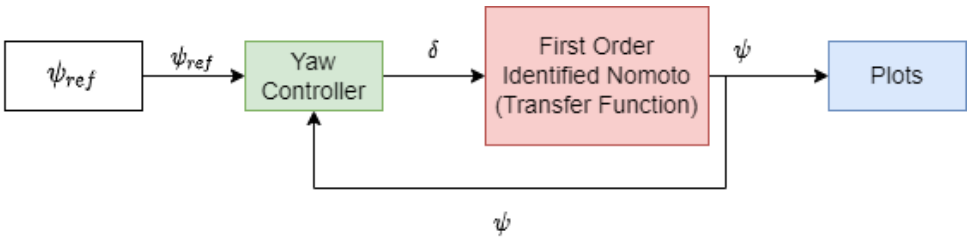
[10] there are other data exchanged between blocks in some cases, but they are not fundamental to the description of the system and would pollute the schematic.

The project makes use of other sub-architectures for the development of the blocks portrayed in Figure 26. These other architectures are explored in the rest of this section. It is important to state that the tuning environments, which will be presented in the following subsections, consider the craft subjected to wind and current, but not waves.

5.3.2.1 Yaw control development architectures

In order to control yaw motion of the vessel, the *First Order Nomoto Model* is used as the control design model. The control design environment of yaw control can be seen in Figure 27.

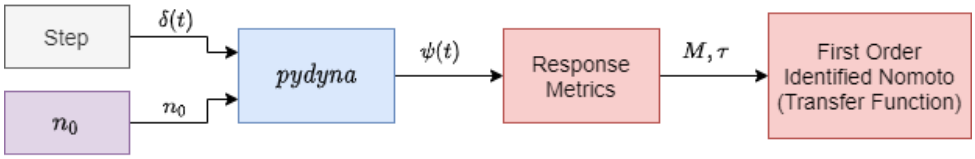
Figure 27 – Architecture of the yaw control design environment



Source: Elaborated by the author.

In order to identify the model, the gain M and the time constant τ of the first-order model are extracted from the step response. The architecture of the *Nomoto Model* identification can be seen in Figure 28.

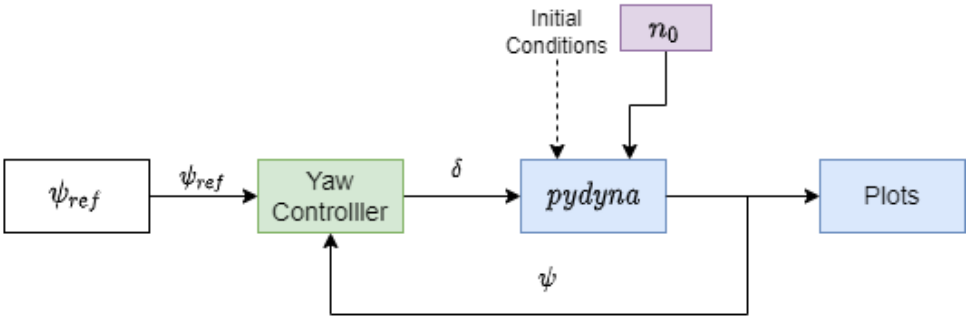
Figure 28 – Architecture of the *Nomoto Model* Identification environment



Source: Elaborated by the author.

After the controller is designed, it is tuned with the simulator to obtain a better response. The architecture for this step is illustrated in Figure 29.

Figure 29 – Architecture of the yaw control tuning environment



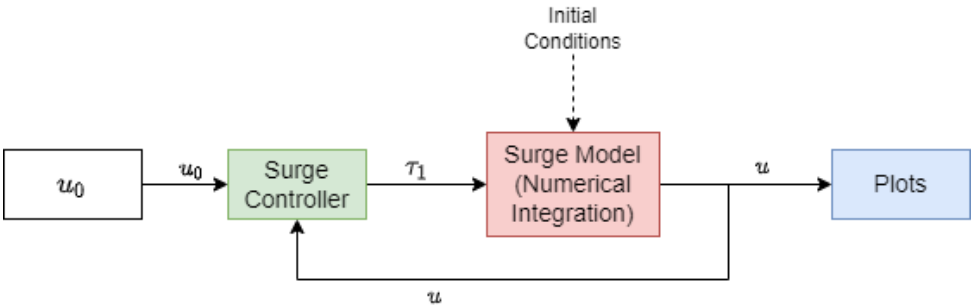
Source: Elaborated by the author.

Where n_0 is a constant propeller angular velocity. The value of n_0 can be set to 0 for simplicity of analysis.

5.3.2.2 Surge control development architectures

The surge control design environment is portrayed in Figure 30.

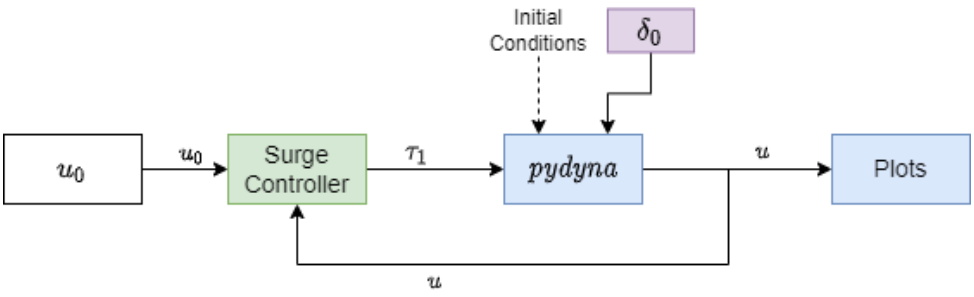
Figure 30 – Architecture of the surge control design environment



Source: Elaborated by the author.

After the controller is designed, it is tuned with the simulator to obtain a better response. The architecture for this step is illustrated in Figure 31.

Figure 31 – Architecture of the surge control tuning environment



Source: Elaborated by the author.

Where δ_0 is a constant rudder angle. The value of δ_0 can be set to 0 for simplicity of analysis.

5.4 Sensor emulation

There is a block in the architecture which has the job of emulating a suite of sensors, more specifically GPS and IMU, illustrated in Figure 26 as "GPS/IMU Simulator" block. The objective is, by introducing a *hardware-in-the-loop* component, approximate the simulation to the real system operating with non ideal sensors.

The input of the block is the actual state variables, and the output are simulated state variables. The output is the input summed with a certain zero mean Gaussian noise, depending on the variable. The variance of the noise, for each variable, is extracted from *GPS* and *IMU data-sheets*. The state transformation along with the *data-sheets* are presented below:

$$state_s = \begin{bmatrix} x_s \\ y_s \\ \theta_s \\ u_s \\ \mathrm{v}_s \\ r_s \end{bmatrix} = \begin{bmatrix} x + x_n \sim \mathcal{N}(\mu = 0, \sigma = x_\sigma) \\ y + y_n \sim \mathcal{N}(\mu = 0, \sigma = y_\sigma) \\ \theta + \theta_n \sim \mathcal{N}(\mu = 0, \sigma = \theta_\sigma) \\ u + u_n \sim \mathcal{N}(\mu = 0, \sigma = u_\sigma) \\ \mathrm{v} + \mathrm{v}_n \sim \mathcal{N}(\mu = 0, \sigma = \mathrm{v}_\sigma) \\ r + r_n \sim \mathcal{N}(\mu = 0, \sigma = r_\sigma) \end{bmatrix} \tag{5.5}$$

where: $x_\sigma = 5.46\ m$, $y_\sigma = 5.46\ m$, $\theta_\sigma = 0.05\ rad$ are "horizontal acc", "horizontal acc", "heading acc", respectively, from San Jose Technology, Inc (2021); $r_\sigma = 0.005\ rad/s$ is calculated using "random walk" from Inertial Labs (); and u_σ and v_σ are obtained from the error propagation of $\dot{x}_\sigma$ and $\dot{y}_\sigma$ as follows:

$$\begin{bmatrix} u_\sigma \\ \\ v_\sigma \end{bmatrix} = \begin{bmatrix} \sqrt{(\cos(\theta)\cdot\dot{x}_\sigma)^2 + (\sin(\theta)\cdot\dot{y}_\sigma)^2 + ((-\sin(\theta)\cdot\dot{x} + \cos(\theta)\cdot\dot{y})\cdot\theta_\sigma)^2} \\ \\ \sqrt{(-\sin(\theta)\cdot\dot{x}_\sigma)^2 + (\cos(\theta)\cdot\dot{y}_\sigma)^2 + ((\cos(\theta)\cdot\dot{x} - \sin(\theta)\cdot\dot{y})\cdot\theta_\sigma)^2} \end{bmatrix} \tag{5.6}$$

where $\dot{x}_\sigma = 0.1\ m/s$ and $\dot{y}_\sigma = 0.1\ m/s$ are "velocity acc" (industrial) from u-blox (). The formula is derived from the rotation matrix used to change from the frame $[\dot{x}_\sigma,\ \dot{y}_\sigma]^T$ to the frame $[u_\sigma,\ v_\sigma]^T$.

5.5 The *path_following* package

This section is the user documentation of the *path_following* package.

5.5.1 Overview

The *path_following* package is a complement to *pydyna_ros* package as it integrates the architecture of **Navigation**, **Guidance** and **Control** determined in subsection 5.3.2 to *pydyna's* dynamical system.

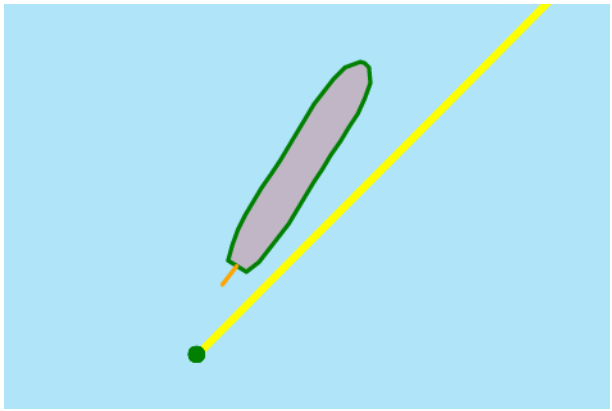
This package not only complements *pydyna* as a way of implementing it into the architecture of a path following vessel, but also includes a GUI to visualize the vessel. Given the starting state and the desired *waypoints*, the vessel will then follow a path given by straight lines connecting the *waypoints* positions, following also the surge velocity given by the *waypoints* velocities.

The ship and its path can be seen through *Venus*, illustrated in Figure 32, where the path to follow is dictated by straight lines in yellow, connecting the starting point in green and the *waypoints* in red. A close-up image of the vessel can be seen in Figure 33. The green contour represents the structure of the ship, and the orange one represents the rudder.

Figure 32 – Vessel reproduced with *Venus* in Angra dos Reis

Source: Elaborated by the author.

Figure 33 – Vessel reproduced with *Venus* in close-up



Source: Elaborated by the author.

5.5.2 Requirements

The requirements can be found in the following list:

- *Python 3.6.8*;
- *ROS2 Galactic*;
- *pydyna*[11];
- *venus*[12];
- *Python* public libraries listed in Appendix C;

5.5.3 Features

With the *nodes* active, it's possible to visualize the vessel in <http://localhost:6150>. Using the packages' HTTP API, then send requests for the start of the simulation, the initial state of the vessel, and the desired *waypoints* to <http:localhost:5000>.

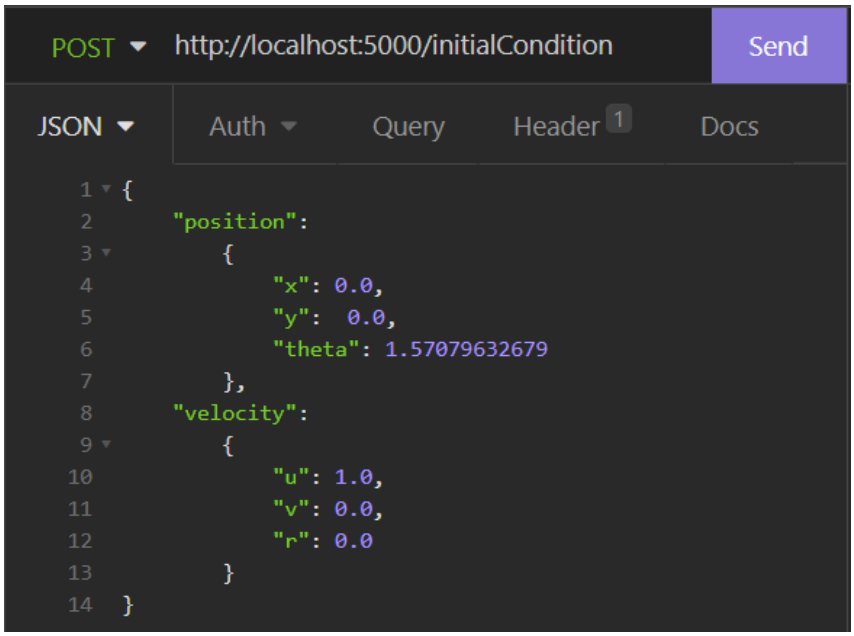
In order to run the simulation with desired parameters, the user must send two *POST* and one *GET* request. Send the initial state to `/initial_condition`, illustrated in Figure 34, and desired *waypoints* to `/waypoints`, illustrated in Figure 35. In `/waypoints`, the "`from_gui`" property, for the time being, is always zero. This is because the *waypoints* are given solely through the HTTP Client, however, in future work the value 1 will say to the backend that it should ignore the *waypoints* in the *payload*, and instead get them from the GUI's application server. Then, the user sends a *GET* request to `/start`, illustrated in Figure 36.

The user can also end the simulation in two ways: killing only *pydyna* node or killing all nodes except for *backend node*. The first one is achieved with a GET *request* to `/end`, illustrated

[11] *pydyna* is private to TPN
[12] *venus* is private to TPN

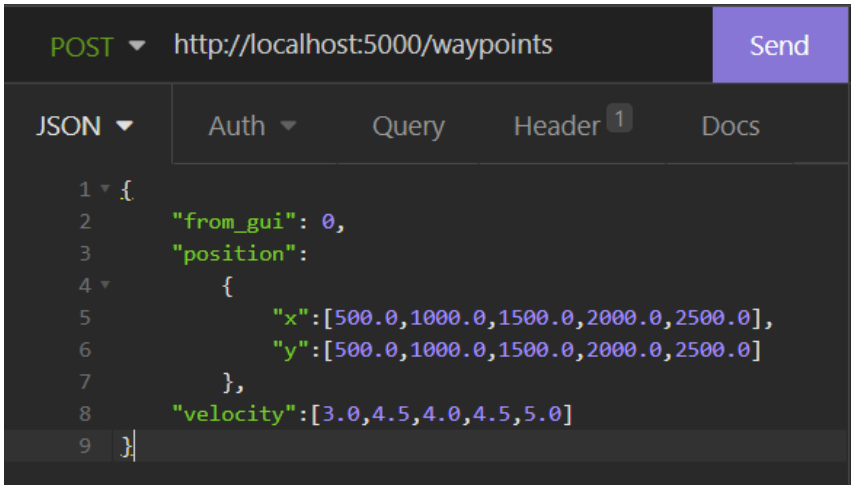
in Figure 37; while the second option is achieved with a GET *request* to `/shutdown`, illustrated in Figure 38.

Figure 34 – Example of the *POST* request for setting the vessel's initial state, using *Insomnia* client



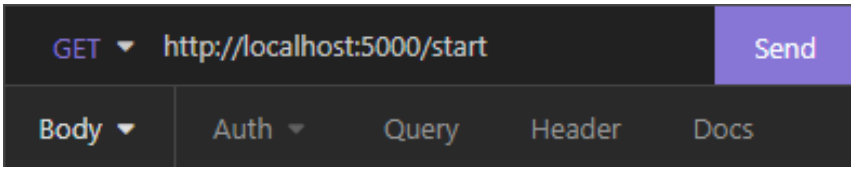
Source: Elaborated by the author.

Figure 35 – Example of the *POST* request for setting *waypoints*, using *Insomnia* client



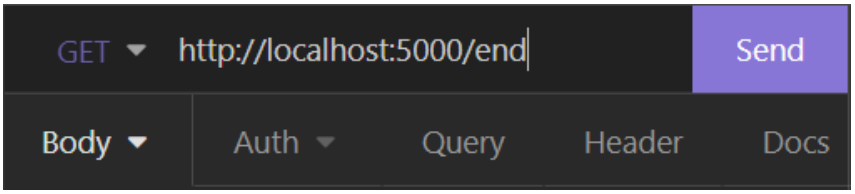
Source: Elaborated by the author.

Figure 36 – Example of the *GET* request for starting the simulation, using *Insomnia* client



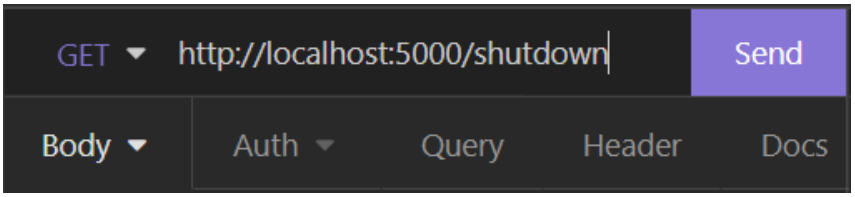
Source: Elaborated by the author.

Figure 37 – Example of the *GET* request for killing only *pydyna* node, using *Insomnia* client



Source: Elaborated by the author.

Figure 38 – Example of the *GET* request for killing all nodes but *backend* node, using *Insomnia* client



Source: Elaborated by the author.

5.5.4 Getting Started

For the setup of the *ROS2* and *pydyna* environments, it's necessary for the user to follow the steps regarded in the *ROS2 Galactic* and *TPN*[13] pages respectively. Then, the user must clone this repository. The first two steps are identical to the ones in 5.2.4.

1. To build packages, run the following command in `main_ws/src` with the x64 Prompt for Visual Studio 2019 terminal as admin:

```
~/tcc-autonomous-ship/src/main_ws/src>colcon build --merge
-install
```

2. On the newly created install directory, run[14]:

```
~/tcc-autonomous-ship/src/main_ws/install>call setup.bat
```

3. Next, run the launch file. This will create *rosbags* in the same directory. Therefore, the recommendation is to run it in `main_ws/install/share/pydyna_simple/db` which is intended to store *rosbags*.

```
~/tcc-autonomous-ship/src/main_ws/install/share/
pydyna_simple/db>ros2 launch path_following path_following.launch.
py
```

4. In order to verify active nodes and topics, the user may run:

```
~/>ros2 topic list -t
```

```
~/>rqt_graph
```

When using *rqt_graph*, a new window opens with the contents of Figure 39. The text within the oval shapes represent the nodes, while the text within the rectangles are the topics.

5. Just like for the *pydyna_simple node*, the user can publish and subscribe to different topics, or call services; to verify if they are working accordingly.

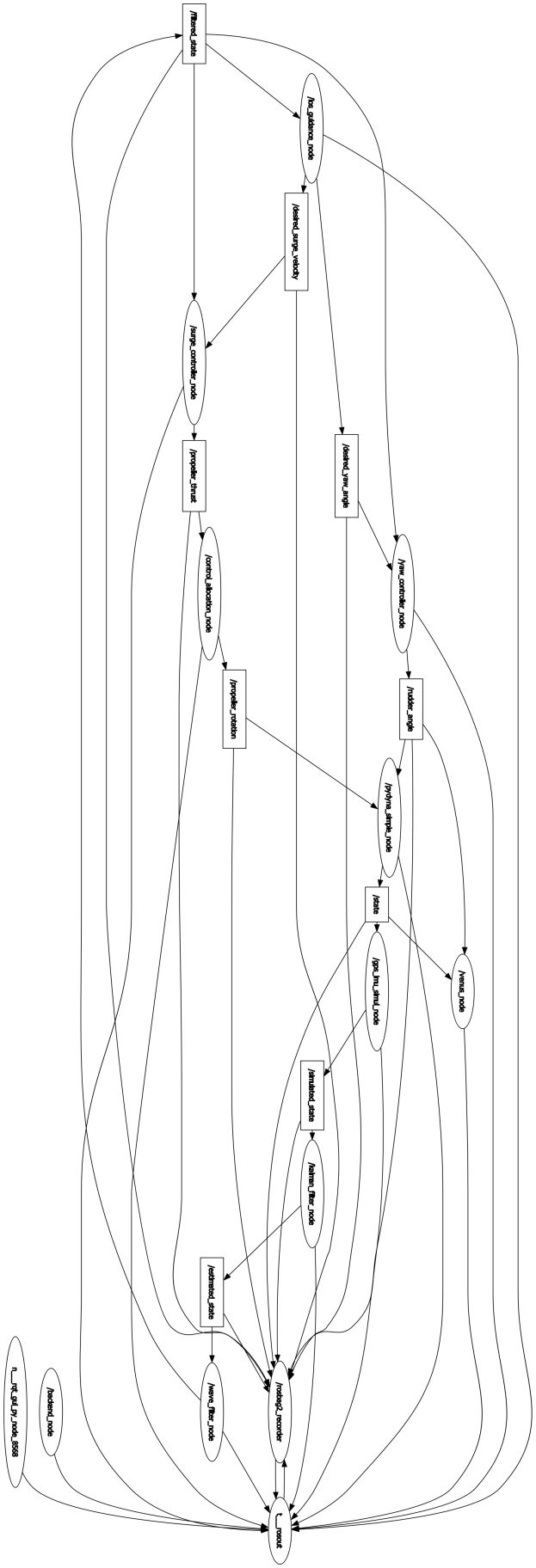
6. With all the nodes working, the user can visualize the vessel through *Venus* on <http://localhost:6150>

[13] TPN page is private

[14] For every new terminal, this command must be run to setup the workspace environment, which sets environment variables that point to your workspace instead of the base environment

7. Once the server is up and running, the user can send HTTP requests, giving the initial conditions and *waypoints*. Illustration of this can be seen in Figure 34 and Figure 35, respectively, using *Insomnia* client. Once this is done, the user can send a request to start the simulation, as illustrated in Figure 36, also using *Insomnia* client.

Figure 39 – Visualization of all active nodes and the topics which they interact with. The direction of the arrows indicates the messages flow. In addition to custom nodes, there are some built-in nodes and topics necessary for *ROS2* and some of it's tools functioning.



Source: Elaborated by the author.

CHAPTER

6

RESULTS AND DISCUSSION

6.1 Results

The results are relative to the case study. These are presented in the format of various *Cases*. Each *Case* tests the path-following task in a different way, by varying some of the conditions. Each *Case* is divided in four sections: *Path-following*, *Conditions*, *Metrics* and *Plots*. The *Path-following* section presents the path and links to the video of the craft performing the path-following task; *Conditions* presents the conditions that define the *Case* and make it reproducible; *Metrics* presents the error metrics obtained for three different runs; *Plots* shows the most significant plots obtained from the first run and links to the other plots.

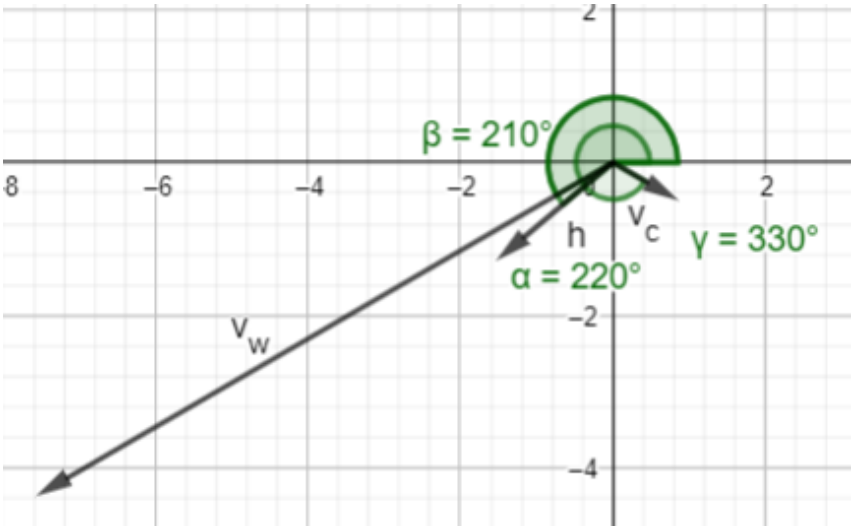
Before the results are presented, it is necessary to describe the environmental condition in which the modules were tuned and the majority of results, cases 1 to 10, were obtained.

6.1.1 Main environmental condition

The *main environmental condition* is defined by three major components, which can be visualized in Figure 40 and are described below:

- **waves**: with $T_w = 12\ s$, $h = 2\ m$ and acting at 225 deg (from east counterclockwise). Where T_w is the wave's period and h is the wave's height.
- **wind**: with $v_w = 8.74\ m/s$ and acting at 210 deg (from east counterclockwise). Where v_w is the wind's velocity.
- **current**: with $v_c = 1\ m/s$ and acting at 330 deg (from east counterclockwise). Where v_c is the current's velocity.

Figure 40 – Vectors that describe the environment conditions. *Errata*: $\alpha = 225$ *deg*



Source: Elaborated by the author.

6.1.2 Case 1

This case portrays the most simple case for the path following algorithm, as the ship only has to follow a straight line.

Path-following

The path to be followed can be seen in Figure 41, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 41 – Case 1: beginning of the path-following task

Source: Elaborated by the author.

Conditions

- Collinear *waypoints*, described in Table 1;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;

  - $\theta = 1.2\ rad$;
  - $u = 2\ m/s$;
  - v = $0\ m/s$;
  - $r = 0\ rad/s$;

- Sensor noise emulation: off (sensor output is the actual state);
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[1];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 1 – Case 1: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	500	3
2	1000	1000	3.5
3	1500	1500	4
4	2000	2000	4.5
5	2500	2500	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 2.

Table 2 – Case 1: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	6.4	27.6	28.7	53.4
2	6.6	27.5	29.2	53.2
3	6.7	27.5	29.1	53.2

Source: Elaborated by the author.

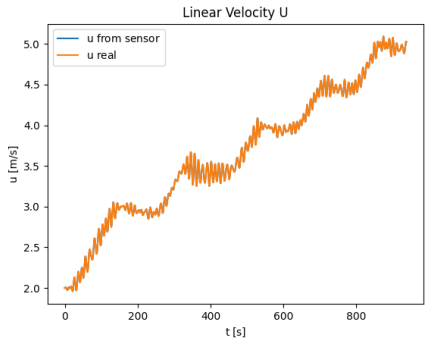
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

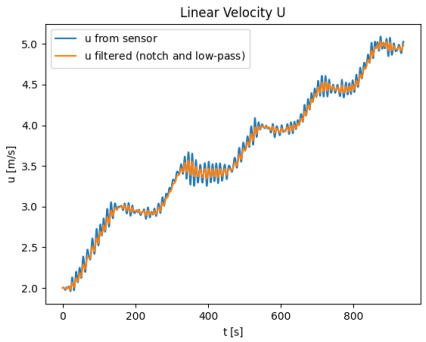
[1] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 42 – Case 1: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



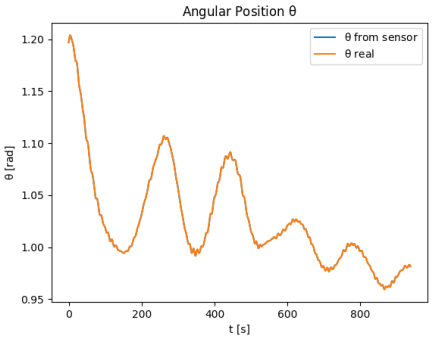
Source: Elaborated by the author.

Figure 43 – Case 1: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



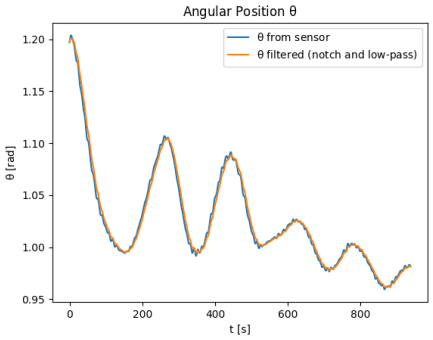
Source: Elaborated by the author.

Figure 44 – Case 1: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



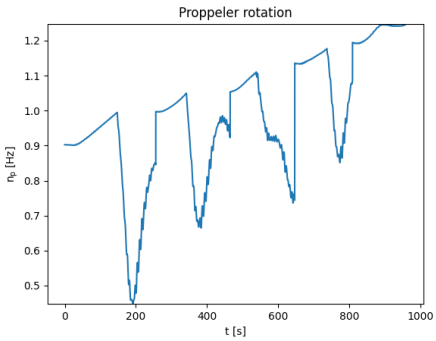
Source: Elaborated by the author.

Figure 45 – Case 1: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



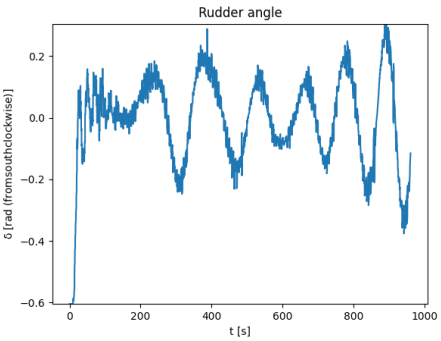
Source: Elaborated by the author.

Figure 46 – Case 1: propeller rotation.



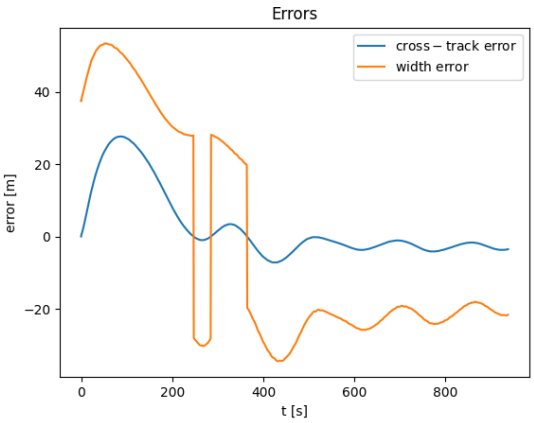
Source: Elaborated by the author.

Figure 47 – Case 1: rudder angle.



Source: Elaborated by the author.

Figure 48 – Case 1: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

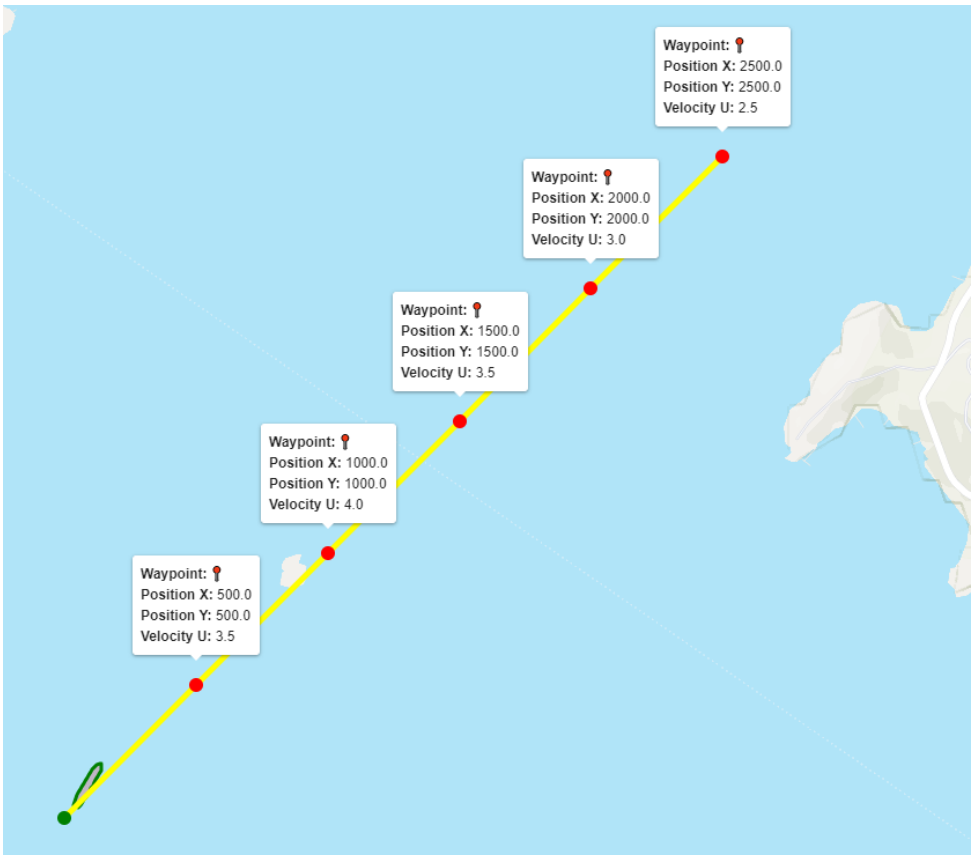
6.1.3 Case 2

This case is just like Case 1, but the desired velocities are different, so that there's a peak of velocity, and then the velocity decreases.

Path-following

The path to be followed can be seen in Figure 49, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 49 – Case 2: beginning of the path-following task



Source: Elaborated by the author.

Conditions

- Collinear *waypoints*, described in Table 3;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = 1.2\ rad$;

  - $u = 2\ m/s$;
  - v = $0\ m/s$;
  - $r = 0\ rad/s$;

- Sensor noise emulation: off (sensor output is the actual state);
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[2];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 3 – Case 2: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	500	3.5
2	1000	1000	4
3	1500	1500	3.5
4	2000	2000	3
5	2500	2500	2.5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 4.

Table 4 – Case 2: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	12.2	38.6	37.8	84.8
2	12.9	44.3	39.0	93.5
3	12.4	39.5	38.4	86.0

Source: Elaborated by the author.

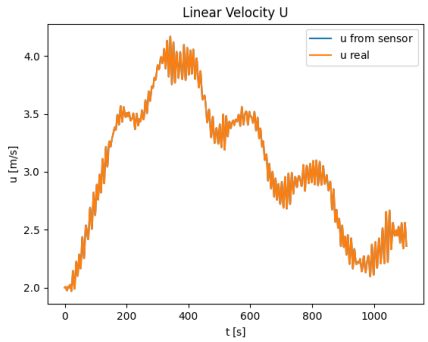
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

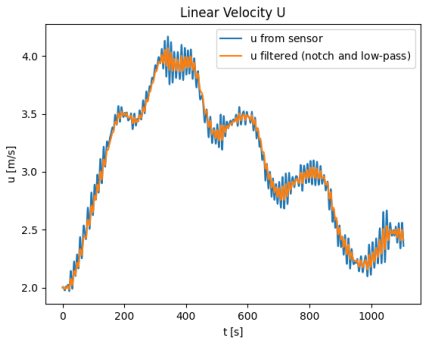
[2] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 50 – Case 2: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



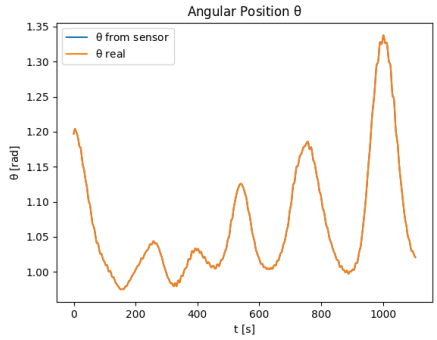
Source: Elaborated by the author.

Figure 51 – Case 2: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



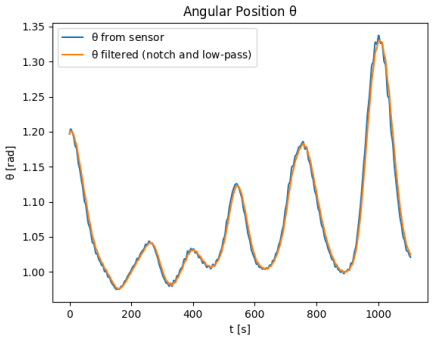
Source: Elaborated by the author.

Figure 52 – Case 2: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



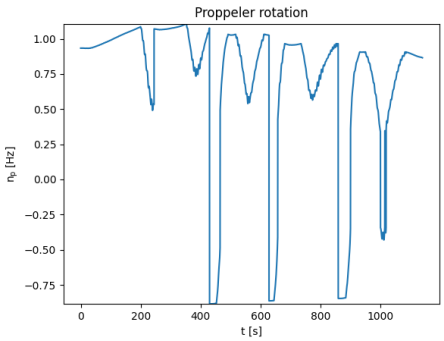
Source: Elaborated by the author.

Figure 53 – Case 2: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



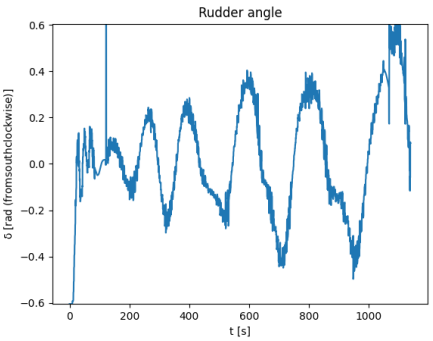
Source: Elaborated by the author.

Figure 54 – Case 2: propeller rotation.



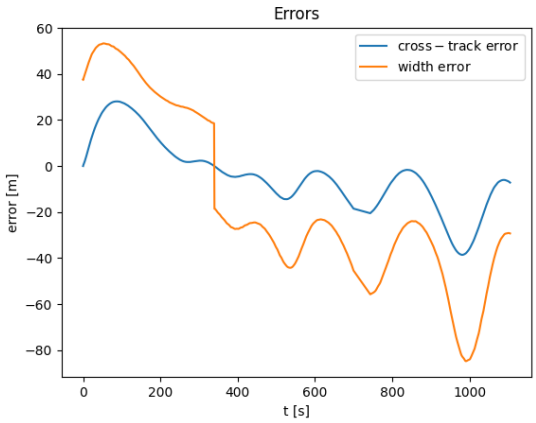
Source: Elaborated by the author.

Figure 55 – Case 2: rudder angle.



Source: Elaborated by the author.

Figure 56 – Case 2: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

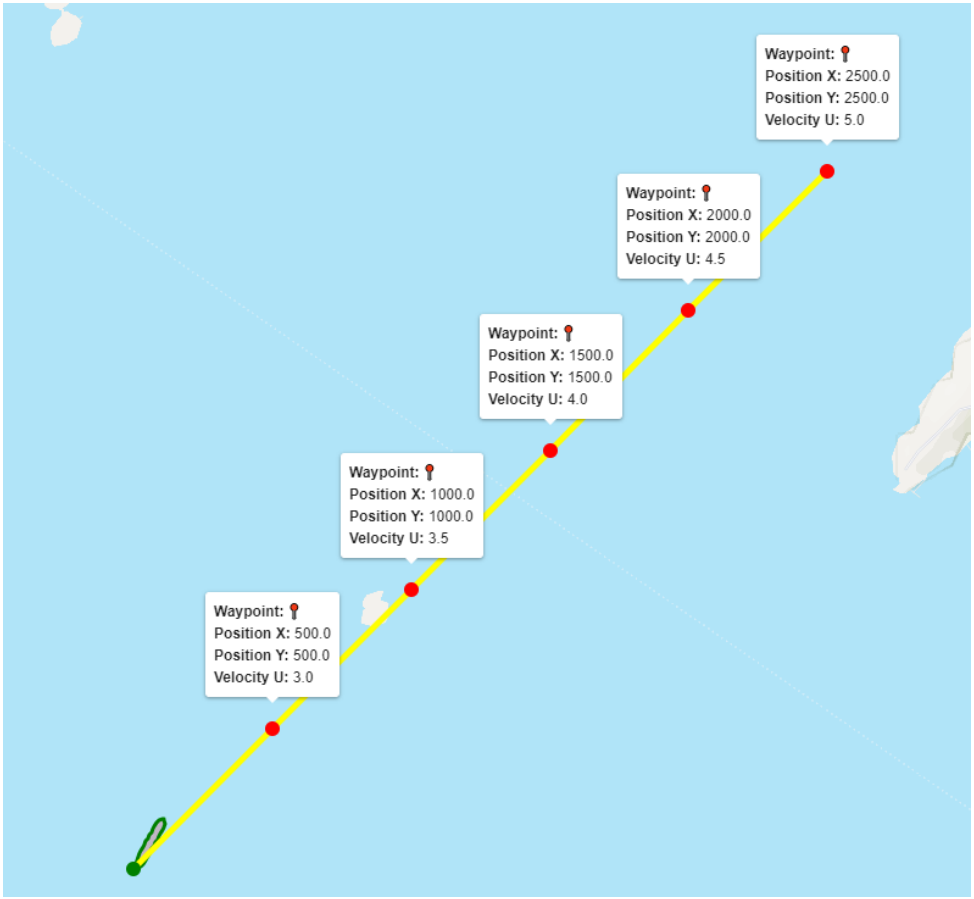
6.1.4 Case 3

This case is just like Case 1, but with the sensor noise emulation on.

Path-following

The path to be followed can be seen in Figure 57, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 57 – Case 3: beginning of the path-following task



Source: Elaborated by the author.

Conditions

- Collinear *waypoints*, described in Table 5;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = 1.2\ rad$;

  - $u = 2\ m/s$;
  - v = $0\ m/s$;
  - $r = 0\ rad/s$;

- Sensor noise emulation: on;
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[3];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 5 – Case 3: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	500	3
2	1000	1000	3.5
3	1500	1500	4
4	2000	2000	4.5
5	2500	2500	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 6.

Table 6 – Case 3: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	9.7	27.7	32.9	63.8
2	10.0	26.5	33.8	64.8
3	11.6	39.9	36.6	89.2

Source: Elaborated by the author.

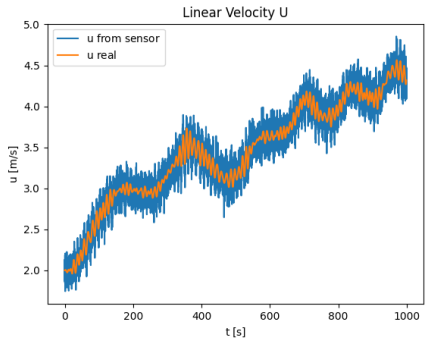
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

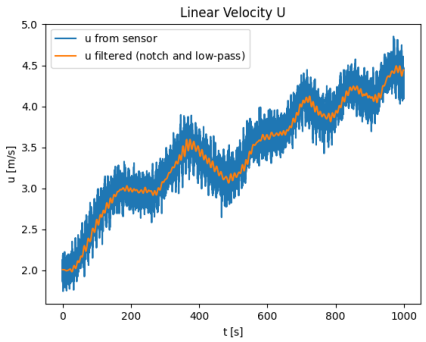
[3] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 58 – Case 3: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



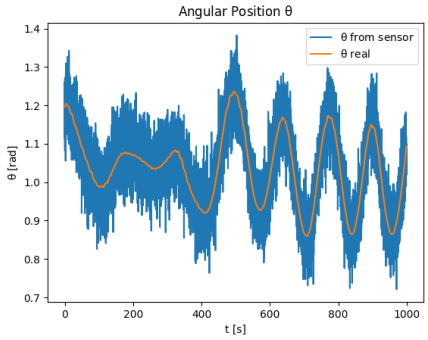
Source: Elaborated by the author.

Figure 59 – Case 3: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



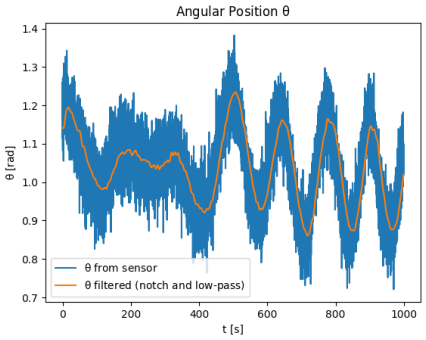
Source: Elaborated by the author.

Figure 60 – Case 3: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



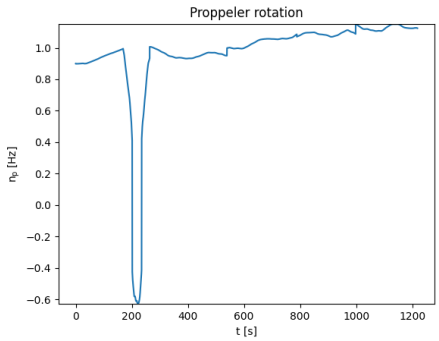
Source: Elaborated by the author.

Figure 61 – Case 3: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



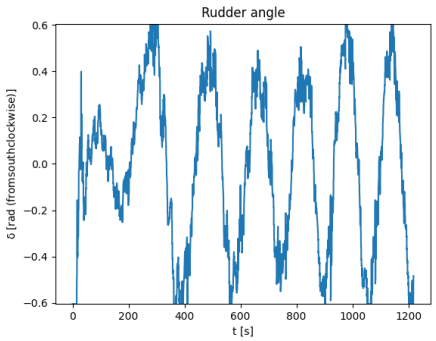
Source: Elaborated by the author.

Figure 62 – Case 3: propeller rotation.



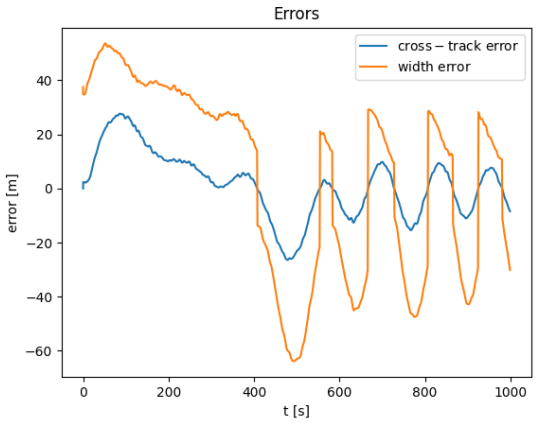
Source: Elaborated by the author.

Figure 63 – Case 3: rudder angle.



Source: Elaborated by the author.

Figure 64 – Case 3: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

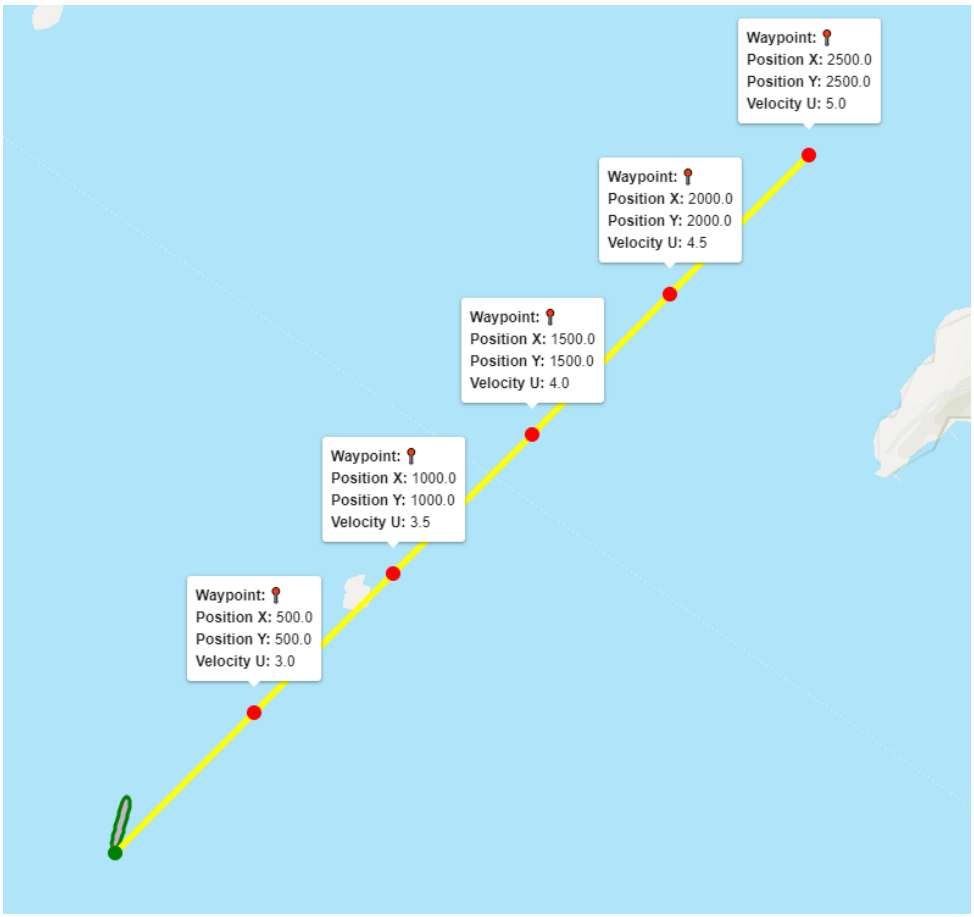
6.1.5 Case 4

This case is just like Case 3, but with $\theta = \frac{\pi}{2}\ rad$ in the initial state.

Path-following

The path to be followed can be seen in Figure 65, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 65 – Case 4: beginning of the path-following task



Source: Elaborated by the author.

Conditions

- Collinear *waypoints*, described in Table 7;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = \frac{\pi}{2}\ rad$;

  - $u = 2\ m/s$;
  - v = $0\ m/s$;
  - $r = 0\ rad/s$;

- Sensor noise emulation: on;
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[4];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 7 – Case 4: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	500	3
2	1000	1000	3.5
3	1500	1500	4
4	2000	2000	4.5
5	2500	2500	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 8.

Table 8 – Case 4: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	20.6	86.6	45.2	112.8
2	20.8	87.7	44.0	112.4
3	21.4	87.7	45.7	112.5

Source: Elaborated by the author.

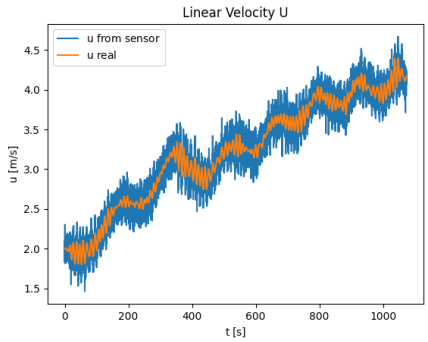
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

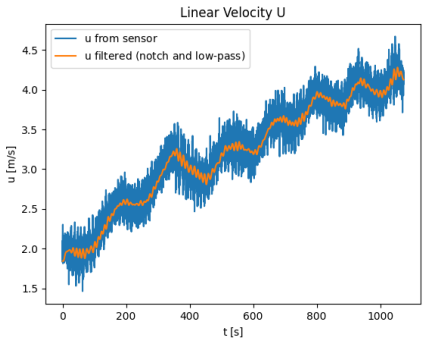
[4] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 66 – Case 4: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



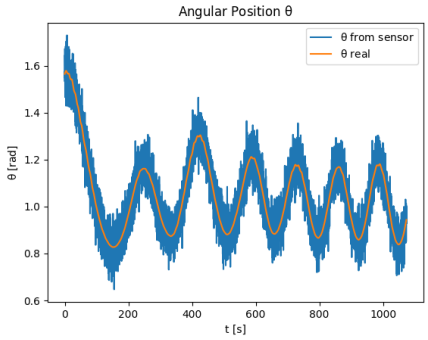
Source: Elaborated by the author.

Figure 67 – Case 4: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



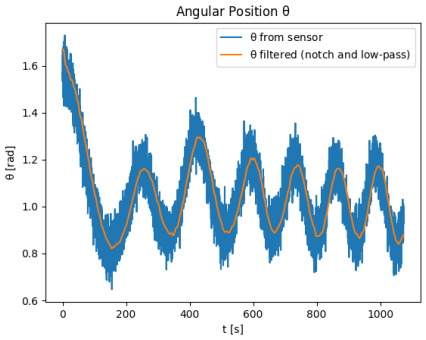
Source: Elaborated by the author.

Figure 68 – Case 4: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



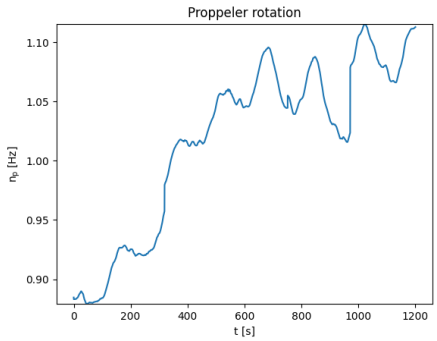
Source: Elaborated by the author.

Figure 69 – Case 4: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



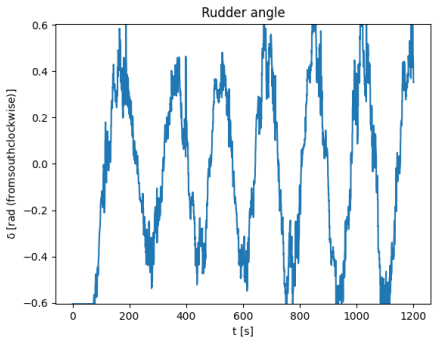
Source: Elaborated by the author.

Figure 70 – Case 4: propeller rotation.



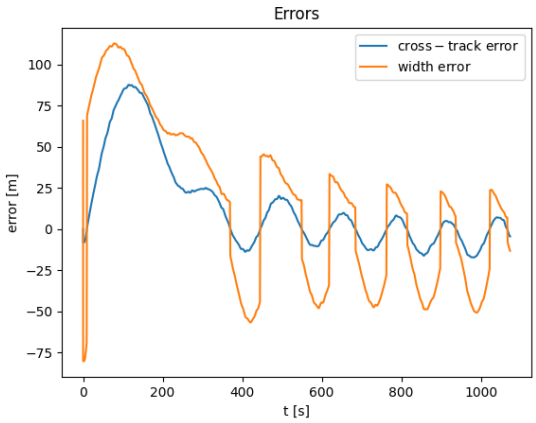
Source: Elaborated by the author.

Figure 71 – Case 4: rudder angle.



Source: Elaborated by the author.

Figure 72 – Case 4: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

6.1.6 Case 5

This case is just like Case 3, but with the starting surge velocity at $u = 1\ m/s$ in the initial state.

Path-following

The path to be followed can be seen in Figure 73, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 73 – Case 5: beginning of the path-following task

Source: Elaborated by the author.

Conditions

- Collinear *waypoints*, described in Table 9;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = 1.2\ rad$;

  - $u = 1\ m/s$;
  - v = $0\ m/s$;
  - $r = 0\ rad/s$;

- Sensor noise emulation: on;
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[5];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 9 – Case 5: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	500	3
2	1000	1000	3.5
3	1500	1500	4
4	2000	2000	4.5
5	2500	2500	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 10.

Table 10 – Case 5: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	10.4	27.3	36.9	72.0
2	9.7	26.5	36.3	59.3
3	9.9	29.8	36.0	62.9

Source: Elaborated by the author.

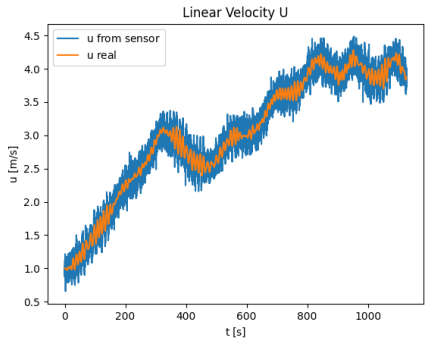
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

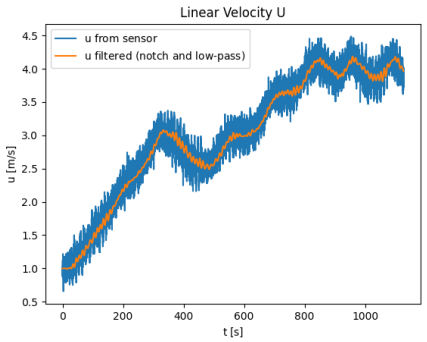
[5] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 74 – Case 5: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



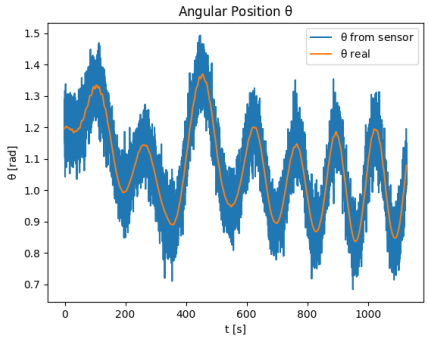
Source: Elaborated by the author.

Figure 75 – Case 5: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



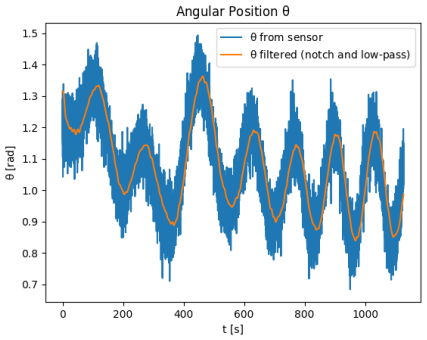
Source: Elaborated by the author.

Figure 76 – Case 5: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



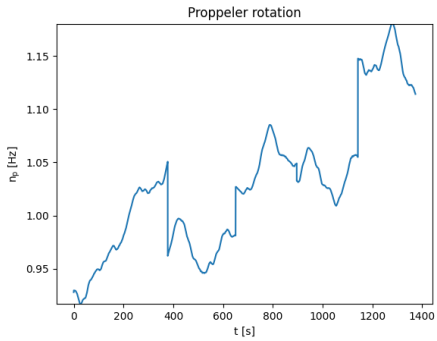
Source: Elaborated by the author.

Figure 77 – Case 5: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



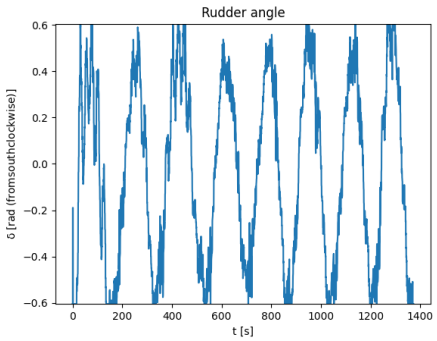
Source: Elaborated by the author.

Figure 78 – Case 5: propeller rotation.



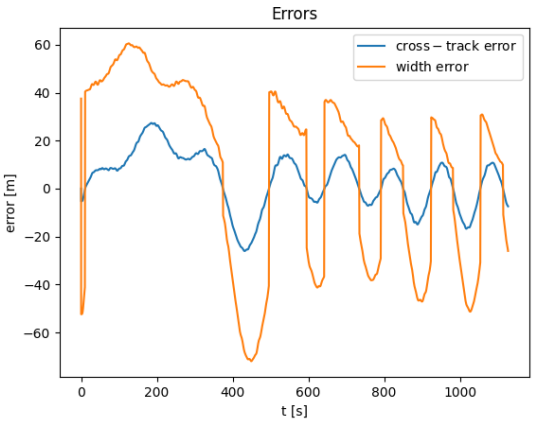
Source: Elaborated by the author.

Figure 79 – Case 5: rudder angle.



Source: Elaborated by the author.

Figure 80 – Case 5: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

6.1.7 Case 6

Just like Case 3, but instead of a collinear path, with a slight zigzag.

Path-following

The path to be followed can be seen in Figure 81, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 81 – Case 6: beginning of the path-following task

Source: Elaborated by the author.

Conditions

- Zigzag *waypoints*, described in Table 11;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = 1.2\ rad$;

  - $u = 2\ m/s$;
  - v = $0\ m/s$;
  - $r = 0\ rad/s$;

- Sensor noise emulation: on;
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[6];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 11 – Case 6: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	600	3
2	1000	900	3.5
3	1500	1600	4
4	2000	1900	4.5
5	2500	2600	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 12.

Table 12 – Case 6: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	38.4	118.6	72.9	179.1
2	40.9	151.4	75.7	218.5
3	43.1	169.5	77.4	234.7

Source: Elaborated by the author.

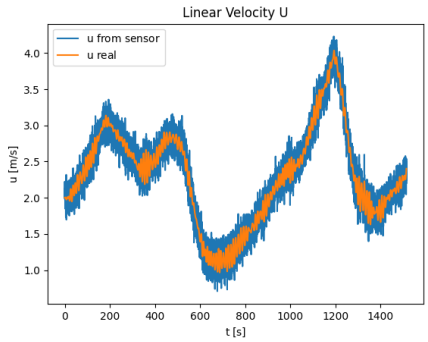
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

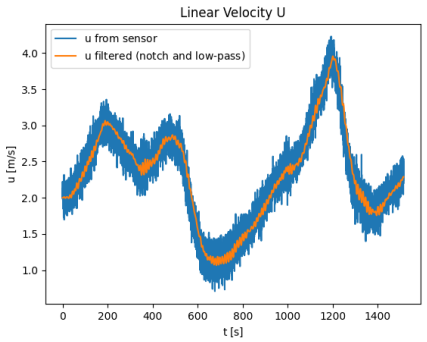
[6] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 82 – Case 6: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



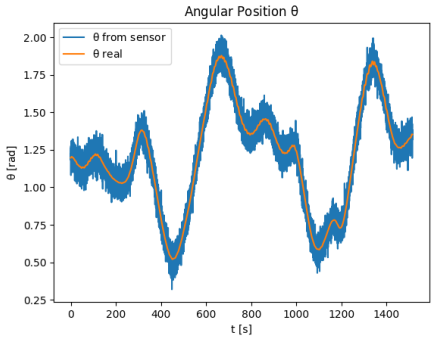
Source: Elaborated by the author.

Figure 83 – Case 6: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



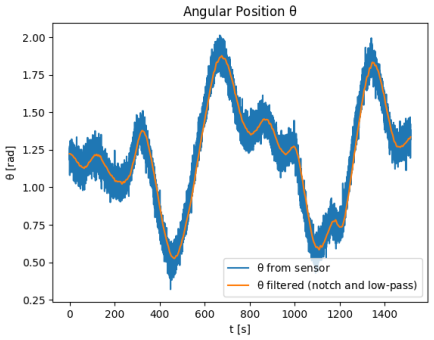
Source: Elaborated by the author.

Figure 84 – Case 6: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



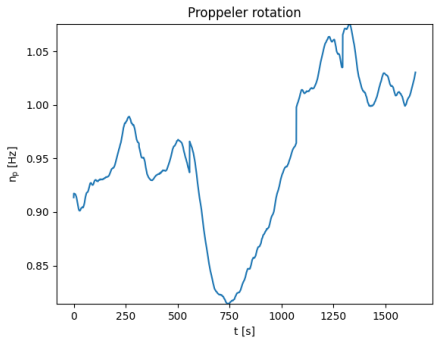
Source: Elaborated by the author.

Figure 85 – Case 6: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



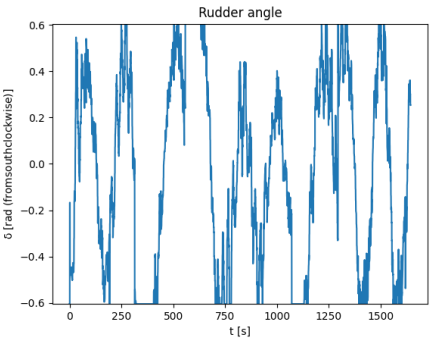
Source: Elaborated by the author.

Figure 86 – Case 6: propeller rotation.



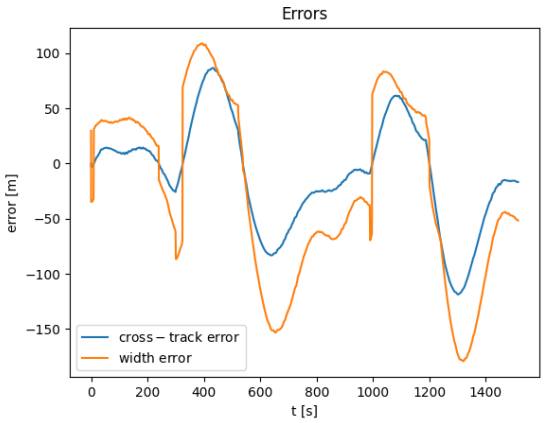
Source: Elaborated by the author.

Figure 87 – Case 6: rudder angle.



Source: Elaborated by the author.

Figure 88 – Case 6: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

6.1.8 Case 7

This case is similar to Case 6, but instead of $R_w = 50\,m$, its with $R_w = R_{los} = 2L_{pp}$.

Path-following

The path to be followed can be seen in Figure 89, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 89 – Case 7: beginning of the path-following task

Source: Elaborated by the author.

Conditions

- Zigzag *waypoints*, described in Table 13;
- Initial State:
  - $x = 0\,m$;
  - $y = 0\,m$;
  - $\theta = 1.2\,rad$;
  - $u = 2\,m/s$;

  - v = 0 m/s;
  - $r = 0\, rad/s$;
- Sensor noise emulation: on;
- $R_w = R_{los} = 2L_{pp}$;
- $R_{wFinal} = 50\, m$;
- Ship initial coordinates: (-23.06255, -44.2772)[7];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 13 – Case 7: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	600	3
2	1000	900	3.5
3	1500	1600	4
4	2000	1900	4.5
5	2500	2600	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 14.

Table 14 – Case 7: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	14.8	35.6	39.3	87.3
2	15.3	34.6	40.6	95.5
3	16.3	47.2	43.2	98.7

Source: Elaborated by the author.

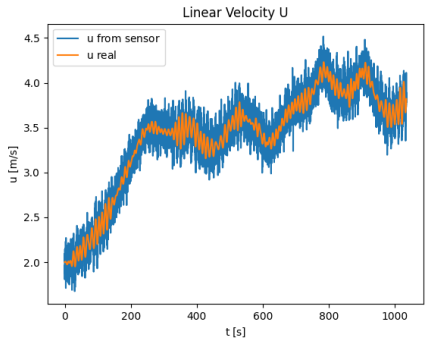
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

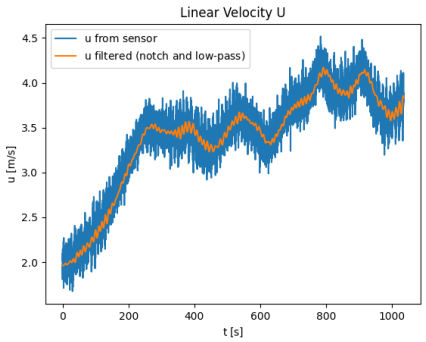
[7] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 90 – Case 7: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



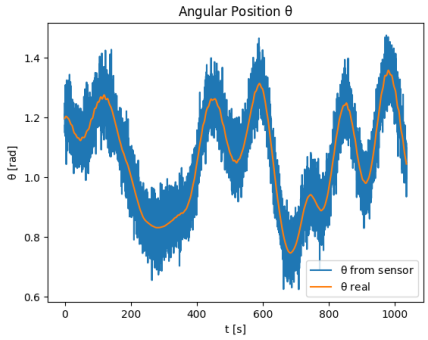
Source: Elaborated by the author.

Figure 91 – Case 7: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



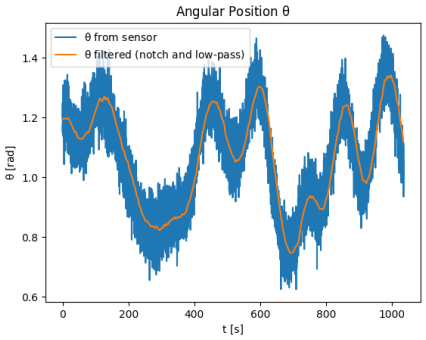
Source: Elaborated by the author.

Figure 92 – Case 7: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



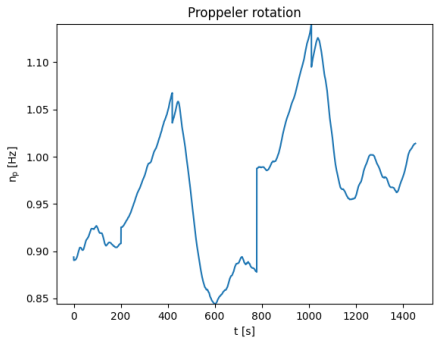
Source: Elaborated by the author.

Figure 93 – Case 7: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



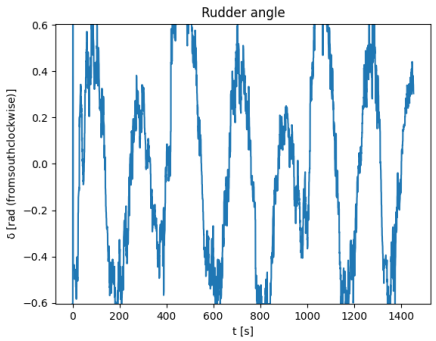
Source: Elaborated by the author.

Figure 94 – Case 7: propeller rotation.



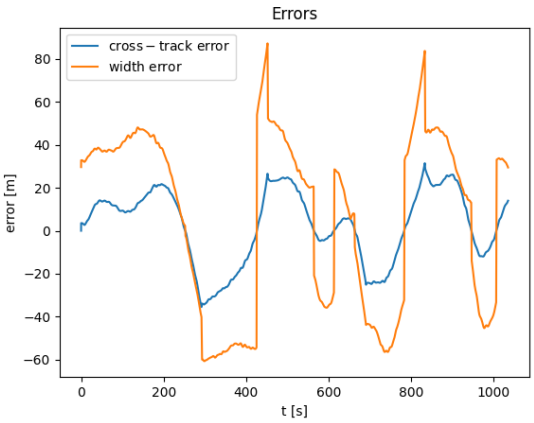
Source: Elaborated by the author.

Figure 95 – Case 7: rudder angle.



Source: Elaborated by the author.

Figure 96 – Case 7: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

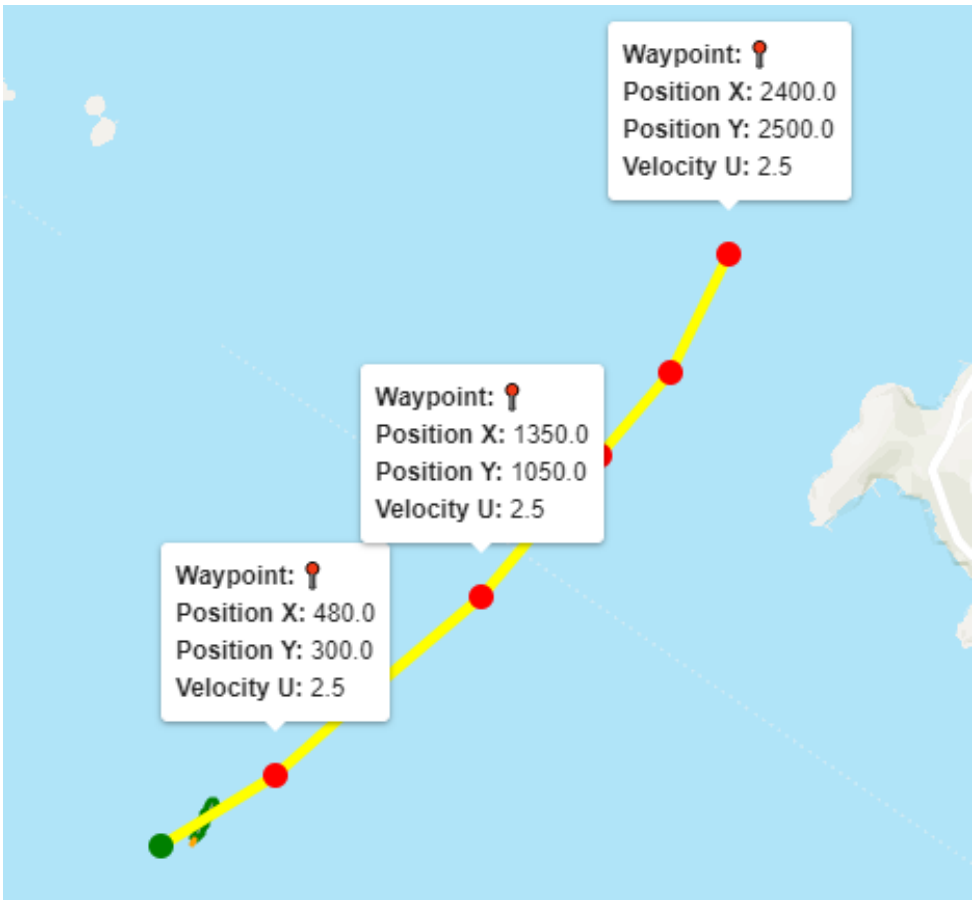
6.1.9 Case 8

This case is a path in which the vessel makes a slight counterclockwise turn, with the initial yaw angle at $\theta = 1\ rad$ in the initial state, and increasing velocity similar to previous cases.

Path-following

The path to be followed can be seen in Figure 97, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 97 – Case 8: beginning of the path-following task



Source: Elaborated by the author.

Conditions

- Curve *waypoints*, described in Table 15;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = 1\ rad$;

  - $u = 2\ m/s$;
  - v = 0 m/s;
  - $r = 0\ rad/s$;

- Sensor noise emulation: on;
- $R_w = 50\ m$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[8];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 15 – Case 8: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	480	300	2.5
2	880	650	3.0
3	1350	1050	3.5
4	1850	1650	4
5	2150	2000	4.5
6	2400	2500	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 16.

Table 16 – Case 8: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	42.7	137.5	83.2	221.8
2	15.5	43.2	44.6	100.2
3	18.3	68.5	55.2	138.5

Source: Elaborated by the author.

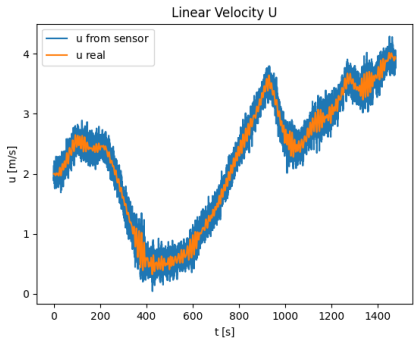
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

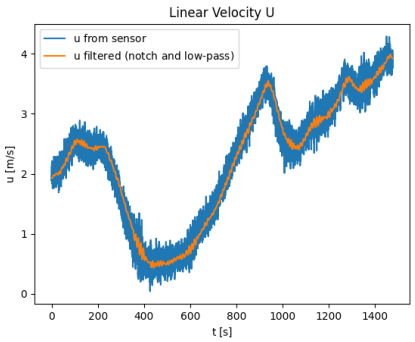
[8] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 98 – Case 8: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



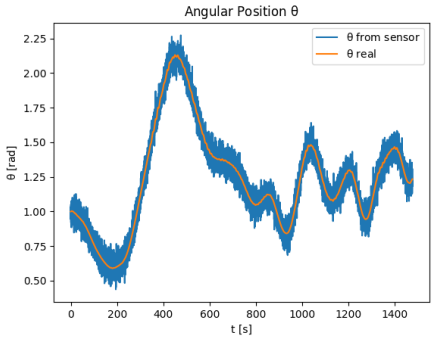
Source: Elaborated by the author.

Figure 99 – Case 8: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



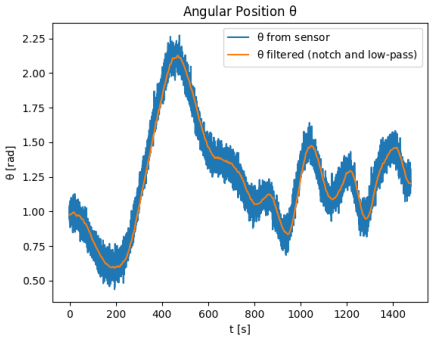
Source: Elaborated by the author.

Figure 100 – Case 8: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



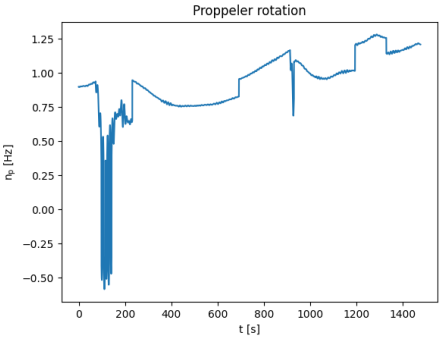
Source: Elaborated by the author.

Figure 101 – Case 8: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



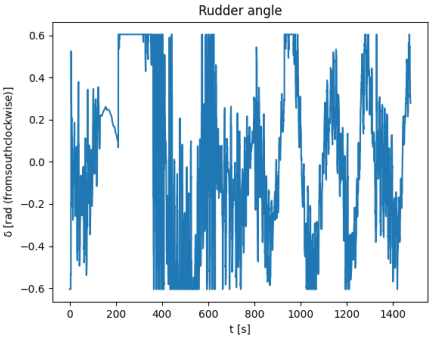
Source: Elaborated by the author.

Figure 102 – Case 8: propeller rotation.



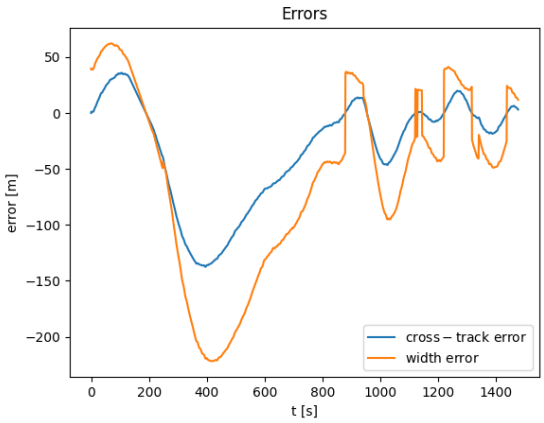
Source: Elaborated by the author.

Figure 103 – Case 8: rudder angle.



Source: Elaborated by the author.

Figure 104 – Case 8: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

6.1.10 Case 9

This case is similar to Case 8, but instead of $R_w = 50\,m$, with $R_w = R_{los} = 2L_{pp}$.

Path-following

The path to be followed can be seen in Figure 105, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 105 – Case 9: beginning of the path-following task

Source: Elaborated by the author.

Conditions

- Curve *waypoints*, described in Table 17;
- Initial State:
  - $x = 0\,m$;
  - $y = 0\,m$;
  - $\theta = 1\,rad$;
  - $u = 2\,m/s$;

  - v = 0 m/s;
  - $r = 0\ rad/s$;

- Sensor noise emulation: on;
- $R_w = R_{los} = 2L_{pp}$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[9];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: yes.

Table 17 – Case 9: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	480	300	2.5
2	880	650	3.0
3	1350	1050	3.5
4	1850	1650	4.0
5	2150	2000	4.5
6	2400	2500	5.0

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 18.

Table 18 – Case 9: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	14.2	32.0	42.0	85.4
2	12.2	36.4	39.5	77.1
3	14.1	32.4	41.5	88.1

Source: Elaborated by the author.

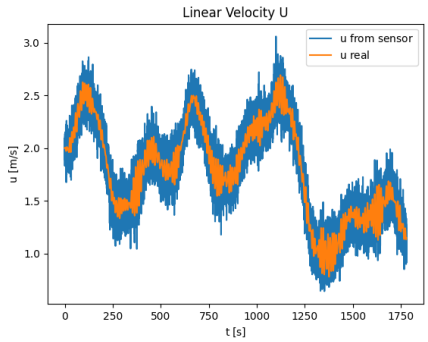
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

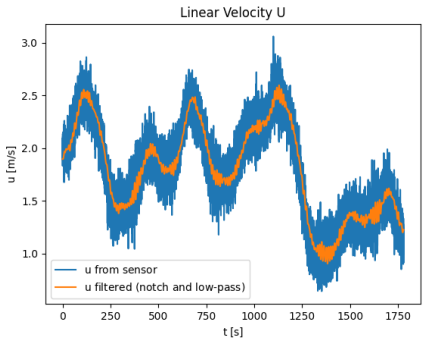
[9] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Figure 106 – Case 9: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



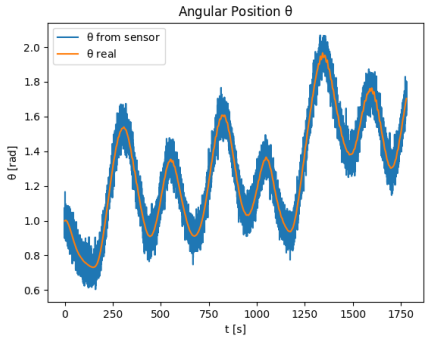
Source: Elaborated by the author.

Figure 107 – Case 9: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



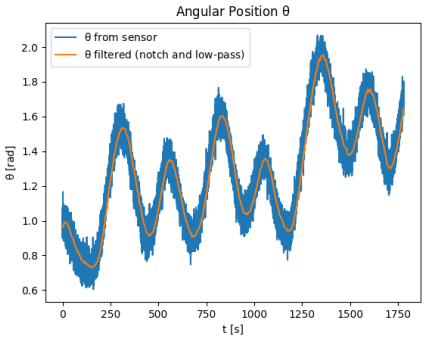
Source: Elaborated by the author.

Figure 108 – Case 9: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



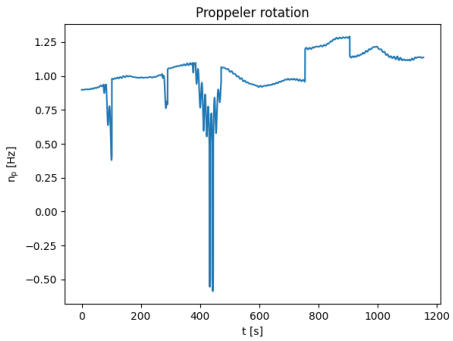
Source: Elaborated by the author.

Figure 109 – Case 9: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



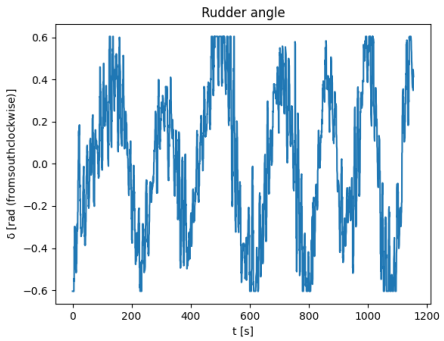
Source: Elaborated by the author.

Figure 110 – Case 9: propeller rotation.



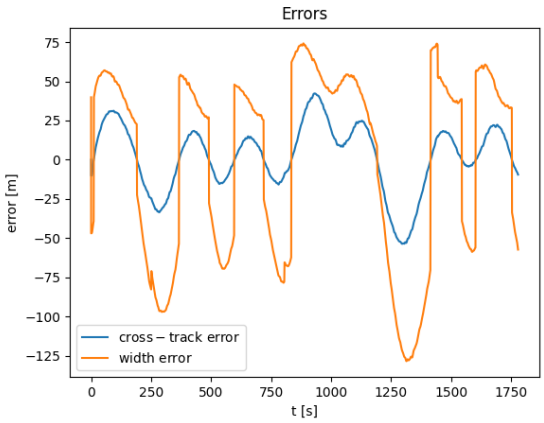
Source: Elaborated by the author.

Figure 111 – Case 9: rudder angle.



Source: Elaborated by the author.

Figure 112 – Case 9: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

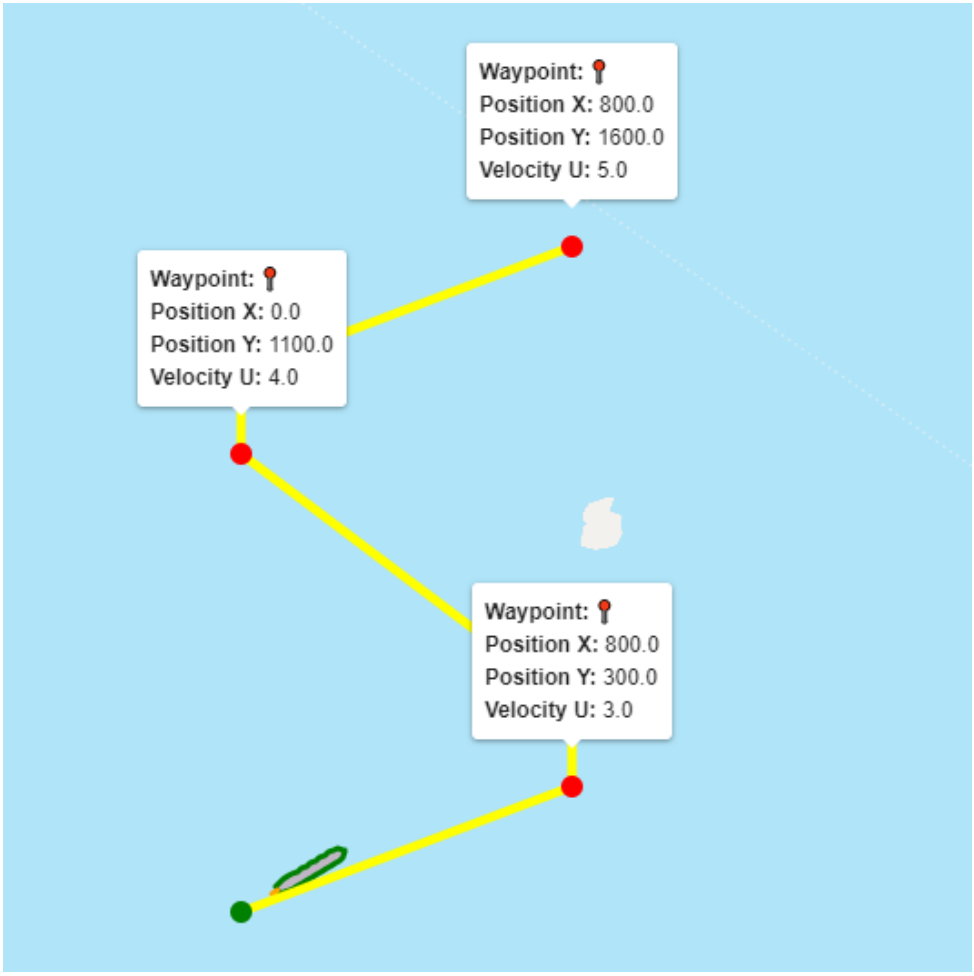
6.1.11 Case 10

Similar to Case 7, but instead of zizag path, an “S” shaped path with strong direction change, and with $\theta = 0.8\ rad$ in the initial state.

Path-following

The path to be followed can be seen in Figure 113, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 113 – Case 10: beginning of the path-following task



Source: Elaborated by the author.

Conditions

- "S" *waypoints*, described in Table 19;
- Initial State:
  - $x = 0\ m$;

  - $y = 0\,m$;
  - $\theta = 0.8\,rad$;
  - $u = 2\,m/s$;
  - v = $0\,m/s$;
  - $r = 0\,rad/s$;

- Sensor noise emulation: on;
- $R_w = R_{los} = 2L_{pp}$;
- $R_{wFinal} = 50\,m$;
- Ship initial coordinates: (-23.06255, -44.2772)[10];
- Environmental condition: *main environmental condition*;
- Reached final *waypoint*: no.

Table 19 – Case 10: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	800	300	3.0
2	800	500	3.5
3	0	1100	4.0
4	0	1300	4.5
5	800	1600	5.0

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 20.

Table 20 – Case 10: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	58.2	234.9	113.6	276.9
2	56.6	261.2	111.7	336.9
3	42.6	256.3	97.6	329.0

Source: Elaborated by the author.

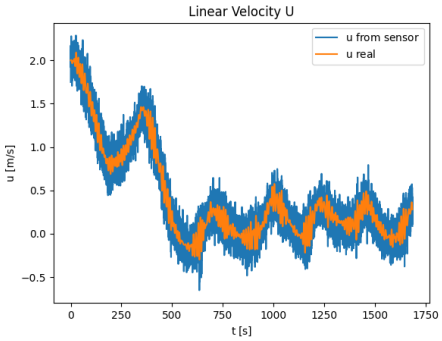
Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

[10] Location is: “Angra dos Reis, Rio de Janeiro, Brazil.”

Plots

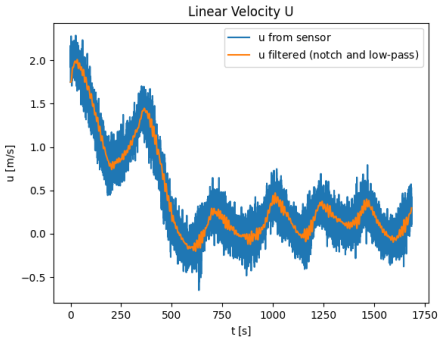
The most relevant plots are presented below. All the generated plots can be found here.

Figure 114 – Case 10: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



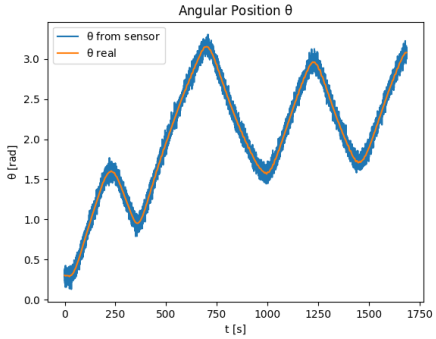
Source: Elaborated by the author.

Figure 115 – Case 10: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



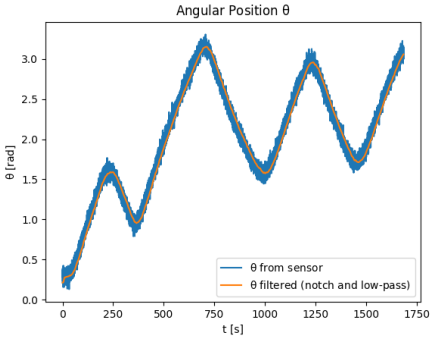
Source: Elaborated by the author.

Figure 116 – Case 10: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



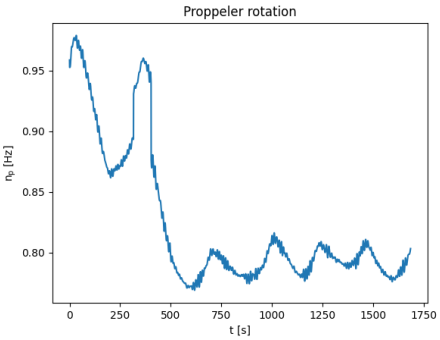
Source: Elaborated by the author.

Figure 117 – Case 10: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



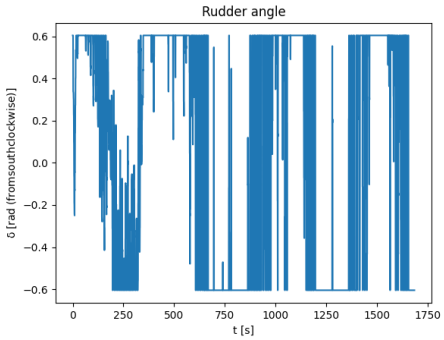
Source: Elaborated by the author.

Figure 118 – Case 10: propeller rotation.



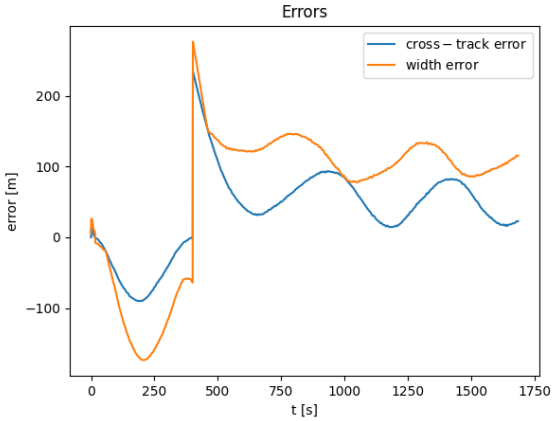
Source: Elaborated by the author.

Figure 119 – Case 10: rudder angle.



Source: Elaborated by the author.

Figure 120 – Case 10: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

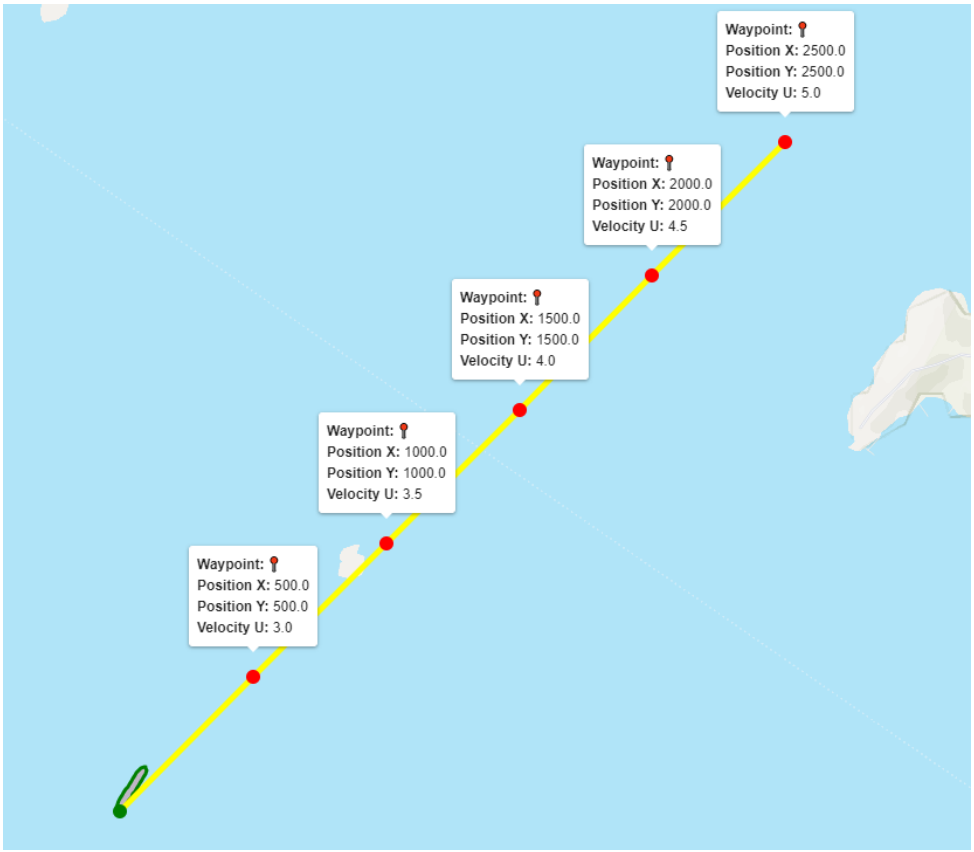
6.1.12 Case 11

This case is just like Case 3, but with the environmental conditions altered. The current's angle is now of 280 deg, the winds are at 10 m/s and the waves are 2.5 m tall.

Path-following

The path to be followed can be seen in Figure 121, which was captured at the beginning of the path-following task. The full video of the task can be found here.

Figure 121 – Case 11: beginning of the path-following task



Source: Elaborated by the author.

Conditions

- Linear *waypoints*, described in Table 21;
- Initial State:
  - $x = 0\ m$;
  - $y = 0\ m$;
  - $\theta = 1.2\ rad$;

  - $u = 2\ m/s$;
  - v = 0 m/s;
  - $r = 0\ rad/s$;
- Sensor noise emulation: on;
- $R_w = R_{los} = 2L_{pp}$;
- $R_{wFinal} = 50\ m$;
- Ship initial coordinates: (-23.06255, -44.2772)[11];
- Environmental condition:
  - waves: with $T_w = 12\ s$, $h = 2.5\ m$ and acting at 225 deg (from east counterclockwise). Where T_w is the wave's period and h is the wave's height;
  - wind: with $v_w = 10\ m/s$ and acting at 210 deg (from east counterclockwise). Where v_w is the wind's velocity;
  - current: with $v_c = 1\ m/s$ and acting at 280 deg (from east counterclockwise). Where v_c is the current's velocity;
- Reached final *waypoint*: no.

Table 21 – Case 11: *waypoints*

Waypoints	Position x $[m]$	Position y $[m]$)	Velocity u $[m/s]$
1	500	500	3
2	1000	1000	3.5
3	1500	1500	4
4	2000	2000	4.5
5	2500	2500	5

Source: Elaborated by the author.

Metrics

The metrics can be seen in Table 22.

Where: cte_m is the mean *cross-track* error; cte_{max} is the max *cross-track* error; we_m is the mean *width* error; we_{max} is the max *width* error.

Plots

The most relevant plots are presented below. All the generated plots can be found here.

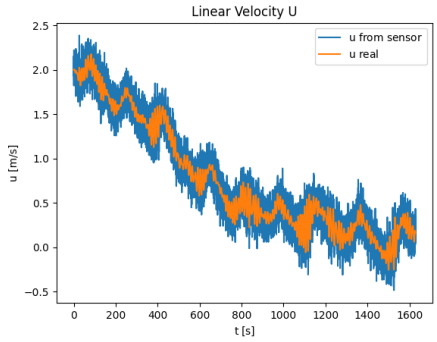
[11] Location is: "Angra dos Reis, Rio de Janeiro, Brazil."

Table 22 – Case 11: metrics

Run	cte_m $[m]$	cte_{max} $[m]$	we_m $[m]$	we_{max} $[m]$
1	37.3	116.7	81.8	157.4
2	25.8	68.8	66.8	117.6
3	21.3	73.5	61.0	138.2

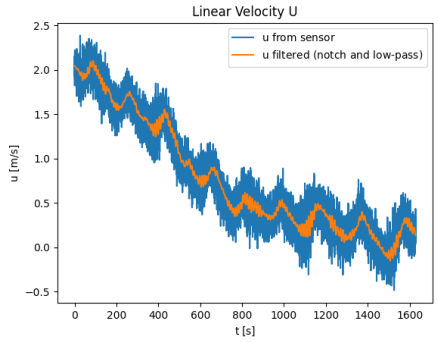
Source: Elaborated by the author.

Figure 122 – Case 11: real and sensor surge velocity. Real surge velocity is in orange and sensor surge velocity is in blue.



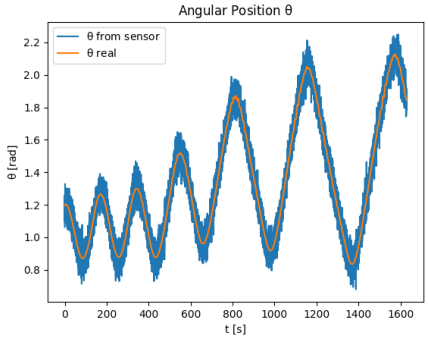
Source: Elaborated by the author.

Figure 123 – Case 11: filtered and sensor surge velocity. Filtered surge velocity is in orange and sensor surge velocity is in blue.



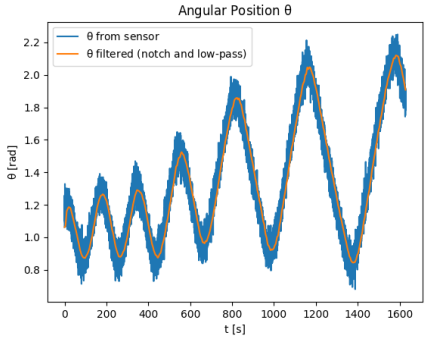
Source: Elaborated by the author.

Figure 124 – Case 11: real and sensor yaw angle. Real yaw angle is in orange and sensor yaw angle is in blue.



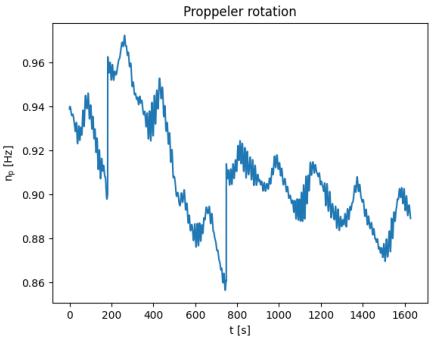
Source: Elaborated by the author.

Figure 125 – Case 11: filtered and sensor yaw angle. Filtered yaw angle is in orange and sensor yaw angle is in blue.



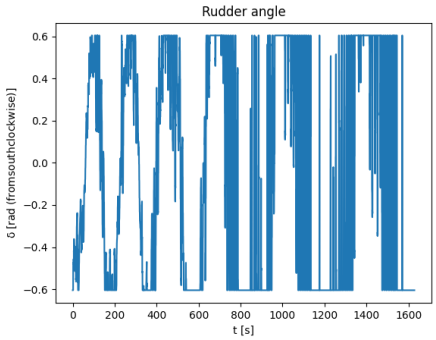
Source: Elaborated by the author.

Figure 126 – Case 11: propeller rotation.



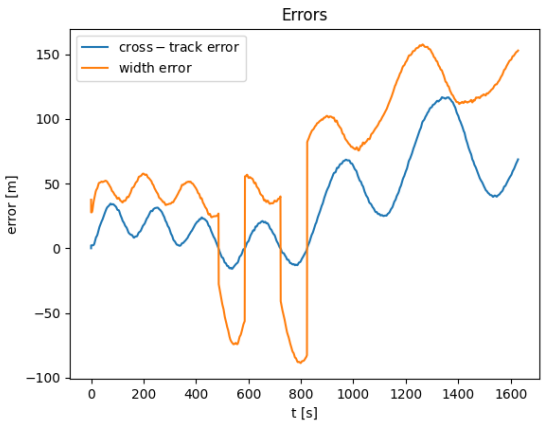
Source: Elaborated by the author.

Figure 127 – Case 11: rudder angle.



Source: Elaborated by the author.

Figure 128 – Case 11: *cross-track* and *width* errors. *Cross-track* error is in blue and *width* error is in orange.



Source: Elaborated by the author.

6.2 Discussion

The discussion is done, separately, for different signals extracted from the results. The topics, in order, are the following: path-following ability; ability to reach final *waypoint*; velocity-following ability; actuators behaviour; radius of acceptance influence; initial condition influence; environmental condition influence and sensor noise emulation influence.

6.2.1 Path-following ability

To see if the cases were successful at the path-following task, the first thing to check is if the craft reached the final *waypoint*. If it did not reach it, the case was not successful. If it reached it, attention is turned to the mean cross-track error cte_m. Taking the mean of cte_m, over all three runs, it is possible to state that a *Case* was successful. This can be done by using the evaluation metric exposed in Equation 5.2. Therefore, success is defined by:

$$\frac{\sum_{run=1}^{3} cte_m}{3} < 16.1$$

With this in mind, Case 1, Case 2, Case 3, Case 5, Case 7, and Case 9 were successful; whereas Case 4, Case 6, Case 8, Case 10, Case 11 were not. However Case 6 and Case 8 only vary by a controlled parameter from Case 7 and Case 9 respectively. Therefore, the conditions which the craft is not able to perform well the path-following task are the ones described in Case 4, Case 10 and Case 11.

The situations where the craft is successful present themselves as smooth paths, where the craft's initial yaw angle is close to the angle of the path line and the craft is under the *main environmental condition*, in which the modules were tuned. The situations where the craft under performs, are when some of these factors do not hold. In Case 4 initial yaw angle is not favourable; in Case 10 the path is not smooth; and in Case 11 the craft is subjected to a different environmental condition.

It is also important to notice that in all cases, the angular position does not stabilize very well, it keeps oscillating. This behaviour is not ideal, but did not affect too much the ability to retain a low error. When the *PID* yaw controller was tuned, this behaviour was identified. Efforts were made to try to make the craft stabilize, by increasing the integral term and decreasing the gain. However, this affected other dynamics and increased the path-following error. Therefore, it was chosen to remain with this oscillating behaviour. Improvements should be done to this oscillating response, but require a careful look at the methods chosen and implications on the overall task. One important factor is that, by not using a Kalman Filter type of filter, the error in the state estimation remained significant throughout the whole path, instead of converging to a low steady-state error in some sense.

6.2.2 Ability to reach final waypoint

In all cases, but Case 10 and Case 11, the craft reached the final *waypoint* (in a reasonable amount of time, approximately 5 minutes). These situations where the craft got stuck in the middle of the way, may be explained by the conditions of these situations, together with the simplifying assumptions adopted for the models and by the type of surge controller used.

In cases Case 10 and Case 11 the craft is subjected to harsh direction changes and new external forces, respectively. These factors can cause the craft to become less stable, oscillate around the path and rotate around it's axis. Specifically, Case 11 also increases the amount of latitudinal force the craft is being subjected to, increasing the sway velocity. The simplified models used in this monograph adopted the assumption that the sway linear velocity v and yaw angular velocity r were small. This gap between the simulated scenarios and the assumptions can be related to the poor performance of the craft. Finally, the controller used is not very dynamic, it is relatively constant between two *waypoints*. This means that, for example, if some novel effect occurs to the craft that makes it lose velocity along the way, the controller doesn't compensate for it by increasing the gain. The maximum gain of the controller is a constant.

6.2.3 Velocity-following ability

The ability not only to reach the desired *waypoints* locations, but also, the *waypoints* velocities is defined as the velocity-following ability. This is not part of the path-following task, however, it is a bonus desired behaviour. The craft is able to follow very well the desired velocities in Case 1 and Case 2; follow reasonably well in Case 3, Case 4 and Case 5; and was not able to follow in the other cases.

The fact that the filter used for noise removal is a simple low-pass filter may have an influence on the damage in performance from the cases without noise emulation to the cases with noise emulation. The noise is Gaussian around the actual values, therefore has presence all over the frequency spectrum. The low-pass filter only filters out the high-frequency noise, however, leaves the low-frequency noise that can damage the craft's performance.

A possible explanation for the poor performances, with respect to velocity-following, may be the simplified models adopted. The craft suffered to follow the velocities when subjected to noise and it got even worse when it had to follow a curvy path. These both effects cause the craft to exhibit important rotation, important enough to affect significantly the surge velocity. This coupling effect is not considered in the monograph, and therefore may be a source of theoretical-practical mismatch for the unsuccessfully situations.

Another possible explanation for poor performance is the use of $R_w = R_{los} = 2 \cdot L_{pp}$. This is a big radius of acceptance. Although it is beneficial for the path-following task, it ruins the accuracy of the velocity control. Way before reaching exactly the desired *waypoint*, the craft changes it's desired *waypoint* to the next one, and therefore, also starts following another velocity.

When the craft actually crosses the *waypoint* it is in the middle of trying to reach the next *waypoint's* velocity.

6.2.4 Actuators behaviour

Ideally, the actuators (propeller and rudder) would exhibit smooth curves. When high frequency components are present, it causes the actuators to wear faster and the craft to oscillate frequently. With this in mind, the propeller rotation and rudder angle curves are analyzed.

In general the propeller rotation appeared to be relatively smooth, but in some cases like Case 9 has some moments of unwanted (medium and high amplitude) high frequency behaviour. The rudder presented (low and medium amplitude) high frequency components in all cases, however, it got worse in Case 8 with the high frequency component having more amplitude; and in Case 10 and Case 11 it shattered.

All above situations could be employed in a real-scenario, except the ones were the input shattered. These require a rethinking of the modelling, control or parameters used. For the other situations, a smoothing filter, such as moving average filter, could be used to smoothen these curves. However, these would add significant response delay that can damage a lot the path-following ability, therefore, it is necessary to analyze if the actuator wear and craft oscillation are bad enough to give away path-following performance.

6.2.5 Radius of acceptance influence

The radius of acceptance R_w does not influence the path-following task for the cases where the *waypoints* are collinear, because the path lines to follow are exactly the same. That is why, for these cases, this parameter does not vary. A small value $R_w = 50$ is chosen so that the surge velocity control can work as expected.

Outside the specific case mentioned before, this parameter plays a huge role. When the path has changes in direction, varying this parameters influences a lot the path-following task. If the radius of acceptance is big, the craft starts adapting to the new path line way before it reaches the exact location of the current *waypoint*. This is extremely helpful in settings like this one, where the craft has a slow response (large time constant). The drawback is that the craft doesn't reach exactly the desired *waypoint* and depending on the type of path and craft, may adapt to early to the next path line, losing grip with the actual path line.

In the cases exposed, it was extremely advantageous the use of a large radius of acceptance. The advantages can be seen by comparing Case 6 to Case 7 and Case 8 to Case 9. The errors were are all way smaller, for Case 7 and Case 9, which are the ones using larger radius.

6.2.6 Initial state influence

The initial state influences a lot the the path-following task. Especially, the initial yaw angle and surge velocity, which should be the ones that vary the most in real scenarios. This is evidenced when comparing Case 4 and Case 5 to Case 3.

In Case 4 the initial yaw angle is much less favourable, as the craft needs to turn a lot to get to the desired angle. This causes the craft to have very large errors at the beginning of the trajectory, which influences a lot the overall error and makes it not successful. The craft also oscillates around the desired path in higher frequency in this case.

In Case 5 the initial surge velocity is smaller. The surge velocity at 1 m/s is out of the assumptions adopted, which stated that surge velocity would be higher than 2 m/s. However, the craft did not deteriorate it's path-following performance. Also produced higher frequency oscillations around the desired path, and additionally, deteriorated a bit it's velocity-following performance.

6.2.7 Environmental condition influence

The modules were tuned at the *main environmental condition*. Since wave, wind and current forces are not modelled directly neither measured, these disturbances were embedded in the controllers, by tuning them to perform well at this environment. Unfortunately, the craft does not perform well when subjected to a different environmental condition. It would be necessary to tune the modules again, to the new environment, to obtain good performance.

The loss of performance can be seen by comparing Case 3 to Case 11. In the last case, the craft gets stuck in the middle of the way and does not complete the path. Further investigations and analysis should be done to understand better this behaviour and make the craft more robust.

6.2.8 Sensor noise emulation influence

Sensor noise emulation is an important part of the architecture, because approximates real-world scenarios, where systems rely on non-ideal sensors to get the actual state of the craft. The influence of turning on this block can be seen by comparing Case 1 to Case 3. When noise was included in the state signal, it made the task harder. Therefore, it can be seen that the resultant behaviour got worse in many dimensions. The velocity-following went from a very good to a reasonably good; the angular position instead of being damped, remained with high oscillations; and the actuators presented more harsh variations due to the inconsistent behaviour of noise and the oscillations just mentioned. That being said, in Case 3 the craft was still able to be successful at the path-following task.

CHAPTER

7

CONTRIBUTIONS

The main contributions of the project are:

- **The integration of *pydyna* with *ROS2*.** This integration opens a new door for the use of all the simulator's capabilities. The *ROS2* environment enables faster developing cycles, modular development, fast experimenting, scalability, additional robotics tools, and support of a large community. Also, it may provide the community with a high-fidelity ship maneuvering simulator as a *ROS2* package, which can contribute to a wide range of applications all over the world.
- **The development of a project to be used as a proof of concept and starting point for future maritime applications.** This project can guide TPN's first efforts with the *ROS2* framework. Although the *framework* is well known, there is not much material on the internet, especially for more complex applications. Furthermore, the project provides a *path-following* ship, which should be useful in the context of *fast-time* simulations. It is not expected that the project substitutes directly TPN's current piece of software, however, chunks of knowledge, modified versions of the project or applications built on top of it could be used. Applications other than *canal design* may also be impacted, such as *dynamic positioning* of ships.
- **The development of a very simple form of an autonomous ship.** The *path-following* ship obtained in this project is far for a fully autonomous ship application, as are the methods used far from the most accurate and general ones. With this in mind, the project does contribute with a simple form of autonomy, were the ship is able follow feasible desired paths, and in some cases, also reach the desired velocities; given a set of human-crafted *waypoints*. This monograph can help guide future projects related to providing some form of simple autonomy to ships, as a source of theory or starting point for practical implementations.

CHAPTER

8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

The conclusions are divided in two parts: a revisit to the monograph's objectives and a revisit to the monographs' results.

Revisiting objectives

An integration of *pydyna* with *ROS2* in the form of a *ROS2* package was built. This package can be used in a modular manner within the *ROS2* environment. Additionally, a proof-of-concept study case was developed to demonstrate an application using *pydyna* and *ROS2* together. The application was designed to be close to the *fast-time* simulations done by TPN, in actual routines regarding canal design. Finally, user documentation was provided for the packages.

Therefore, all the primary objetives were achieved.

With regards to the extensions: a path-following package was developed with control, sensing and visualization capabilities; a private maintenance documentation is being made at the moment, in conjunction with TPN staff; neither one of the packages were distributed to the scientific community because the subject was not discussed yet with TPN and a few code optimizations and test routines are to be done.

Therefore, one of the extensions was achieved.

Revisiting results

The results show that the craft was able to succeed at the path-following task, for a favourable initial yaw angle, at the *main environmental condition* and for simple paths such as collinear, slight zigzag or a slight curve. When the craft starts with a discrepant yaw angle, when paths have more harsh direction changes, or different environmental conditions are acting, the craft does not succeed at the task. These results can have a lot to do with the fact that the craft has a very slow time response (large craft) and was tuned in a specific environmental condition.

The craft was able to reach very well the desired velocities at the *waypoints* only in the collinear without sensor noise emulation case, but also managed to reach them reasonably well when noise emulation was on. In other cases, the velocity control did not work so well. These results have a lot to do with the simplified decoupled models for surge and yaw, and with the filters used. The decoupled models ignore important dynamics of the craft, for example, when the craft starts rotating it affects the surge velocity, but this effect is not modelled. The filter used for sensor noise removal was not the best option. The low-pass filter filtered high-frequency noise, however, low-frequency noise still remained.

The code for this project can be found here.

8.2 Future work

Due to the scope of the project and time constraints, some capabilities were not developed or not explored to their maximum potential. These can be divided in two large groups: code and methods. Code is about the implementation itself, not related to GNC theory. Methods are the theoretical techniques employed for the development of the GNC system. These two categories are explored below:

- **Code.** The code written for this project is far from optimized code. Therefore, there is plenty of room for making the system more efficient. Efficiency can not only make the simulations run faster, but are key for the quality of the path-following task, since the modules operate asynchronously. This means the faster the modules run, the lower is the delay of craft's actions.

  The graphical user interface can also have several improvements. As for the moment of this monograph, it only acts as a visualization tool. The GUI could also let the user change *waypoints* by dragging them in the screen, track the path developed by the craft and run several simulations without ending and starting the process again.

- **Methods.** The models used for surge and yaw are consequence of the adoption of a set of simplifying assumptions. When the scenario drifts away from these assumptions, the performance of the task drops a lot. The decoupling of surge from yaw is a strong factor that restrains the performance of the system to some level.

  Also, environmental disturbances are not modelled directly neither measured, therefore, they are a source of unmodelled dynamics. Tuning of the controllers solves some of the problems in theoretical modelling, however, is not the best approach, because it doesn't generalize. A robustness analysis would be a recommended next step, mainly to gain quality in different environmental conditions.

  The choice of *Sliding Mode* method for the surge controller may not have been the best choice, since it provides a relatively constant controller for each path line. In many cases the craft suffers some problems along the way, and would need to adjust it's control action severely, based on it's current state.

  With regards to the *PID* yaw controller, it did not damp well the oscillating yaw angle for most of the cases. Ideally, the craft should stabilize at the steady state yaw angle. Therefore, efforts can be made in this direction as well.

  As for navigation, the use of a Kalman Filter for state estimation and wave filtering would probably be the best option to avoid maintaining low-frequency noise, but adds a significant amount of complexity. Finally, with regards to guidance, path maneuvering or even trajectory tracking could be next steps to obtain a more powerful autonomous ship.

Tackling these problems can be subject of future work that wish to improve, or build on top of, this project. This will lead to a more efficient, robust, and user-friendly system that may bridge the gap between a proof of concept to a production level application.

BIBLIOGRAPHY

AMENDOLA, J.; MIURA, L. S.; COSTA, A. H. R.; COZMAN, F. G.; TANNURI, E. A. Navigation in restricted channels under environmental conditions: Fast-time simulation by asynchronous deep reinforcement learning. **IEEE Access**, Institute of Electrical and Electronics Engineers (IEEE), v. 8, p. 149199–149213, 2020. Citations on pages 37 and 38.

CAMPBELL, S.; NAEEM, W.; IRWIN, G. A review on improving the autonomy of unmanned surface vehicles through intelligent collision avoidance manoeuvres. **Annual Reviews in Control**, Elsevier BV, v. 36, n. 2, p. 267–283, dec 2012. Citations on pages 39, 42, and 46.

DAVIES, E. A. J. **Ship**. 2012. Available: <https://www.britannica.com/technology/ship/Dynamic-stability>. Citation on page 86.

EUROPEAN MARITIME SAFETY AGENCY. **ANNUAL OVERVIEW OF MARINE CASUALTIES ANDINCIDENTS**. 2017. Available: <https://www.isesassociation.com/wp-content/uploads/2017/11/Annual-overview-of-marine-casualties-and-incidents-2017_final.pdf>. Citation on page 29.

FEDORENKO, R.; GURENKO, B. Local and global motion planning for unmanned surface vehicle. **MATEC Web of Conferences**, EDP Sciences, v. 42, p. 01005, 2016. Citation on page 70.

FOSSEN, T. I. **Handbook of Marine Craft Hydrodynamics and Motion Control**. [S.l.]: John Wiley & Sons, Ltd, 2011. Citations on pages 39, 40, 41, 42, 43, 45, 46, 47, 48, 51, 53, 58, 59, and 63.

GITHUB. 2021. Available: <https://github.com>. Citation on page 167.

HASAN, K. S. B. **What, Why and How of ROS**. 2019. Available: <https://towardsdatascience.com/what-why-and-how-of-ros-b2f5ea8be0f3>. Citation on page 67.

INERTIAL LABS. **High Performance Advanced MEMS Industrial Tactical Grade Inertial Measurement Units**. [S.l.]. Citation on page 90.

IOVINO, S.; SAVVARIS, A.; TSOURDOS, A. Experimental testing of a path manager for unmanned surface vehicles in survey missions. **IFAC-PapersOnLine**, Elsevier BV, v. 51, n. 29, p. 226–231, 2018. Citation on page 70.

ISERMANN, R.; MüNCHHOF, M. **Identification of Dynamic Systems**. [S.l.]: Springer Berlin Heidelberg, 2011. Citations on pages 55 and 56.

KOFI, L. G. **The Ever Given – Suez Canal Accident: The Genesis to the current legal proceedings**. 2021. Available: <https://odomankoma.com/2021/04/19/the-ever-given-suez-canal-accident-the-genesis-to-the-current-legal-proceedings/>. Citation on page 30.

KRUPINSKI, S.; MAURELLI, F. Positioning aiding using LiDAR in GPS signal loss scenarios. In: **2018 IEEE 8th International Conference on Underwater System Technology: Theory and Applications (USYS)**. [S.l.]: IEEE, 2018. Citation on page 70.

LIU, Z.; ZHANG, Y.; YU, X.; YUAN, C. Unmanned surface vehicles: An overview of developments and challenges. **Annual Reviews in Control**, Elsevier BV, v. 41, p. 71–93, 2016. Citations on pages 39, 43, 46, 55, and 65.

NILLER, E. **The Robot Ships Are Coming ... Eventually**. 2020. Available: <https://www.wired.com/story/mayflower-autonomous-ships/#:~:text=Ithasidentifiedfourlevels,is\a\fully\autonomous\ship>. Citation on page 31.

NUMERICAL OFFSHORE TANK. **Expertise**. 2020. Available: <https://tpn.usp.br/simulador/Expertise.html>. Citations on pages 31 and 33.

____. **Homepage**. 2020. Available: <https://tpn.usp.br>. Citation on page 38.

____. **Simulators**. 2020. Available: <https://tpn.usp.br/simulador/Simulators.html>. Citation on page 32.

____. **Technical Studies**. 2020. Available: <https://tpn.usp.br/simulador/TechnicalStudies.html>. Citations on pages 32, 33, and 35.

____. **Profile**. 2021. Available: <https://www.linkedin.com/company/numerical-offshore-tank-tpn-/?originalSubdomain=br>. Citation on page 31.

SAN JOSE TECHNOLOGY, INC. **Marine GPS Receiver**. [S.l.], 2021. Citation on page 90.

SMITH, P.; DUNBABIN, M. High-fidelity autonomous surface vehicle simulator for the maritime RobotX challenge. **IEEE Journal of Oceanic Engineering**, Institute of Electrical and Electronics Engineers (IEEE), v. 44, n. 2, p. 310–319, apr 2019. Citation on page 70.

SPECIALIST, G. G. T. **What is Port and Starboard ?** Available: <https://www.globalior.com/what-is-port-and-starboard/port-and-starboard-diagram/>. Citation on page 52.

STEIGRAD, A. **Giant ship blocking Suez Canal freed, but economic impact looms**. 2021. Available: <https://nypost.com/2021/03/29/giant-ship-blocking-suez-canal-freed-but-economic-impact-looms/>. Citation on page 29.

TANNURI, E. A. **Desenvolvimento de metodologia de projeto de sistema de posicionamento dinâmico aplicado a operações em alto-mar.** Phd Thesis (PhD Thesis), 2002. Citation on page 63.

THE ROBOTICS BACK-END. **What is ROS?** 2021. Available: <https://roboticsbackend.com/what-is-ros/>. Citations on pages 67 and 68.

TPNSHIP. **pydyna documentation**. 2021. Available: <https://doccode.tpn.usp.br/projetos/tpnship/install/dyna_deploy_7_2_3/pydyna/doc/site/>. Citation on page 66.

TZENG, C.; CHEN, J. Fundamental properties of linear ship steering dynamic models. In: . [S.l.: s.n.], 2002. Citations on pages 51 and 54.

U-BLOX. **NEO-6u-blox 6 GPS Modules**. [S.l.]. Citation on page 90.

VELAMALA, S. S.; PATIL, D.; MING, X. Development of ROS-based GUI for control of an autonomous surface vehicle. In: **2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)**. [S.l.]: IEEE, 2017. Citation on page 70.

VILLA, J.; AALTONEN, J.; KOSKINEN, K. T. Path-following with LiDAR-based obstacle avoidance of an unmanned surface vehicle in harbor conditions. **IEEE/ASME Transactions on Mechatronics**, Institute of Electrical and Electronics Engineers (IEEE), v. 25, n. 4, p. 1812–1820, aug 2020. Citation on page 70.

WHAT IT COSTS. **How Much Did the Panama Canal Cost to Build**. 2016. Available: <https://www.whatitcosts.com/panama-canal-cost-build>. Citation on page 30.

WIKI, R. **Documentation**. 1. ed. [S.l.], 2020. Citations on pages 35 and 67.

ZHOU, C.; GU, S.; WEN, Y.; DU, Z.; XIAO, C.; HUANG, L.; ZHU, M. The review unmanned surface vehicle path planning: Based on multi-modality constraint. **Ocean Engineering**, Elsevier BV, v. 200, p. 107043, mar 2020. Citation on page 41.

APPENDIX

A

RELATED WORK

Table 23 – Related Work

Title	Objectives	Integration
Radar-based target tracking for Obstacle Avoidance for an Autonomous Surface Vehicle (ASV)	Obstacle avoidance, near ship harbors	"The system was designed for ROAZ II ASV belonging to INESC TEC/ISEP and implemented in Robot Operating System (ROS) for easier integration with the already existing software"
"Development of ROS-based GUI for control of an autonomous surface vehicle"	GUI for ROS	Integrated mainly with Qt for GUI and OpenCV for media transportation and processing. Qt already has built-in support in ROS, however, OpenCv required a ROS bridge
"A compact, low-cost unmanned surface vehicle for shallow inshore applications"	USV to aid and support police search teams in shallow-water and inshore reconnaissance operations	The system is built purely with ROS and it's packages
"Local and global motion planning for unmanned surface vehicle"	Control System for motion planning of USV	The author built a plugin package for hte ROS navigation stack
"Obstacle detection system design for an autonomous surface vehicle using a mechanical scanning sonar"	Use a mechanical scanning sonar (sensor) for object detection iin ASV	Author implemented a custom Object Detection ROS package

Title	Objectives	Integration
"High-Fidelity Autonomous Surface Vehicle Simulator for the Maritime RobotX Challenge"	ASV simulator	Integration of ASV simulator with ROS, like what we have to do
"Experimental Testing of a Path Manager for Unmanned Surface Vehicles in Survey Missions"	GNC for USV. An adaptive path planner that can switch between following stricltly waypoints or "interpolating" between lines to optimize	GNC in Matlab inntegrated with ROS
"AUV docking and recovery with USV: An experimental study"	Docking and recovery of ASV with an USV	Seems like the Matlab was integrated with ROS (based on author citing that object recognition was done in Matlab)
"Path-Following with LiDAR-Based Obstacle Avoidance of an Unmanned Surface Vehicle in Harbor Conditions"	Combining a pre-existing USV model with ROS in order to enable their model to navigate in harbor-like conditions with no human interference	GNC architecture of USV + ROS
"Development of a Simulator for the Study of Path Planning of An Autonomous Surface Vehicle in Lake Environments"	Simulator for ASV (Autonomous Surface Vehicles) testing two different frameworks, Matlab and Robotarium, for use in lake environments	ASV integrated with the framework of Matlab or Robotarium. Simulator is integrated to ROS
"Model-Based Control Architecture for a Twin Jet Unmanned Surface Vessel"	Implement control architecture for USV using LOS path algorithm with the control system done by Matlab's Simulink and ROS, in order to make the process easier for non-programmer researchers	Control architecture for USV using Matlab's Simulink and ROS
"Positioning aiding using LiDAR in GPS signal loss scenarios"	In order to avoid dangerous situations when a ship is in an area with many obstacles and no GPS signal, LiDAR would be used to gather information on the environment and generate positioning data	Data processing and positioning pipeline was implemented on a Reef Explorer 4 (REx4) with ROS environment, and using Point Cloud Library for operations on 3D point clouds

Title	Objectives	Integration
"Experimental validation of boundary tracking using the suboptimal sliding mode algorithm"	Validate the results obtained from the tracking algorithm used based on the 'suboptimal sliding mode' on their ASV	ASV running by a onshore laptop (Decision Making Module) via Wi-fi that uses Matlab and ROS
"Design of a self-moving autonomous buoy for the localization of underwater targets"	ASV buoy able to identify underwater targets, and communicate with them	Not many details beside the fact that ROS is used as the onboard software
"A low-cost and small USV platform for water quality monitoring"	Propose a open-source, low-cost USV for measuring near-surface water quality in real time	ROS master is connected to bluetooth, controller, mapviz and driver node
"MallARD: An autonomous aquatic surface vehicle for inspection and monitoring of wet nuclear storage facilities"	Develop an ASV able to monitor and inspect wet nuclear storage facilities, in order to minimize risks for human personnel	Ethernet Switch connecting Control laptop, Lidar (ROS) and a Raspberry Pi
"Cloud-based mission control of USV fleet: Architecture, implementation and experiments"	Integrate USVs to cloud-based architecture using ROS	Remote client can control the cloud server, that in turn sends commands to the fleet

Table 23 – Related Work

Source: Elaborated by the author.

APPENDIX

B

ROS SV PACKAGES

The *ROS Index* search only found one package[1], while in *GitHub* were found over 20 packages.

A table was constructed for all the packages found with the features mentioned at the end of section 3.2, plus the repository URL as the source. When information about the project is not presented, the field *Relevant Features* automatically receives the value "Not presented".

Table 24 – *ROS* packages for SV applications

Source	Summary	*Stars*	*Forks*	Information available	Relevant features
https://github.com/bsb808/usv_-gazebo_plugins	"Unmanned Surface Vehicle plugins for *Gazebo* simulation"	17	19	No	*Gazebo plugins*; *Gazebo* has a lot of community adoption and good documentation
https://github.com/hongsj235/End_-to_end_USV	"End_to_end learning to control autonomous ship(ROS/ Gazebo)"	3	1	Yes	Cameras as sensors; used in *Gazebo*
https://github.com/OUXT-Polaris/robotx_-packages	"ROS packages for Maritime RobotX Challenge"	22	7	Yes	"Simulation,Remote Viewer, Control System, etc..."
https://github.com/alexglzg/vtec_-usv_ros_pkg	"ROS package for USV guidance, navigation and control research."	1	1	No	Not presented

[1] Which is the first row of Table 24.

Source	Summary	*Stars*	*Forks*	Information available	Relevant features
https://github.com/vanttec/usv_simulation	"USV simulation ROS package"	2	1	No	Not presented
https://github.com/Unmanned-Surface-Vehicle/usv_-mission_planner	"USV Mission Planner ROS package."	2	1	Yes	Not presented
https://github.com/alexglzg/sensors	"ROS package to test USV GNC systems"	0	0	No	Not presented
https://github.com/vanttec/usv_master	"Master ROS Package for the VantTec USV."	0	0	No	Not presented
https://github.com/rohantiw96/USV-ROS-Stack	"ROS Stack for Unmanned Surface Vehicle"	0	0	No	Not presented
https://github.com/bsb808/usv_control	"ROS Low-Level PID Feedback Control for Unmanned Surface Vessel"	1	0	No	Not presented
https://github.com/vanttec/usv_-comms	"ROS Package for the communications between the USV and the Land Station using Digi XTends."	0	1	No	Not presented
https://github.com/vanttec/usv_control	"ROS package for usv autonomous control of position, speed and heading."	0	0	No	Not presented

Source	Summary	*Stars*	*Forks*	Information available	Relevant features
https://github.com/osrf/vrx	"Virtual RobotX (VRX) resources."	77	30	Yes	"This repository is the home to the source code and software documentation for the VRX Simulation and the VRX Challenge. Challenge documentation is available on the project wiki [...]"
https://github.com/ingeniarius-ltd/aquatic_simulator	"This work use is based on uuv_simulator and usv_vrx to create a test bed simulation world in Gazebo, to deploy multi-robot system MRS algorithm in ROS framework"	0	0	Yes	Simulation World with multiple USV's; based on *Gazebo's* Unmanned Underwater Vehicle (UUV) and *Virtual RobotX* resources
https://github.com/uf-mil/NaviGator	"NaviGator ASV on-board software"	30	55	Yes	project competing in *Maritime RobotX Challenge*
https://github.com/thomsten/ros_asv_system	Not presented	10	6	No	Contains modules for the core motion simulator, obstacle avoidance and path-following
https://github.com/srmainwaring/asv_wave_sim	"This package contains plugins that support the simulation of waves and surface vessels in Gazebo."	11	6	Yes	Requirements are *Ubuntu 18.04* and *ROS Melodic Morenia*
https://github.com/srv/xiroi_stack	"Xiroi ASV software architecture based on ROS"	0	0	No	Not presented

Source	Summary	*Stars*	*Forks*	Information available	Relevant features
https://github.com/disaster-robotics-proalertas/awa-sv	"Autonomous Water Assessment Surface Vehicle (AWA-SV) software meta repository"	0	0	Yes	Requirements are *Ubuntu* and *ROS Indigo/Kinetic*
https://github.com/iscumd/snapping_turtle	"A ROS based ASV garbage collection system for calm waters."	1	1	No	Not presented
https://github.com/CRAWlab/roboboat	"Repository of ROS code for the University of Louisiana at Lafayette entry into the 2020 RoboBoat competition. General documentation and development code is in a separate repository."	0	0	No	Not presented
https://github.com/jhlenes/complete_coverage	"ROS implementation of online complete coverage maneuvering for unmanned surface vehicles"	20	11	Yes	Details can be found in the author's master thesis and video implementation.
https://github.com/Liquid-ai/Plankton	"Open source simulator for maritime robotics researchers"	57	7	Yes	Actually an UUV simulator
https://github.com/OUXT-Polaris/ros_-ship_packages	"USV simulator for ROS"	53	32	No	Contains also control, simulated sensors, gazebo plugins, navigation recognition and visualization packages

Source	Summary	*Stars*	*Forks*	Information available	Relevant features
https://github.com/Southampton-Maritime-Robotics/Autonomous-Ship-and-Wavebuoys	"ROS and Arduino code for the autonomous operation of a 1:60 scale model of a tanker"	3	0	Yes	"[...] autonomous self-propulsion vessel which could self-monitor its performance (seakeeping, manoeuvring and powering)."
https://github.com/wangzhao9562/usv_navigation	"About usv_navigation Modified navigation pkg for USVs and under-actuated AUVs based on ros navigation stack Origin navigation stack"	10	6	Yes	Based on *ROS navigation stack*

Table 24 – *ROS* packages for SV applications

Source: GitHub (2021)

APPENDIX

C

PYTHON PUBLIC LIBRARIES

The following are the public libraries needed for *Python* in order to use *ROS*.

- atomicwrites==1.4.0
- attrs==21.2.0
- catkin-pkg==0.4.23
- cffi==1.14.6
- click==8.0.1
- colcon-cmake==0.2.26
- colcon-common-extensions==0.2.1
- colcon-core==0.6.1
- colcon-defaults==0.2.5
- colcon-devtools==0.2.2
- colcon-library-path==0.2.1
- colcon-metadata==0.2.5
- colcon-notification==0.2.13
- colcon-output==0.2.12
- colcon-package-information==0.3.3
- colcon-package-selection==0.2.10
- colcon-parallel-executor==0.2.4
- colcon-pkg-config==0.1.0

- colcon-powershell==0.3.6
- colcon-python-setup-py==0.2.7
- colcon-recursive-crawl==0.2.1
- colcon-ros==0.3.21
- colcon-test-result==0.3.8
- colorama==0.4.4
- coloredlogs==15.0.1
- coverage==5.5
- cryptography==3.4.8
- cycler==0.10.0
- dataclasses==0.8
- distlib==0.3.2
- distro==1.6.0
- docutils==0.17.1
- empy==3.3.4
- flake8==3.9.2
- flake8-blind-except==0.2.0
- flake8-builtins==1.5.3
- flake8-class-newline==1.6.0
- flake8-comprehensions==3.6.1
- flake8-deprecated==1.3
- flake8-docstrings==1.6.0
- flake8-import-order==0.18.1
- flake8-quotes==3.3.0
- Flask==2.0.1
- geographiclib==1.52
- humanfriendly==9.2
- ifcfg==0.22

- importlib-metadata==4.8.1
- importlib-resources==5.2.2
- iniconfig==1.1.1
- itsdangerous==2.0.1
- Jinja2==3.0.1
- kiwisolver==1.3.1
- lark-parser==0.12.0
- lxml==4.6.3
- MarkupSafe==2.0.1
- matplotlib==3.3.4
- mccabe==0.6.1
- mock==4.0.3
- mypy==0.761
- mypy-extensions==0.4.3
- netifaces==0.11.0
- numpy==1.19.5
- opencv-python==4.5.3.56
- packaging==21.0
- pep8==1.7.1
- Pillow==8.3.2
- pluggy==1.0.0
- psutil==5.8.0
- py==1.10.0
- pycairo==1.20.1
- pycodestyle==2.7.0
- pycparser==2.20
- pydocstyle==6.1.1
- pydot==1.4.2

- pyflakes==2.3.1
- pyparsing==2.4.7
- PyQt5==5.15.4
- PyQt5-Qt5==5.15.2
- PyQt5-sip==12.9.0
- pyreadline==2.1
- pytest==6.2.5
- pytest-cov==2.12.1
- pytest-mock==3.6.1
- pytest-repeat==0.9.1
- pytest-rerunfailures==10.1
- python-dateutil==2.8.2
- pywin32==301
- PyYAML==5.4.1
- rosdistro==0.8.3
- rospkg==1.3.0
- six==1.16.0
- snowballstemmer==2.1.0
- toml==0.10.2
- tornado==6.1
- typed-ast==1.4.3
- typing==3.7.4.3
- typing-extensions==3.10.0.2
- vcstool==0.3.0
- Werkzeug==2.0.1
- zipp==3.5.0

APPENDIX

D

PYDYNA_ROS CODE

The files that compose the *pydyna_simple* package, which is the first version of the ideal *pydyna_ros* package, can be found below.

D.1 pydyna_simple.py

```
import sys
import os
import numpy as np

import traceback

import pydyna

import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32
from std_msgs.msg import Bool
# custom interface
from path_following_interfaces.msg import State
#custom service
from path_following_interfaces.srv import InitValues

class PydynaSimpleNode(Node):

    def __init__(self):
        super().__init__('pydyna_simple_node')

        self.TIME_STEP = 0.1

        # need to declare them before
        self.declare_parameter('pkg_dir', './')
        self.declare_parameter('pkg_share_dir', './')
        # this will be overwritten by launch file
        self.declare_parameter('p3d_file', './')

        self.pkg_dir = self.get_parameter('pkg_dir').get_parameter_value().string_value
```

```
        self.pkg_share_dir = self.get_parameter('pkg_share_dir').get_parameter_value().string_value
        self.p3d_file = self.get_parameter('p3d_file').get_parameter_value().string_value

        self.num_simul = 0
        self.end_simul = 0

        self.subscription_end = self.create_subscription(
            Bool,
            '/end',
            self.callback_end,
            1)

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.server_init_simul = self.create_service(InitValues, '/init_simul', self.callback_init_simul)

        self.subscription_propeller = self.create_subscription(
            Float32,
            '/propeller_rotation',
            self.callback_propeller,
            1)

        self.subscription_rudder = self.create_subscription(
            Float32,
            '/rudder_angle',
            self.callback_rudder,
            1)

        self.publisher_state = self.create_publisher(State, 'state', 1)

    def callback_end(self):
        self.get_logger().info('User requested simulation to end')
        sys.exit()

    def callback_shutdown(self, _):
        sys.exit()

    def callback_init_simul(self, req, res):
        if self.num_simul != 0:
            pydyna.destroy_report(self.rpt)

        self.get_logger().info('Initializing Simulation')

        # self.propeller_rotation = 0
        # self.rudder_angle = 0
        self.propeller_rotation = req.surge
        self.rudder_angle = req.yaw
        self.subscriptions_synced = False

        self.rpt = pydyna.create_text_report(os.path.join(self.pkg_share_dir, 'logs', 'pydynalogs', f'pydyna_log_{s
        self.sim = pydyna.create_simulation(os.path.join(self.pkg_dir, 'config', self.p3d_file))

        self.ship = self.sim.vessels['104']
```

```
        x, y, theta = req.initial_state.position.x, req.initial_state.position.y, req.initial_state.position.
        u, v, r = req.initial_state.velocity.u, req.initial_state.velocity.v, req.initial_state.velocity.r
        self.ship.linear_position = [x, y, 0]
        self.ship.angular_position = [0, 0, theta]
        self.ship.linear_velocity = [u, v, 0]
        self.ship.angular_velocity = [0, 0, r]
        self.proppeler_counter = 0
        self.rudder_counter = 0

        self.num_simul += 1

        self.state = req.initial_state
        self.log_state('server')

        return res

    def callback_propeller(self, msg):
        self.get_logger().info('listened propeller rotation: %f' % msg.data)
        self.propeller_rotation = msg.data
        self.proppeler_counter += 1

    def callback_rudder(self, msg):
        self.get_logger().info('listened rudder angle: %f' % msg.data)
        self.rudder_angle = msg.data
        self.rudder_counter += 1

    def extrapolate_state(self):
        propeller = self.ship.thrusters['0']
        propeller.dem_rotation = self.propeller_rotation
        rudder = self.ship.rudders['0']
        # pydyna uses counterclockwise convention
        rudder.dem_angle = -self.rudder_angle

        self.sim.step()

        self.state.position.x = self.ship.linear_position[0]
        self.state.position.y = self.ship.linear_position[1]
        self.state.position.theta = self.ship.angular_position[2]%(2*np.pi)
        self.state.velocity.u = self.ship.linear_velocity[0]
        self.state.velocity.v = self.ship.linear_velocity[1]
        self.state.velocity.r = self.ship.angular_velocity[2]

        self.state.time += self.TIME_STEP

    def publish_state(self):
        self.publisher_state.publish(self.state)
        self.rpt.write(self.state.time, self.ship)
        self.log_state('publisher')

    def log_state(self, communicator):
        log_str = 'responded request with inital' if communicator == 'server' else 'published'
        self.get_logger().info(
            '%s state: {position: {x: %f, y: %f, theta: %f}, velocity: {u: %f, v: %f, r: %f}, time: %f}'
            % (
                log_str,
                self.state.position.x,
                self.state.position.y,
```

```
                self.state.position.theta, # yaw angle
                self.state.velocity.u,
                self.state.velocity.v,
                self.state.velocity.r,
                self.state.time
            )
        )

def main(args=None):
    try:
        rclpy.init(args=args)
        my_pydyna_node = PydynaSimpleNode()
        my_pydyna_node.get_logger().info('started main')
        while rclpy.ok():
            my_pydyna_node.get_logger().info('entered rclpy.ok loop')
            rclpy.spin_once(my_pydyna_node)
            if my_pydyna_node.proppeler_counter == my_pydyna_node.rudder_counter:
                if not my_pydyna_node.subscriptions_synced:
                    my_pydyna_node.extrapolate_state()
                    my_pydyna_node.publish_state()
                    my_pydyna_node.subscriptions_synced = True
            else:
                my_pydyna_node.subscriptions_synced = False
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
    except:
        print(traceback.format_exc())
    finally:
        my_pydyna_node.get_logger().info('Ended Simulation')
        my_pydyna_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

D.2 setup.py

```
import os
from glob import glob
from setuptools import setup

package_name = 'pydyna_simple'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages', [os.path.join('resource', package_name)]),
        (os.path.join('share', package_name), ['package.xml']),
        (os.path.join('lib', package_name, 'config'), glob('config/*')),
        (os.path.join('share', package_name, 'logs', 'mylogs'), []),
        (os.path.join('share', package_name, 'logs', 'pydynalogs'), []),
        (os.path.join('share', package_name, 'logs', 'roslogs'), []),
        (os.path.join('share', package_name, 'db'), []),
```

```
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='bruno',
    maintainer_email='bruno.c.scaglione@gmail.com',
    description='High Fidelity Ship Maneuvering Simulator',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'simul = pydyna_simple.pydyna_simple:main',
        ],
    },
)
```

D.3 package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/X
<package format="3">
  <name>pydyna_simple</name>
  <version>0.0.0</version>
  <description>High Fidelity Ship Maneuvering Simulator</description>
  <maintainer email="bruno.c.scaglione@gmail.com">Bruno</maintainer>
  <license>Apache License 2.0</license>

  <exec_depend>std_msgs</exec_depend>
  <exec_depend>pydyna</exec_depend>
  <exec_depend>os</exec_depend>
  <exec_depend>traceback</exec_depend>
  <exec_depend>sys</exec_depend>
  <exec_depend>numpy</exec_depend>

  <depend>path_following_interfaces</depend>
  <depend>rclpy</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

D.4 pydyna_simple.launch.py

```
import os

from ament_index_python.packages import get_package_share_directory, get_package_prefix
from launch import LaunchDescription
from launch.actions import ExecuteProcess
from launch_ros.actions import Node

# obs: not exaclty shure where generate_launch_description is invoked
# however absolute paths should do, with get_package_share_directory

def generate_launch_description():
    P3D_FILES = [
        'TankerL186B32_T085.p3d',
        'NoWaves_TankerL186B32_T085.p3d',
        'NoCurrent&Wind_TankerL186B32_T085.p3d',
        'NoWaves&Current&Wind_TankerL186B32_T085.p3d'
    ]

    pkg_share_dir = get_package_share_directory('pydyna_simple')
    pkg_install_dir = get_package_prefix('pydyna_simple')
    pkg_dir = os.path.join(pkg_install_dir, 'lib', 'pydyna_simple')
    logs_dir = os.path.join(pkg_share_dir, 'logs')
    p3d = P3D_FILES[0] # change p3d here

    os.environ['ROS_LOG_DIR'] = os.path.join(logs_dir, 'roslogs')
    # Set LOG format
    os.environ['RCUTILS_CONSOLE_OUTPUT_FORMAT'] = '[{severity} {time}] [{name}]: {message} ({function_name}() at {

    ld = LaunchDescription()

    rosbag_record_all = ExecuteProcess(
        cmd=['ros2', 'bag', 'record', '-a'], # '-o', 'rosbags'
        output='screen'
    )

    start_pydyna_simple_node = Node(
        package='pydyna_simple',
        executable='simul',
        name='pydyna_simple_node',
        output='screen',
        parameters=[
                {'pkg_share_dir': pkg_share_dir},
                {'pkg_dir': pkg_dir},
                {'p3d': p3d}
        ]
    )

    # start pydyna_simple_node
    ld.add_action(rosbag_record_all)
    ld.add_action(start_pydyna_simple_node)
    return ld
```

D.5 setup.cfg

```
[develop]
script_dir=$base/lib/pydyna_simple
[install]
install_scripts=$base/lib/pydyna_simple
```

APPENDIX

E

PATH_FOLLOWING CODE

The files that compose the *path_following* package can be found below.

E.1 backend.py

```
import os
import glob
import json
from datetime import datetime
import traceback

from flask import Flask, request

import rclpy
from rclpy.node import Node

from std_msgs.msg import Bool
# custom interfaces
from path_following_interfaces.msg import Waypoints
from path_following_interfaces.srv import InitValues

class Backend(Node):
    def __init__(self):
        super().__init__('backend_node')

        self.declare_parameter('db_dir', './')
        self.db_dir = self.get_parameter('db_dir').get_parameter_value().string_value

        self.end_msg = Bool()
        self.end_msg.data = True
        self.shutdown_msg = Bool()
        self.shutdown_msg.data = True

        self.init_values_srv = InitValues.Request()
        self.waypoints_msg = Waypoints()

        waypoints = {'position': {}}
        waypoints['position']['x'] = list(self.waypoints_msg.position.x)
```

```
        waypoints['position']['y'] = list(self.waypoints_msg.position.y)
        waypoints['velocity'] = list(self.waypoints_msg.velocity)

        self.save_waypoints(waypoints)

        self.client_init_setpoints = \
            self.create_client(InitValues, '/init_setpoints')

        self.client_init_surge_control = \
            self.create_client(InitValues, '/init_surge_control')

        self.client_init_yaw_control = \
            self.create_client(InitValues, '/init_yaw_control')

        self.client_init_simul = \
            self.create_client(InitValues, '/init_simul')

        self.publisher_waypoints = self.create_publisher(
            Waypoints,
            '/waypoints',
            1)

        self.publisher_end = self.create_publisher(
            Bool,
            '/end',
            1)

        self.publisher_shutdown = self.create_publisher(
            Bool,
            '/shutdown',
            1)

    def save_waypoints(self, waypoints):
        now = datetime.now()
        time_stamp = now.strftime("%Y_%m_%d-%H_%M_%S")

        waypoints_dir = os.path.join(self.db_dir, 'waypoints')
        waypoints_path = os.path.join(waypoints_dir, f'waypoints_{time_stamp}.json')

        # clean before
        files = glob.glob(os.path.join(waypoints_dir, '*.json'))
        for f in files:
            os.remove(f)

        with open(waypoints_path, 'w', encoding='utf-8') as f:
            json.dump(waypoints, f, ensure_ascii=False, indent=4)

    def log_state(self, state):
        self.get_logger().info(
            'Received from client initial state: {position: {x: %f, y: %f, theta: %f}, velocity: {u: %f, v: %f, r:
            % (
                state['position']['x'],
                state['position']['y'],
                state['position']['theta'], # yaw angle from west spanning [0, 2pi]
                state['velocity']['u'],
                state['velocity']['v'],
                state['velocity']['r'],
```

```
            )
        )

def wait_future(node, future_str):
    my_future = getattr(node, future_str)
    rclpy.spin_until_future_complete(node, my_future, timeout_sec=15)
    if my_future.done():
        try:
            return my_future.result(), 0
        except:
            return None, 0
    else:
        return None, 1

rclpy.init(args=None)
backend_node = Backend()

app = Flask(__name__)

@app.route("/waypoints", methods=['POST'])
def receive_waypoints():
    try:
        waypoints = request.json
        if not waypoints['from_gui']:
            backend_node.save_waypoints(waypoints)

            num_waypoints = len(waypoints['position']['x'])
            backend_node.get_logger().info('received from client %d waypoints' % num_waypoints)
            for i in range(num_waypoints):
                backend_node.get_logger().info(
                    'Received waypoint %d: %f %f %f' %
                    (i, waypoints['position']['x'][i], waypoints['position']['y'][i], waypoints['velocity'][i
                )

            backend_node.waypoints_msg.position.x = waypoints['position']['x']
            backend_node.waypoints_msg.position.y = waypoints['position']['y']
            backend_node.waypoints_msg.velocity = waypoints['velocity']

            backend_node.init_values_srv.waypoints.position.x = waypoints['position']['x']
            backend_node.init_values_srv.waypoints.position.y = waypoints['position']['y']
            backend_node.init_values_srv.waypoints.velocity = waypoints['velocity']

            backend_node.get_logger().info('Returning HTTP OK to client')
            return json.dumps({'success':True}), 200, {'ContentType':'application/json'}
        else:
            # TODO: service for getting the updated waypoints from venus server
            pass
    except:
        backend_node.get_logger().info(
            "Waypoints received from client are not valid\n"
            "Returning HTTP bad request to client"
        )
        return json.dumps({'success':False}), 400, {'ContentType':'application/json'}

@app.route("/initialCondition", methods=['POST'])
def receive_initial_condition():
    try:
```

```
        initial_condition = request.json

        backend_node.log_state(initial_condition)

        backend_node.init_values_srv.initial_state.position.x = \
            initial_condition['position']['x']
        backend_node.init_values_srv.initial_state.position.y = \
            initial_condition['position']['y']
        backend_node.init_values_srv.initial_state.position.theta = \
            initial_condition['position']['theta']
        backend_node.init_values_srv.initial_state.velocity.u = \
            initial_condition['velocity']['u']
        backend_node.init_values_srv.initial_state.velocity.v = \
            initial_condition['velocity']['v']
        backend_node.init_values_srv.initial_state.velocity.r = \
            initial_condition['velocity']['r']

        backend_node.get_logger().info('Returning HTTP OK to client')
        return json.dumps({'success':True}), 200, {'ContentType':'application/json'}
    except:
        backend_node.get_logger().info(
            "Initial condition received from client is not valid\n"
            "Returning HTTP bad request to client"
        )
        return json.dumps({'success':False}), 400, {'ContentType':'application/json'}

@app.route("/start")
def start_system():
    try:
        backend_node.get_logger().info("Starting system")

        backend_node.publisher_waypoints.publish(backend_node.waypoints_msg)

        # get initial setpoints
        backend_node.future_init_setpoints = backend_node.client_init_setpoints. \
            call_async(backend_node.init_values_srv)
        res_setpoints, timeout_setpoints = wait_future(backend_node, "future_init_setpoints")
        delattr(backend_node, "future_init_setpoints")
        service_failed = False if timeout_setpoints else True

        # set initial setpoints and get initial control actions
        backend_node.init_values_srv.surge = res_setpoints.surge
        backend_node.future_init_surge_control = backend_node.client_init_surge_control. \
            call_async(backend_node.init_values_srv)
        res_surge_control, timeout_surge_control = wait_future(backend_node, "future_init_surge_control")
        delattr(backend_node, "future_init_surge_control")
        service_failed = False if not service_failed and not timeout_surge_control else True


        backend_node.init_values_srv.yaw = res_setpoints.yaw
        backend_node.future_init_yaw_control = backend_node.client_init_yaw_control. \
            call_async(backend_node.init_values_srv)
        res_yaw_control, timeout_yaw_control = wait_future(backend_node, "future_init_yaw_control")
        delattr(backend_node, "future_init_yaw_control")
        service_failed = False if not service_failed and not timeout_yaw_control else True

        # set inital state and control action of the simulation
```

```
        backend_node.init_values_srv.surge = res_surge_control.surge
        backend_node.init_values_srv.yaw = res_yaw_control.yaw
        backend_node.future_init_simul = backend_node.client_init_simul. \
            call_async(backend_node.init_values_srv)
        _, timeout_simul = wait_future(backend_node, "future_init_simul")
        delattr(backend_node, "future_init_simul")
        service_failed = False if not service_failed and not timeout_simul else True

        backend_node.get_logger().info('returning HTTP OK to client')
        return json.dumps({'success':True}), 200, {'ContentType':'application/json'}
    except:
        if not service_failed:
            backend_node.get_logger().info('returning HTTP Gateaway timedout to client')
            return json.dumps({'success':False}), 504, {'ContentType':'application/json'}
        else:
            backend_node.get_logger().info('returning HTTP internal server error to client')
            return json.dumps({'success':False}), 500, {'ContentType':'application/json'}

@app.route("/end")
def end_simul():
    try:
        backend_node.get_logger().info("Ending system")
        backend_node.publisher_end.publish(backend_node.end_msg)

        backend_node.get_logger().info('returning HTTP OK to client')
        return json.dumps({'success':True}), 200, {'ContentType':'application/json'}
    except:
        backend_node.get_logger().info('returning HTTP internal server error to client')
        return json.dumps({'success':False}), 500, {'ContentType':'application/json'}

@app.route("/shutdown")
def shutdown_nodes():
    try:
        backend_node.get_logger().info("Shutting down nodes")
        backend_node.publisher_shutdown.publish(backend_node.shutdown_msg)

        backend_node.get_logger().info('Returning HTTP OK to client')
        return json.dumps({'success':True}), 200, {'ContentType':'application/json'}
    except:
        backend_node.get_logger().info('returning HTTP internal server error to client')
        return json.dumps({'success':False}), 500, {'ContentType':'application/json'}

def main():
    try:
        app.run() # port 5000 by default
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
    except:
        print(traceback.format_exc())
    finally:
        backend_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

E.2 control_allocation.py

```
import sys
import os
import glob
import traceback
import math

import numpy as np

import matplotlib.pyplot as plt

import rclpy
from rclpy.node import Node

from std_msgs.msg import Bool
from std_msgs.msg import Float32

# custom iterfaces
from path_following_interfaces.msg import State

class ControlAllocation(Node):
    def __init__(self):
        super().__init__('control_allocation_node')

        self.declare_parameter('plots_dir', './')
        self.plots_dir = self.get_parameter('plots_dir').get_parameter_value().string_value

        self.TIME_STEP = 0.1

        # controller parameters
        self.C1 = 0.036
        self.C2 = 3.53
        self.C3 = 0.06696

        self.PROPELLER_SAT = 1.75 # Hz

        self.propeller_history = [] #debugging

        self.rotation_msg = Float32()

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_filtered_state = self.create_subscription(
            State,
            '/filtered_state',
            self.callback_filtered_state,
            1)

        self.subscription_propeller_thrust = self.create_subscription(
            Float32,
            '/propeller_thrust',
            self.callback_propeller_thrust,
```

```
            1)

        self.publisher_propeller_rotation = self.create_publisher(
            Float32,
            '/propeller_rotation',
            1)

    def callback_shutdown(self, _):
        sys.exit()

    def callback_filtered_state(self, msg):
        self.get_logger().info('listened filtered surge velocity: %f' % msg.velocity.u)
        self.surge_velocity = msg.velocity.u

    def callback_propeller_thrust(self, msg):
        self.get_logger().info('listened propeller thrust: %f' % msg.data)
        propeller_rotation_msg = self.control_allocation(msg.data, self.surge_velocity)
        self.publisher_propeller_rotation.publish(propeller_rotation_msg)
        self.get_logger().info('published propeller rotation: %f' % propeller_rotation_msg.data)

    def control_allocation(self, tau, u):
        if tau > 0:
            Np = self.C1*(math.sqrt(self.C2*(u**2) + tau)) + self.C3*u
        else:
            Np = -(self.C1*(math.sqrt(self.C2*(u**2) - tau)) + self.C3*u)
        # saturation of the propeller (with 1% safety margin)
        # real sat is 1.75 Hz
        Np = max(-self.PROPELLER_SAT*0.99, min(Np, self.PROPELLER_SAT*0.99))
        self.propeller_history.append(Np)
        self.rotation_msg.data = Np
        return self.rotation_msg

    def generate_plots(self):
        # clean before
        files = glob.glob(os.path.join(self.plots_dir, 'propellerRotation*.png'))
        for f in files:
            os.remove(f)

        params = {'mathtext.default': 'regular'}
        plt.rcParams.update(params)

        t = self.TIME_STEP*np.array(range(len(self.propeller_history)))
        fig, ax = plt.subplots(1)
        ax.set_title("Proppeler rotation")
        ax.plot(t, self.propeller_history)
        ax.set_xlabel("t [s]")
        ax.set_ylabel(r"$n_p\;[Hz]$")
        # ax.set_ylim(min(self.propeller_history),max(self.propeller_history))

        graphics_file = "propellerRotation.png"
        fig.savefig(os.path.join(self.plots_dir, graphics_file))

def main(args=None):
    try:
        rclpy.init(args=args)
        control_allocation_node = ControlAllocation()
        rclpy.spin(control_allocation_node)
```

```
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
        control_allocation_node.get_logger().info('Stopped with user interrupt')
    except SystemExit:
        pass
    except:
        print(traceback.format_exc())
    finally:
        control_allocation_node.generate_plots()
        control_allocation_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

E.3 gps_imu_simul.py

```
'''
Datasheets used:
    1. GPS 1: https://pdf.nauticexpo.com/pdf/san-jose-technology-inc/marine-gps-receiver/23414-76823.html
    2. GPS 2 (just for velocity): {
        https://www.u-blox.com/sites/default/files/products/documents/NEO-6_DataSheet_(GPS.G6-HW-09005).pdf or
        https://docs.rs-online.com/c0e9/0900766b80df94d1.pdf or
        https://www.generationrobots.com/media/GP-635T-121130-datasheet.pdf or
        https://4.imimg.com/data4/SO/GH/MY-23669504/gps-shield-skg13c-module.pdf
    }
    3. IMU (indutrial): https://inertiallabs.com/wp-content/uploads/2020/09/IMU-P_Datasheet.rev3_.3_Sept_2020.pdf
'''

import sys
import os
import glob
import traceback

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

import rclpy
from rclpy.node import Node

from std_msgs.msg import Bool
# custom interface
from path_following_interfaces.msg import State

class GpsImuSimulator(Node):
    def __init__(self):
        super().__init__('gps_imu_simulator_node')

        self.declare_parameter('plots_dir', './')
        self.plots_dir = self.get_parameter('plots_dir').get_parameter_value().string_value

        # GPS_rate == 10 # [Hz]
        # IMU_rate == 2000 # [Hz]
        # both rates are faster than the simulation
```

```
        # (which is 10-e Hz where e is due to the time do do 1 step of computations)
        # therefore wont have sampling effect

        self.TIME_STEP = 0.1

        # assuming sensors are already calibrated (no bias)
        # assuming same variance for x and y, and 0 covariance
        self.SIGMA_X = 5.46112744197 # GPS 1: from horizontal acc
        self.SIGMA_Y = 5.46112744197 # GPS 1: from horizontal acc
        self.SIGMA_THETA = 0.0523599 # GPS 1: from heading acc
        # self.sigma_u is calculated dynamically
        # self.sigma_v is calculated dynamically
        self.SIGMA_R = 0.0005 # IMU: from random walk

        self.SIGMA_XDOT = 0.1 # GPS 2: from velocity acc
        self.SIGMA_YDOT = 0.1 # GPS 2: from velocity acc

        self.state_history = [[],[],[],[],[],[]]
        self.simulated_state_history = [[],[],[],[],[],[]]

        # State() has by default all zeros
        self.last_two_states = collections.deque([State()], maxlen=2)

        self.xs_msg = State()

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_state = self.create_subscription(
            State,
            '/state',
            self.callback_state,
            1)

        self.publisher_simulated_state = self.create_publisher(
            State,
            '/simulated_state',
            1)

    def callback_shutdown(self, _):
        sys.exit()

    def callback_state(self, msg):
        self.log_state(msg, 'subscriber')

        self.state_history[0].append(msg.position.x)
        self.state_history[1].append(msg.position.y)
        self.state_history[2].append(msg.position.theta)
        self.state_history[3].append(msg.velocity.u)
        self.state_history[4].append(msg.velocity.v)
        self.state_history[5].append(msg.velocity.r)

        self.last_two_states.appendleft(msg)
        self.calculate_velocity_sigmas()
```

```
        simulated_state_msg = self.state_simul(msg)
        self.publisher_simulated_state.publish(simulated_state_msg)
        self.log_state(simulated_state_msg, 'publisher')

    def calculate_velocity_sigmas(self):
        x_dot = ( list(self.last_two_states)[0].position.x - list(self.last_two_states)[1].position.x )/self.TIME_S
        y_dot = ( list(self.last_two_states)[0].position.y - list(self.last_two_states)[1].position.y )/self.TIME_S
        sigma_theta = self.SIGMA_THETA
        theta = list(self.last_two_states)[0].position.theta
        # error propagation ([u, v].T = R@[xdot, ydot].T linear transformation or rotation matrix)
        self.sigma_u = ( (np.cos(theta)*self.SIGMA_XDOT)**2 + (np.sin(theta)*self.SIGMA_YDOT)**2 \
            + ((-np.sin(theta)*x_dot + np.cos(theta)*y_dot)*sigma_theta)**2 )**0.5
        self.sigma_v = ( (-np.sin(theta)*self.SIGMA_XDOT)**2 + (np.cos(theta)*self.SIGMA_YDOT)**2 \
            + ((-np.cos(theta)*x_dot - np.sin(theta)*y_dot)*sigma_theta)**2 )**0.5

    def state_simul(self, x):
        self.xs_msg.position.x = x.position.x + np.random.normal(0, self.SIGMA_X) # gps
        self.xs_msg.position.y = x.position.y + np.random.normal(0, self.SIGMA_Y) # gps
        self.xs_msg.position.theta = x.position.theta + np.random.normal(0, self.SIGMA_THETA) # gyrocompass

        self.xs_msg.velocity.u = x.velocity.u + np.random.normal(0, self.sigma_u) # gps
        self.xs_msg.velocity.v = x.velocity.v + np.random.normal(0, self.sigma_v) # gps
        self.xs_msg.velocity.r = x.velocity.r + np.random.normal(0, self.SIGMA_R) # imu

        self.xs_msg.time = x.time

        self.simulated_state_history[0].append(self.xs_msg.position.x)
        self.simulated_state_history[1].append(self.xs_msg.position.y)
        self.simulated_state_history[2].append(self.xs_msg.position.theta)
        self.simulated_state_history[3].append(self.xs_msg.velocity.u)
        self.simulated_state_history[4].append(self.xs_msg.velocity.v)
        self.simulated_state_history[5].append(self.xs_msg.velocity.r)

        return self.xs_msg

        # # code below is to not implement gps_imu_simul
        # self.simulated_state_history[0].append(x.position.x)
        # self.simulated_state_history[1].append(x.position.y)
        # self.simulated_state_history[2].append(x.position.theta)
        # self.simulated_state_history[3].append(x.velocity.u)
        # self.simulated_state_history[4].append(x.velocity.v)
        # self.simulated_state_history[5].append(x.velocity.r)

        return x

    def log_state(self, state, communicator):
        log_str = 'listened' if communicator == 'subscriber' else 'published simulated'
        self.get_logger().info(
            '%s state: {position: {x: %f, y: %f, theta: %f}, velocity: {u: %f, v: %f, r: %f}, time: %f}'
            % (
                log_str,
                state.position.x,
                state.position.y,
                state.position.theta, # yaw angle
                state.velocity.u,
                state.velocity.v,
                state.velocity.r,
```

```
                state.time
            )
        )

    def generate_plots(self):

        params = {'mathtext.default': 'regular'}
        plt.rcParams.update(params)

        # clean before
        files = glob.glob(os.path.join(self.plots_dir, 'simulatedState', '*.png'))
        for f in files:
            os.remove(f)

        t = self.TIME_STEP*np.array(range(len(self.simulated_state_history[0])))
        ss_dir = "simulatedState"
        simulated_state_props = [
            {
                "title": "Simulated Linear Position X",
                "ylabel": "x [m]",
                "file": "simulatedLinearPositionX.png"
            },
            {
                "title": "Simulated Linear Position Y",
                "ylabel": "y [m]",
                "file": "simulatedLinearPositionY.png"
            },
            {
                "title": "Simulated Angular Position Theta",
                "ylabel": r"$\theta\;[rad\;(from\;east\;counterclockwise)]$",
                "file": "simulatedAngularPositionTheta.png"
            },
            {
                "title": "Simulated Linear Velocity U",
                "ylabel": "u [m/s]",
                "file": "simulatedLinearVelocityU.png"
            },
            {
                "title": "Simulated Linear Position V",
                "ylabel": "v [m/s (port)]",
                "file": "simulatedLinearVelocityV.png"
            },
            {
                "title": "Simulated Angular Velocity R",
                "ylabel": "r [rad/s (counterclockwise)]",
                "file": "simulatedAngularVelocityR.png"
            },
        ]

        files = glob.glob(os.path.join(self.plots_dir, 'state', '*.png'))
        for f in files:
            os.remove(f)

        s_dir = "state"
        state_props = [
            {
                "title": "Linear Position X",
```

```
            "ylabel": "x [m]",
            "file": "linearPositionX.png"
        },
        {
            "title": "Linear Position Y",
            "ylabel": "y [m]",
            "file": "linearPositionY.png"
        },
        {
            "title": "Angular Position Theta",
            "ylabel": r"$\theta\;[rad\;(from\;east\;counterclockwise)]$",
            "file": "angularPositionTheta.png"
        },
        {
            "title": "Linear Velocity U",
            "ylabel": "u [m/s]",
            "file": "linearVelocityU.png"
        },
        {
            "title": "Linear Position V",
            "ylabel": "v [m/s (port)]",
            "file": "linearVelocityV.png"
        },
        {
            "title": "Angular Velocity R",
            "ylabel": "r [rad/s (counterclockwise)]",
            "file": "angularVelocityR.png"
        },
    ]
    dirs = [ss_dir, s_dir]
    propss = [simulated_state_props, state_props]
    histories = [self.simulated_state_history, self.state_history]
    for (dir, props, history) in list(zip(dirs, propss, histories)):
        for j in range(len(history)):
            fig, ax = plt.subplots(1)
            ax.set_title(props[j]["title"])
            ax.plot(t, history[j])
            ax.set_xlabel("t [s]")
            ax.set_ylabel(props[j]["ylabel"])
            # ax.set_ylim(min(history[j]), max(history[j]))

            fig.savefig(os.path.join(self.plots_dir, dir, props[j]["file"]))

    ############## report plots
    files = glob.glob(os.path.join(self.plots_dir, "reportPlots", "gpsImuSimul", '*.png'))
    for f in files:
        os.remove(f)

    # u simulated and u filtered together
    fig, ax = plt.subplots(1)
    ax.set_title("Linear Velocity U")
    ax.plot(t, self.simulated_state_history[3])
    ax.plot(t, self.state_history[3])
    ax.set_xlabel(r"$t\;[s]$")
    ax.set_ylabel("u [m/s]")
    ax.legend([r"$u$ from sensor", r"$u$ real"])
    fig.savefig(os.path.join(self.plots_dir, "reportPlots", "gpsImuSimul", "surgeReal&Simulated.png"))
```

```
        # theta simulated and theta filtered together
        fig, ax = plt.subplots(1)
        ax.set_title(r"Angular Position $\theta$")
        ax.plot(t, self.simulated_state_history[2])
        ax.plot(t, self.state_history[2])
        ax.set_xlabel("t [s]")
        ax.set_ylabel(r"$\theta\;[rad]$")
        ax.legend([r"$\theta$ from sensor", r"$\theta$ real"])
        fig.savefig(os.path.join(self.plots_dir, "reportPlots", "gpsImuSimul", "yawReal&Simulated.png"))

def main(args=None):
    try:
        rclpy.init(args=args)
        gps_imu_simulator_node = GpsImuSimulator()
        rclpy.spin(gps_imu_simulator_node)
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
        gps_imu_simulator_node.get_logger().info('Stopped with user interrupt')
    except SystemExit:
        pass
    except:
        print(traceback.format_exc())
    finally:
        gps_imu_simulator_node.generate_plots()
        gps_imu_simulator_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

E.4 los_guidance.py

```
import sys
import os
import glob
import traceback

import matplotlib.pyplot as plt
import math
import numpy as np
from sympy import symbols, Eq, solve
# import stackprinter

import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32
from std_msgs.msg import Bool
#custom service
from path_following_interfaces.msg import State, Control
from path_following_interfaces.srv import InitValues

class LosGuidance(Node):
    def __init__(self):
```

```
        super().__init__('los_guidance_node')

        self.declare_parameter('plots_dir', './')
        self.plots_dir = self.get_parameter('plots_dir').get_parameter_value().string_value

        self.TIME_STEP = 0.1

        # los parameters
        self.SHIP_LENGHT = 186.4
        # los radius
        self.R = self.SHIP_LENGHT*2

        ##########################################
        # VERY IMPORTANT, CHANGING THIS VALUE ALLOWS TRADEOFF
        ## BETWEEN SMOOTH TRAJECTORY (higher value) AND REACHING WAYPOINTS PRECISELY
        # When craft is inside acceptance radius for a waypoint that
        # it considers waypoint was reached
        # tuned for:
        # self.R_ACCEPTANCE = 50 # use this for linear waypoints
        self.R_ACCEPTANCE = self.SHIP_LENGHT*2  # use this for zigzag waypoints
        ##########################################

        # final radius of accceptance
        self.R_ACCEPTANCE_FINAL = 50

        # Size of radius around last waypoint.
        # When craft is outside this radius it should have stopped
        # self.R_STOP = 100.0

        self.desired_values_history = {
            'values': [[],[]],
            'time': []
        }

        self.path_error = []
        self.width_error = []

        # When true, completed all waypoints
        self.no_more_waypoints = False

        # When true, craft is within self.R_ACCEPTANCE_FINAL of the final waypoint
        self.finished = False

        # index of waypoint the ship has to reach next (first waypoint is starting position)
        self.current_waypoint = 1

        self.des_yaw_msg = Control()
        # self.des_velocity_msg = Float32()
        self.des_velocity_msg = Control()

        self.shutdown_msg = Bool()
        self.shutdown_msg.data = True

        self.server_init_setpoints = self.create_service(
            InitValues, '/init_setpoints', self.callback_init_setpoints
        )
```

```
        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_filtered_state = self.create_subscription(
            State,
            '/filtered_state',
            self.callback_filtered_state,
            1)

        self.publisher_desired_yaw_angle = self.create_publisher(
            Control,
            '/desired_yaw_angle',
            1)

        self.publisher_desired_surge_velocity = self.create_publisher(
            Control,
            '/desired_surge_velocity',
            1)

        self.publisher_shutdown = self.create_publisher(
            Bool,
            '/shutdown',
            1)

    def callback_shutdown(self, _):
        sys.exit()

    def callback_init_setpoints(self, req, res):
        req.waypoints.position.x.insert(0, req.initial_state.position.x)
        req.waypoints.position.y.insert(0, req.initial_state.position.y)
        req.waypoints.velocity.insert(0, req.initial_state.velocity.u)
        self.waypoints = req.waypoints # {position: {x: [...], y: [...]} velocity: [...]}

        self.get_steady_state_yaw_angles(self.waypoints)

        self.num_waypoints = len(req.waypoints.position.x)
        self.get_logger().info('initial waypoint + listened %d waypoints' % (self.num_waypoints-1))
        for i in range(self.num_waypoints):
            self.get_logger().info('listened waypoint %d: %f %f %f' % (i, req.waypoints.position.x[i], req.wa

        des_velocity_msg, des_yaw_msg = self.los(req.initial_state)
        res.surge, res.yaw = des_velocity_msg.desired_value, des_yaw_msg.desired_value
        return res

    def get_steady_state_yaw_angles(self, waypoints):
        self.desired_steady_state_yaw_angles = []
        for i in range(1, len(waypoints.velocity)):
            desired_steady_state_yaw_angle = math.atan2(
                waypoints.position.y[i]-waypoints.position.y[i-1],
                waypoints.position.x[i]-waypoints.position.x[i-1]
            )
            self.desired_steady_state_yaw_angles.append(desired_steady_state_yaw_angle)

    def callback_filtered_state(self, msg):
```

```
        try: # need have received waypoints first
            self.log_state(msg)
            des_velocity_msg, des_yaw_msg = self.los(msg)
            self.publisher_desired_yaw_angle.publish(des_yaw_msg)
            self.get_logger().info('published desired yaw angle: %f' % des_yaw_msg.desired_value)
            self.publisher_desired_surge_velocity.publish(des_velocity_msg)
            self.get_logger().info('published desired velocity: %f' % des_velocity_msg.desired_value)
        except AttributeError:
            self.get_logger().info('Has not received waypoints yet, will ignore listened state')

    def reached_next_waypoint(self, xf, R_acceptance):
        x, y = xf.position.x, xf.position.y
        idx = self.current_waypoint
        wx_next , wy_next  = self.waypoints.position.x[idx], self.waypoints.position.y[idx]
        return 1 if (x-wx_next)**2 + (y-wy_next)**2 <= R_acceptance**2 else 0

    def missed_waypoint(self, xf):
        # when craft passes line that is orthogonal to los line and
        # intercept the next waypoint
        # instead of going back trying to reach waypoint missed,
        # craft will change to next waypoint
        x, y = xf.position.x, xf.position.y
        idx = self.current_waypoint
        wx, wy = self.waypoints.position.x[idx-1], self.waypoints.position.y[idx-1]
        wx_next , wy_next = self.waypoints.position.x[idx], self.waypoints.position.y[idx]

        if (wx_next - wx) == 0:
            self.get_logger().info('1')
            passed_wnext = True if (wy_next-wy)*(y - wy_next) > 0 else False
        elif (wy_next - wy) == 0:
            passed_wnext = True if (wx_next-wx)*(x - wx_next) > 0 else False
        else:
            c = (wy_next - wy)/(wx_next - wx)
            self.get_logger().info('c: %f' % c)
            a_wnext = -1/c
            b_wnext = wy_next - a_wnext*wx_next

            passed_wnext = True if (wx_next-wx)*a_wnext*(a_wnext*x + b_wnext - y) > 0 \
                else False

        self.get_logger().info('passed_wnext: True') if passed_wnext else \
            self.get_logger().info('passed_wnext: False')

        return passed_wnext

    def get_xy_los(self, x, y, wx, wy, wx_next, wy_next):

        x_los, y_los = symbols('x_los, y_los')
        eq1 = Eq((x_los-x)**2 + (y_los-y)**2, self.R**2)

        if (wx_next - wx) == 0:
            eq2 = Eq(x_los, wx)
        elif (wy_next - wy) == 0:
            eq2 = Eq(y_los, wy)
        else:
            eq2 = Eq(((wy_next - wy)/(wx_next - wx)), (y_los - wy)/(x_los - wx))
```

```
        sol = solve([eq1, eq2], [x_los, y_los])
        soln = [tuple(v.evalf() for v in s) for s in sol] # evaluated numerically

        x_los1, y_los1 = soln[0]
        self.get_logger().info('x_los1: %f, y_los1: %f' % (x_los1, y_los1))
        x_los2, y_los2 = soln[1]
        self.get_logger().info('x_los2: %f, y_los2: %f' % (x_los2, y_los2))

        # dot product -> projection of (los intercept - craft positio) vector
        # onto los line vector (connecting waypoints)
        (x_los, y_los) = (x_los1, y_los1) if (wx_next-wx)*(x_los1-x) + (wy_next-wy)*(y_los1-y) > 0 else (x_lo

        self.get_logger().info('wx_next: %f' % wx_next)
        self.get_logger().info('x_los: %f, y_los: %f' % (x_los, y_los))

        return (x_los, y_los)

    def get_current_width_error(self, idx, theta):
        if math.radians(90) < theta < math.radians(180) or math.radians(270) < theta < math.radians(360):
            self.get_logger().info('theta = %f : in second or forth quadrants' % theta)
            zeta = math.radians(90) - self.desired_steady_state_yaw_angles[idx-1]
            beta = math.radians(90) - zeta
            alfa = beta - (theta - math.radians(90))
            self.get_logger().info('alfa: %f' % alfa)
            if self.path_error[-1] == 0:
                if math.radians(90) < theta < math.radians(180):
                    return self.path_error[-1] + abs((self.SHIP_LENGHT/2)*np.cos(alfa))
                else:
                    return self.path_error[-1] - abs((self.SHIP_LENGHT/2)*np.cos(alfa))
            elif self.path_error[-1] > 0:
                return self.path_error[-1] + abs((self.SHIP_LENGHT/2)*np.cos(alfa))
            else:
                return self.path_error[-1] + -abs((self.SHIP_LENGHT/2)*np.cos(alfa))
        else:
            self.get_logger().info('theta = %f : in first or third quadrants' % theta)
            chi = theta - self.desired_steady_state_yaw_angles[idx-1]
            alfa = math.radians(90) - chi
            self.get_logger().info('alfa: %f' % alfa)
            if self.path_error[-1] == 0:
                if 0 <= theta <= math.radians(90):
                    return self.path_error[-1] + abs((self.SHIP_LENGHT/2)*np.cos(alfa))
                else:
                    return self.path_error[-1] - abs((self.SHIP_LENGHT/2)*np.cos(alfa))
            elif self.path_error[-1] > 0:
                return self.path_error[-1] + abs((self.SHIP_LENGHT/2)*np.cos(alfa))
            else:
                return self.path_error[-1] + -abs((self.SHIP_LENGHT/2)*np.cos(alfa))

    def append_current_errors(self, x, y, theta, wx, wy, wx_next, wy_next):
        idx = self.current_waypoint
        use_current_waypoint = True
        if idx > 1:
            # cx + d is line connecting waypoints
            # ax +  b is orthogonal to cx + d, passing through a waypoint or through the craft
            wx_before, wy_before = self.waypoints.position.x[idx-2], self.waypoints.position.y[idx-2]

            calc_intercept_result_with_check = self.calc_intercept(x, y, wx_before, wy_before, wx, wy, check=
```

```
            if calc_intercept_result_with_check[3]:
                (xi, yi, positive_error, _) = self.calc_intercept(x, y, wx, wy, wx_next, wy_next)
            else:
                use_current_waypoint = False
                (xi, yi, positive_error, _) = calc_intercept_result_with_check
        else:
            (xi, yi, positive_error, _) = self.calc_intercept(x, y, wx, wy, wx_next, wy_next)

        current_path_error = ((x - xi)**2 + (y - yi)**2)**0.5
        if not positive_error:
            current_path_error = - current_path_error
        self.get_logger().info('current_path_error: %f' % current_path_error)

        self.path_error.append(current_path_error)

        if use_current_waypoint:
            current_width_error = self.get_current_width_error(idx, theta)
        else:
            current_width_error = self.get_current_width_error(idx-1, theta)

        self.width_error.append(current_width_error)

    @staticmethod
    def calc_intercept(x, y, wx1, wy1, wx2, wy2, check=False):
        if check:
            xi = None
            yi = None
            positive_error = None
            not_between_waypoints = False
            if (wy1 - wy2) == 0:
                if (wx2-wx1)*(x-wx2) < 0:
                    xi = x
                    yi = wy2

                    positive_error = True if y > wy1 else False
                else:
                    not_between_waypoints = True
            elif (wx1 - wx2) == 0:
                if (wy2-wy1)*(y-wy2) < 0:
                    xi = wx2
                    yi = y

                    positive_error = True if x < wx1 else False
                else:
                    not_between_waypoints = True
            else:
                c = (wy1 - wy2)/(wx1 - wx2)
                d = wy1 - c*wx1
                a = -1/c
                b = y - a*x
                e = wy2 - a*wx2

                if (wx2 - wx1)*a*(a*x + e - y) < 0:
                    # craft changed from waypoint a to waypoint b, but hasnt passed
                    # waypoint a yet, so error will be relative to line that goes to waypoint a
                    # waypoint a is w, waypoint b is w_next, and waypoint before a is w_before
                    xi = (b - d)/(c - a)
```

```
                    yi = c*xi + d

                    positive_error = True if y > yi else False
                else:
                    not_between_waypoints = True

            return (xi, yi, positive_error, not_between_waypoints)

        else:
            if (wy1 - wy2) == 0:
                xi = x
                yi = wy2

                positive_error = True if y > wy1 else False

            elif (wx1 - wx2) == 0:
                xi = wx2
                yi = y

                positive_error = True if x < wx1 else False
            else:
                c = (wy1 - wy2)/(wx1 - wx2)
                d = wy1 - c*wx1
                a = -1/c
                b = y - a*x
                # craft changed from waypoint a to waypoint b, but hasnt passed
                # waypoint a yet, so error will be relative to line that goes to waypoint a
                # waypoint a is w, waypoint b is w_next, and waypoint before a is w_before
                xi = (b - d)/(c - a)
                yi = c*xi + d

                positive_error = True if y > yi else False

            return (xi, yi, positive_error, None)

    def los(self, xf):
        if not self.no_more_waypoints:
            if self.reached_next_waypoint(xf, self.R_ACCEPTANCE) or self.missed_waypoint(xf):
                if self.current_waypoint == self.num_waypoints - 1:
                    self.no_more_waypoints = True
                else:
                    self.current_waypoint += 1
                    self.get_logger().info('changed waypoint at time: %f' % xf.time)
        else:
            self.finished = True if self.reached_next_waypoint(xf, self.R_ACCEPTANCE_FINAL) else False

        idx = self.current_waypoint
        x, y, theta = xf.position.x, xf.position.y, xf.position.theta
        u, v = xf.velocity.u, xf.velocity.v
        U = (u**2 + v**2)**0.5
        wx_next, wy_next, wv_next = self.waypoints.position.x[idx], self.waypoints.position.y[idx], self.wayp
        wx, wy = self.waypoints.position.x[idx-1], self.waypoints.position.y[idx-1]

        if not self.finished:
            # Find x_los and y_los by solving 2 eq.
            # Analytic solution:
            # 1. isolating x_los
```

```
            # ((wy - wy_past)/(wx - wx_past))*(x_los - wx) == (y_los - wy)
            # x_los == ((y_los - wy) + wx*((wy - wy_past)/(wx - wx_past)))/((wy - wy_past)/(wx - wx_past))
            # 2.  substitute and solve for y_los
            # (x_los-wx)**2 + (y_los-wy)**2 == self.R**2
            # 3.  get x_los
            # x_los == ((y_los - wy) + wx*((wy - wy_past)/(wx - wx_past)))/((wy - wy_past)/(wx - wx_past))

            x_los, y_los = self.get_xy_los(x, y, wx, wy, wx_next, wy_next)
            beta = math.asin(v/U)
            chi_d = math.atan2(x_los - x, y_los - y)
            psi_d = chi_d + beta

            # theta is how pydyna_simple measures yaw (starting from west, spanning [0,2pi])
            desired_value = 1.57079632679 - psi_d # psi to theta (radians)
            # format to positive angles
            if desired_value < 0:
                desired_value = 6.28318530718 + desired_value

            self.des_yaw_msg.desired_value = desired_value
            self.des_velocity_msg.desired_value = wv_next

            distance_waypoints = ((wx_next - wx)**2 + (wy_next - wy)**2)**0.5
            self.get_logger().info('wx_next: %f' % wx_next)
            self.get_logger().info('wx: %f' % wx)
            self.get_logger().info('distance_waypoints: %f' % distance_waypoints)
            self.des_yaw_msg.distance_waypoints = distance_waypoints
            self.des_velocity_msg.distance_waypoints = distance_waypoints
        else:
            # Will shutdown all nodes when reached final waypoint
            self.publisher_shutdown.publish(self.shutdown_msg)

        # norm of vector from craft location to path, making 90 degrees with path line
        self.append_current_errors(x, y, theta, wx, wy, wx_next, wy_next)

        self.desired_values_history['values'][0].append(self.des_velocity_msg.desired_value)
        self.desired_values_history['values'][1].append(self.des_yaw_msg.desired_value)

        self.desired_values_history['time'].append(xf.time)

        self.des_velocity_msg.current_waypoint, self.des_yaw_msg.current_waypoint = idx, idx

        return (self.des_velocity_msg, self.des_yaw_msg)

    def generate_plots(self):
        #clean before
        files = glob.glob(os.path.join(self.plots_dir, 'setpoints', '*.png'))
        for f in files:
            os.remove(f)

        params = {'mathtext.default': 'regular'}
        plt.rcParams.update(params)

        t = self.desired_values_history['time']

        ss_dir = "setpoints"
        desired_values_props = [
            {
```

```
            "title": "Linear Veloicity U Setpoint",
            "ylabel": r"$u_{des}\;[m/s]$",
            "file": "linearvelocityUSetpoint.png"
        },
        {
            "title": "Angular Position Theta Setpoint",
            "ylabel": r"$\theta_{des}\;[rad]$",
            "file": "angularpositionThetaSetpoint.png"
        },
    ]

    for i in range(len(self.desired_values_history['values'])):
        fig, ax = plt.subplots(1)
        ax.set_title(desired_values_props[i]["title"])
        ax.plot(t, self.desired_values_history['values'][i])
        ax.set_xlabel(r"$t\;[s]$")
        ax.set_ylabel(desired_values_props[i]["ylabel"])
        # ax.set_ylim(min(self.desired_values_history['values'][i]), max(self.desired_values_history['valu

        fig.savefig(os.path.join(self.plots_dir, ss_dir, desired_values_props[i]["file"]))


    # Program may be interrupted when self.path_error was already updated (appended value)
    # but t was not
    if len(self.path_error) > len(t):
        self.path_error = self.path_error[:-1]

    if len(self.width_error) > len(t):
        self.width_error = self.width_error[:-1]

    # clean before
    files = glob.glob(os.path.join(self.plots_dir, 'errors', 'error*.png'))
    for f in files:
        os.remove(f)

    fig, ax = plt.subplots(1)
    ax.set_title("Path error")
    ax.plot(t, self.path_error)
    ax.set_xlabel("t [s]")
    ax.set_ylabel("path error [m]")
    # ax.set_ylim(min(self.path_error), max(self.path_error))
    fig.savefig(os.path.join(self.plots_dir, "errors", "errorPath.png"))

    fig, ax = plt.subplots(1)
    ax.set_title("Width error")
    ax.plot(t, self.width_error)
    ax.set_xlabel("t [s]")
    ax.set_ylabel("width error [m]")
    # ax.set_ylim(min(self.width_error), max(self.width_error))
    fig.savefig(os.path.join(self.plots_dir, "errors", "errorWidth.png"))

    ######## report plots

    # clean before
    files = glob.glob(os.path.join(self.plots_dir, "reportPlots", "losGuidance", '*.png'))
    for f in files:
        os.remove(f)
```

```
        fig, ax = plt.subplots(1)
        ax.set_title("Errors")
        ax.plot(t, self.path_error)
        ax.plot(t, self.width_error)
        ax.set_xlabel("t [s]")
        ax.set_ylabel("error [m]")
        ax.legend([r"$cross-track$ error", r"$width$ error"])
        fig.savefig(os.path.join(self.plots_dir, "reportPlots", "losGuidance", "errors.png"))

    def print_metrics(self):
        mean_path_error = np.mean(np.abs(self.path_error))
        print('Mean path error: ', mean_path_error)
        self.get_logger().info('Mean path error: %f' % mean_path_error)

        max_path_error = np.max(np.abs(self.path_error))
        print('Max path error: ', max_path_error)
        self.get_logger().info('Max path error: %f' % max_path_error)

        mean_width_error = np.mean(np.abs(self.width_error))
        print('Mean width error: ', mean_width_error)
        self.get_logger().info('Mean width error: %f' % mean_width_error)

        max_width_error = np.max(np.abs(self.width_error))
        print('Max width error: ', max_width_error)
        self.get_logger().info('Max width error: %f' % max_width_error)

    def log_state(self, msg):
        self.get_logger().info(
            'listened filtered state: {position: {x: %f, y: %f, theta: %f}, velocity: {u: %f, v: %f, r: %f}, time:
            % (
                msg.position.x,
                msg.position.y,
                msg.position.theta, # yaw angle
                msg.velocity.u,
                msg.velocity.v,
                msg.velocity.r,
                msg.time
            )
        )

def main(args=None):
    try:
        rclpy.init(args=args)
        los_guidance_node = LosGuidance()
        rclpy.spin(los_guidance_node)
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
        los_guidance_node.get_logger().info('Stopped with user interrupt')
    except SystemExit:
        if los_guidance_node.finished:
            print('Finished path')
            los_guidance_node.get_logger().info('Finished path')
        else:
            print('Stopped with user shutdown request')
            los_guidance_node.get_logger().info('Stopped with user shutdown request')
    except:
```

```
        print(traceback.format_exc())
    finally:
        los_guidance_node.print_metrics()
        los_guidance_node.generate_plots()
        los_guidance_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

E.5 surge_controller.py

```
import sys
import os
import glob
import traceback
import math

import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import fsolve

import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32
from std_msgs.msg import Bool
# custom interface
from path_following_interfaces.msg import State, Control
from path_following_interfaces.srv import InitValues

class SurgeController(Node):
    def __init__(self):
        super().__init__('surge_controller_node')

        self.declare_parameter('plots_dir', './')
        self.plots_dir = self.get_parameter('plots_dir').get_parameter_value().string_value

        self.TIME_STEP = 0.1

        self.X_ADDED_MASS = -3375
        self.M = 40415

        self.KF_CONSTANT = 13

        self.thrust_history = []

        # self.phi_slope_tuning_factor = 10**-10 # without waves: 10**-10
        #self.phi_offset_tuning_factor = -0.13 # -0.13 without waves: -0.13
        self.phi = 0.19 # last 0.19

        self.kf_constant_tuning_factor = 12 # 8 without waves: 3.4

        self.est_time_correct_tuning_factor = 0.7
```

```
        self.server_init_control = self.create_service(
            InitValues, '/init_surge_control', self.callback_init_control
        )

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_filtered_state = self.create_subscription(
            State,
            '/filtered_state',
            self.callback_filtered_state,
            1)

        self.subscription_desired_surge_velocity = self.create_subscription(
            Control,
            '/desired_surge_velocity',
            self.callback_desired_surge_velocity,
            1)

        self.publisher_propeller_thrust = self.create_publisher(
            Float32,
            '/propeller_thrust',
            1)

        self.thrust_msg = Float32()

    def callback_shutdown(self, _):
        sys.exit()

    def callback_init_control(self, req, res):
        self.waypoints = req.waypoints
        self.initial_state = req.initial_state

        # format waypoints
        self.waypoints.position.x.insert(0, self.initial_state.position.x)
        self.waypoints.position.y.insert(0, self.initial_state.position.y)
        self.waypoints.velocity.insert(0, self.initial_state.velocity.u)

        self.current_waypoint = 1

        self.last_waypoint_yaw_angle = self.initial_state.position.theta
        self.last_waypoint_surge_velocity = self.initial_state.velocity.u
        self.last_waypoint_sway_velocity = self.initial_state.velocity.v

        self.last_xu_d = self.waypoints.velocity[1]

        self.get_steady_state_yaw_angles(self.waypoints)

        self.desired_surge_velocity = req.surge
        self.desired_surge_velocity_old = self.initial_state.velocity.u
        self.distance_waypoints = (
            (self.waypoints.position.x[1] - self.initial_state.position.x)**2 +
            (self.waypoints.position.y[1] - self.initial_state.position.y)**2
        )**0.5
```

```
        self.get_logger().info('initial distance_waypoints: %f' % self.distance_waypoints)
        thrust_msg = self.surge_control(self.initial_state)
        res.surge = thrust_msg.data
        return res

    def callback_filtered_state(self, msg):
        self.get_logger().info('listened filtered surge velocity: %f' % msg.velocity.u)
        self.publisher_propeller_thrust.publish(self.thrust_msg)
        self.thrust_history.append(self.thrust_msg.data)
        self.surge_control(msg)
        self.get_logger().info('published thrust force: %f' % self.thrust_msg.data)

    def callback_desired_surge_velocity(self, msg):
        self.get_logger().info('listened desired surge velocity: %f' % msg.desired_value)
        if self.desired_surge_velocity != msg.desired_value:
            self.desired_surge_velocity_old = self.desired_surge_velocity
            self.desired_surge_velocity = msg.desired_value

        self.distance_waypoints = msg.distance_waypoints
        self.get_logger().info('updated distance_waypoints: %f' % self.distance_waypoints)

        self.current_waypoint = msg.current_waypoint

    def surge_control(self, xf):
        # xu is surge velocity
        xu = xf.velocity.u
        xu_d = self.desired_surge_velocity
        self.get_logger().info('xu_d: %f' % xu_d)
        xu_dold = self.desired_surge_velocity_old
        self.get_logger().info('xu_dold: %f' % xu_dold)

        if xu_d != self.last_xu_d:
            self.last_waypoint_surge_velocity = xu
            self.last_waypoint_yaw_angle = xf.position.theta
            self.last_waypoint_sway_velocity = xf.velocity.v

            self.last_xu_d = xu_d

        # sliping variable
        s = xu - xu_d
        self.get_logger().info('s: %f' % s)
        # distace between waypoints
        distance = self.distance_waypoints
        self.get_logger().info('distance_waypoints: %f' % self.distance_waypoints)
        # estimated time to get from the old waypoint to the next
        # solution of following equation
        # distance(t) = integral of velocity(t) from 0 to est_time (xway_dold*est_time + (xu_d - xway_dold)*
        # velocity(t) = xway_dold + (xu_d - xway_dold)*(1 - exp(-t*k))
        # distance(t) = (xway_dold*est_time + (xu_d - xway_dold)*(est_time + (1/k)*exp(-est_time*k)) -(1/k)*
        kf = self.kf_constant_tuning_factor*(self.KF_CONSTANT)
        est_time = self.get_est_time(distance, kf, self.last_waypoint_surge_velocity, xu_d)
        self.get_logger().info('est_time_corrected: %f' % est_time)
        F = kf*5.18*(10**-5)
        eta = abs(self.last_waypoint_surge_velocity-xu_d)/est_time
        k = F + eta
        self.get_logger().info('k: %f' % k)
        # 0.0106734 is the baseline k (kf=13, from 0 to 5m/s in 500s) from my surge control project
```

```
        # 0.32 is the baseline phi from my surge control project
        #phi = self.phi_slope_tuning_factor*k + (0.32 - self.phi_slope_tuning_factor*0.0106734) + self.phi_offset_
        self.get_logger().info('phi: %f' % self.phi)
        # sat function
        sats = max(-1, min(s/self.phi, 1))
        # sats = np.sign(s) # for tuning only
        self.get_logger().info('sats: %f' % sats)
        # input as function of x (control action)
        f_hatp = xu*abs(xu)*1.9091*(10**-4)
        u = f_hatp - k*sats
        self.get_logger().info('u: %f' % u)
        # thrust
        tau = u*(self.M - self.X_ADDED_MASS)
        self.thrust_msg.data = tau

        return self.thrust_msg

    def get_steady_state_yaw_angles(self, waypoints):
        self.desired_steady_state_yaw_angles = []
        for i in range(1, len(waypoints.velocity)):
            desired_steady_state_yaw_angle = math.atan2(
                waypoints.position.y[i]-waypoints.position.y[i-1],
                waypoints.position.x[i]-waypoints.position.x[i-1]
            )
            self.desired_steady_state_yaw_angles.append(desired_steady_state_yaw_angle)

    def get_est_time(self, distance, kf, initial_velocity, final_velocity):
        data = (distance, kf, initial_velocity, final_velocity)
        # ponderates between linear and exponential response
        # when phi is higher goes from linear to exponential
        est_time_exp = fsolve(self.func, (2*final_velocity + initial_velocity)/3, args=data)[0]
        self.get_logger().info('est_time_exp: %f' % est_time_exp)
        est_time_lin = distance/((initial_velocity+final_velocity)/2)
        self.get_logger().info('est_time_lin: %f' % est_time_lin)

        try:
            est_time = (
                (((final_velocity-initial_velocity) - self.phi)*est_time_lin + self.phi*est_time_exp)
                /(final_velocity-initial_velocity)
            )
        except OverflowError: # est_time = est_time_lin
            est_time = est_time_lin

        self.get_logger().info('est_time: %f' % est_time)

        steady_state_yaw_angle = self.desired_steady_state_yaw_angles[self.current_waypoint-1]
        self.get_logger().info('steady_state_yaw_angle: %f' % steady_state_yaw_angle)
        u_ss = final_velocity
        theta_change_basis = (self.last_waypoint_yaw_angle - steady_state_yaw_angle)
        v_ss = (
            np.cos(np.pi/2 - theta_change_basis)*self.last_waypoint_surge_velocity
            + np.cos(theta_change_basis)*self.last_waypoint_sway_velocity
        )
        self.get_logger().info('v_ss: %f' % v_ss)

        # vss is used as velocity perpendicular to path(of the actual waypoint) at last waypoint
        # uss is used as final surge velocity
```

```
        beta = math.asin(v_ss/(v_ss**2 + u_ss**2)**0.5)
        self.get_logger().info('beta: %f' % beta)
        craft_steady_state_yaw_angle = steady_state_yaw_angle - beta
        self.get_logger().info('craft_steady_state_yaw_angle: %f' % craft_steady_state_yaw_angle)
        # used the results obtained from when there wasnt estimated time correction as baseline for
        # self.est_time_correct_tuning_factor
        # at th begginging the estimated timme was 347 and the actual time was 275 to reach the waypoint
        # with initial surge velocity = 1 and initial yaw angle = 90 for linear waypoints
        # 347 =  self.est_time_correct_tuning_factor*1.125*1*347
        self.get_logger().info('last_waypoint_yaw_angle: %f' % self.last_waypoint_yaw_angle)
        self.get_logger().info('last_waypoint_surge_velocity: %f' % self.last_waypoint_surge_velocity)

        abs_angle_dif = self.last_waypoint_yaw_angle - craft_steady_state_yaw_angle
        if abs_angle_dif > 3.14159265359:
            abs_angle_dif = abs(self.last_waypoint_yaw_angle - (6.28318530718 + craft_steady_state_yaw_angle)

        angle_dif = self.last_waypoint_yaw_angle - craft_steady_state_yaw_angle
        self.get_logger().info('angle_dif: %f' % angle_dif)
        if angle_dif > np.pi:
            angle_dif = -(np.pi - (self.last_waypoint_yaw_angle%np.pi-craft_steady_state_yaw_angle%np.pi))
        elif angle_dif < - np.pi:
            angle_dif = np.pi + (self.last_waypoint_yaw_angle%np.pi-craft_steady_state_yaw_angle%np.pi)

        # est_time_corrected = est_time
        # est_time_corrected = self.est_time_correct_tuning_factor*(abs(angle_dif)/2*np.pi + 1)*self.last_way
        # est_time_corrected = self.est_time_correct_tuning_factor*(abs(angle_dif)/2*np.pi + 1)*est_time
        est_time_corrected = self.est_time_correct_tuning_factor*(abs(angle_dif)/2*np.pi + 1)*est_time*(1/abs

        return est_time_corrected

    def generate_plots(self):
        #clean before
        files = glob.glob(os.path.join(self.plots_dir, 'thrustForce*.png'))
        for f in files:
            os.remove(f)

        params = {'mathtext.default': 'regular'}
        plt.rcParams.update(params)

        t = self.TIME_STEP*np.array(range(len(self.thrust_history)))
        fig, ax = plt.subplots(1)
        ax.set_title("Thrust force")
        ax.plot(t, self.thrust_history)
        ax.set_xlabel("t [s]")
        ax.set_ylabel(r"$\tau_1\;[N]$")
        # ax.set_ylim(min(self.thrust_history), max(self.thrust_history))

        graphics_file = "thrustForce.png"
        fig.savefig(os.path.join(self.plots_dir, graphics_file))

    @staticmethod
    def func(t, *data):
        distance, kf, initial_velocity, final_velocity = data
        return ( # expression == 0
            - distance + (initial_velocity*t + (final_velocity - initial_velocity)
            *(t + (1/(kf*5.18*(10**-5) + (abs(initial_velocity-final_velocity)/t)))
            *np.exp(-t*(kf*5.18*(10**-5) + (abs(initial_velocity-final_velocity)/t))))
```

```
            - (1/(kf*5.18*(10**-5) + (abs(initial_velocity-final_velocity)/t)))
            *(final_velocity - initial_velocity))
        )


def main(args=None):
    try:
        rclpy.init(args=args)
        surge_controller_node = SurgeController()
        rclpy.spin(surge_controller_node)
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
        surge_controller_node.get_logger().info('Stopped with user interrupt')
    except SystemExit:
        pass
    except:
        print(traceback.format_exc())
    finally:
        surge_controller_node.generate_plots()
        surge_controller_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()


# GRAVEYARD

# - Iterating to get est time (for when phi is not a constant)
# est_time = (est_time_lin + est_time_exp)/2
        # but the claculation depends on knowing phi, which depend on k which depends on est_time itself
        # so, begins with a guess for est_time and runs 5 iterations
        # for _ in range(5):
        #     k = (kf*5.18*(10**-5) + (abs(initial_velocity-final_velocity)/(est_time)))
        #     phi_bootstrap = self.phi_slope_tuning_factor*k + (0.32 - self.phi_slope_tuning_factor*0.0106734) + s
        #     est_time = (((final_velocity-initial_velocity) - phi_bootstrap)*est_time_lin + phi_bootstrap*est_tim
```

E.6 venus.py

```
import sys
import traceback
import math

import venus.viewer
from venus.objects import (
    GeoPos,
    Rudder,
    Vessel,
    Beacon,
    Size,
    KeyValue,
    Line
)

import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32
from std_msgs.msg import Bool
# custom interface
from path_following_interfaces.msg import Waypoints, State

class Venus(Node):
    def __init__(self):
        super().__init__('venus_node')

        # could receive initial propeller rotation from init_simul msg
        # but not urgent now, can be done later
        self.propeller_rotation = 0

        self.venus_init()

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_state = self.create_subscription(
            State,
            '/state',
            self.callback_state,
            1)

        self.subscription_propeller = self.create_subscription(
            Float32,
            '/propeller_rotation',
            self.callback_propeller_rotation,
            1)

        self.subscription_rudder = self.create_subscription(
            Float32,
            '/rudder_angle',
            self.callback_rudder_angle,
```

```
            1)

        self.subscription_waypoints = self.create_subscription(
            Waypoints,
            '/waypoints',
            self.callback_waypoints,
            1)

    def callback_shutdown(self, _):
        sys.exit()


    def venus_init(self):
        # GET MAPQUEST API KEY
        self.viewer = venus.viewer.Venus(mapquest_key = "1bZQGGHqFLQBezmB29WKAHTJKBXM0wDl", logging=True, port=615(
        # self.initial_position = GeoPos(-23.06255, -44.2772) # angra dos reis
        self.initial_position = GeoPos(-23.992557, -46.318) # santos
        self.viewer.set_viewport(self.initial_position, 15)
        vessel_config = Vessel(
            position = self.initial_position,
            angle = 0,
            size = Size(32.2, 186.4),
            # could receive initial rudder angle from init_simul msg
            # but not urgent now, can be done later
            rudders=[Rudder(angle=0, length=0.1, visual_options={"color": "orange"})],
            visual_options={
                "stroke": True,
                "color": "green",  # stroke color
                "weight": 3,  # stroke weight
                "opacity": 1.0,  # stroke opacity
                "lineCap": "round",
                "lineJoin": "round",
                "dashArray": None,
                "dashOffset": None,
                "fill": True,
                "fillColor": "red",
                "fillOpacity": 0.2,
                "fillRule": "evenodd",
            },
            data_panel= [
                KeyValue("ID", "104"),
                KeyValue("Width", "32 m"),
                KeyValue("Height", "186 m"),
                KeyValue("Linear Position X", "Waiting simul init"),
                KeyValue("Linear Position Y", "Waiting simul init"),
                KeyValue("Angular Position Theta", "Waiting simul init"),
                KeyValue("Linear Velocity U", "Waiting simul init"),
                KeyValue("Linear Velocity V", "Waiting simul init"),
                KeyValue("Angular Velocity R", "Waiting simul init"),
                KeyValue("Time", "Waiting simul init"),
                KeyValue("Rudder angle", "Waiting simul init"),
                KeyValue("Propeller rotation", "Waiting simul init")
            ]
        )
        self.vessel = self.viewer.add(vessel_config)
        self.viewer.on_object_drag_end = self.on_object_drag_end
```

```
    def callback_state(self, msg):
        state = msg
        self.get_logger().info(
            'listened state: {position: {x: %f, y: %f, theta: %f}, velocity: {u: %f, v: %f, r: %f}, time: %f
            % (
                state.position.x,
                state.position.y,
                state.position.theta, # yaw angle
                state.velocity.u,
                state.velocity.v,
                state.velocity.r,
                state.time
            )
        )
        #sleep(0.05)
        self.vessel.position = self.initial_position.relative(state.position.x, state.position.y)
        # measured from north clockwise (normal convention)
        self.vessel.angle = 90 - math.degrees(state.position.theta) # theta to psi

        # slice indexing was throwing error
        self.vessel.data_panel[3] = KeyValue("Linear Position X", str(round(state.position.x, 2)) + " m")
        self.vessel.data_panel[4] = KeyValue("Linear Position Y", str(round(state.position.y, 2)) + " m")
        self.vessel.data_panel[5] = KeyValue("Angular Position Theta", str(round(state.position.theta, 2)) +
        self.vessel.data_panel[6] = KeyValue("Linear Velocity U", str(round(state.velocity.u, 2)) + " m/s")
        self.vessel.data_panel[7] = KeyValue("Linear Velocity V", str(round(state.velocity.v, 2)) + " m/s")
        self.vessel.data_panel[8] = KeyValue("Angular Velocity R", str(round(state.velocity.r, 4)) + " rad/s
        self.vessel.data_panel[9] = KeyValue("Time", str(round(state.time, 2)) + " s")
        self.vessel.data_panel[10] = KeyValue("Rudder angle", str(round(self.vessel.rudders[0].angle, 2)) + '
        self.vessel.data_panel[11] = KeyValue("Propeller rotation", str(round(self.propeller_rotation, 2)) +

    def callback_rudder_angle(self, msg):
        self.get_logger().info('listened rudder angle: %f' % msg.data)
        # measured from south clockwise (normal convention )
        self.vessel.rudders[0].angle = math.degrees(msg.data)

    def callback_propeller_rotation(self, msg):
        self.get_logger().info('listened propeller rotation: %f' % msg.data)
        self.propeller_rotation = msg.data

    def callback_waypoints(self, msg):
        # initial x,y,u
        msg.position.x.insert(0, 0)
        msg.position.y.insert(0, 0)
        msg.velocity.insert(0, 0) # FILLER (just to maintain same lenght of the lists)
        self.waypoints = msg # {position: {x: [...], y: [...]} velocity: [...]}

        num_waypoints = len(msg.position.x)
        self.get_logger().info('initial waypoint + listened %d waypoints' % (num_waypoints-1))

        # Draw in the map

        # lists inside: [object, GeoPos]
        # GeoPos is only here because there was a problem using
        # the object's position to crate/update lines (need to investigate)
        self.beacons = []
        self.lines = []
        for i in range(num_waypoints):
```

```
            wx, wy, wv = msg.position.x[i], msg.position.y[i], msg.velocity[i]
            self.get_logger().info('listened waypoint %d: %f %f %f' % (i, wx, wy, wv))

            if i == 0:
                background_color = "green"
                beacon_data_panel = [KeyValue("Initial Position", "")]
                draggable = False
            else:
                background_color = "red"
                beacon_data_panel = [
                    KeyValue("Waypoint", ""),
                    KeyValue("Position X", str(wx)),
                    KeyValue("Position Y",  str(wy)),
                    KeyValue("Velocity U", str(wv))
                ]
                draggable = True

            beacon = Beacon(
                position=self.initial_position.relative(wx, wy),
                visual_options={
                    "background-color": background_color,
                    "border-radius": "50%"
                },
                data_panel=beacon_data_panel,
                draggable=draggable
            )
            self.viewer.add(beacon)
            # read below commented section tounderstand why i am doing this
            self.beacons.append([beacon, self.initial_position.relative(wx, wy)])

        ######### <passing beacon position inside points does not work, dont know why> ######
        # for j in range(num_waypoints):
        #     if j != num_waypoints-1:
        #         print(self.beacons[j][0].position.latitude)
        #
        #         line = Line(
        #             points=[self.beacons[j][0].position, self.beacons[j+1][0].position],
        #             visual_options={"color": "yellow", "weight": 5}
        #         )
        #         self.viewer.add(line)
        #         self.lines.append(line)
        ######### <passing beacon position inside points does not work, dont know why/> ######

            if i != num_waypoints-1:
                wx_next, wy_next = msg.position.x[i+1], msg.position.y[i+1]
                line = Line(
                    points=[self.initial_position.relative(wx, wy), self.initial_position.relative(wx_next, wy_next
                    visual_options={"color": "yellow", "weight": 5}
                )
                self.viewer.add(line)
                self.lines.append(line)

    def on_object_drag_end(self, obj, new_position):
        num_beacons = len(self.beacons)
        beacons_objects = [el[0] for el in self.beacons]
        beacon_idx = beacons_objects.index(obj)
        if 0 < beacon_idx < num_beacons-1:
```

```
            line_before = self.lines[beacon_idx-1]
            line_after = self.lines[beacon_idx]

            new_line_before = Line(
                points=[self.beacons[beacon_idx-1][1], new_position],
                visual_options={"color": "yellow", "weight": 5},
                draggable=False
            )

            new_line_after = Line(
                points=[new_position, self.beacons[beacon_idx+1][1]],
                visual_options={"color": "yellow", "weight": 5},
                draggable=False
            )

            self.viewer.remove(line_before)
            self.viewer.remove(line_after)

            self.viewer.add(new_line_before)
            self.viewer.add(new_line_after)

            self.lines[beacon_idx-1] = new_line_before
            self.lines[beacon_idx] = new_line_after
        elif beacon_idx == 0:
            line_after = self.lines[beacon_idx]

            new_line_after = Line(
                points=[new_position, self.beacons[beacon_idx+1][1]],
                visual_options={"color": "yellow", "weight": 5},
                draggable=False
            )

            self.viewer.remove(line_after)
            self.viewer.add(new_line_after)
            self.lines[beacon_idx] = new_line_after
        else:
            line_before = self.lines[beacon_idx-1]

            new_line_before = Line(
                points=[self.beacons[beacon_idx-1][1], new_position],
                visual_options={"color": "yellow", "weight": 5},
                draggable=False
            )

            self.viewer.remove(line_before)
            self.viewer.add(new_line_before)
            self.lines[beacon_idx-1] = new_line_before

        self.beacons[beacon_idx][0].position = new_position
        self.beacons[beacon_idx][1] = new_position # just to pass inside lines
        # FILLERS: str(1) -> need to get relative coordinates back
        self.beacons[beacon_idx][0].data_panel[1] = KeyValue("Position X", str(1))
        self.beacons[beacon_idx][0].data_panel[2] = KeyValue("Position Y", str(1))

def main(args=None):
    try:
        rclpy.init(args=args)
```

```
        venus_node = Venus() # port 6150
        rclpy.spin(venus_node)
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
    except SystemExit:
        pass
    except:
        print(traceback.format_exc())
    finally:
        venus_node.viewer.stop()
        venus_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

E.7 wave_filter.py

```
import sys
import os
import glob
import traceback

import matplotlib.pyplot as plt
import numpy as np
from scipy import signal
from scipy.signal import sosfreqz

import rclpy
from rclpy.node import Node

from std_msgs.msg import Bool
# custom interface
from path_following_interfaces.msg import State

class WaveFilter(Node):
    def __init__(self):
        super().__init__('wave_filter_node')

        self.TIME_STEP = 0.1

        self.declare_parameter('plots_dir', './')
        self.plots_dir = self.get_parameter('plots_dir').get_parameter_value().string_value

        self.filtered_state_history = [[],[],[],[],[],[]]
        self.simulated_state_history = [[],[],[],[],[],[]]

        #considering wn = [0.4, 0.63, 1]
        #self.num = np.array([1, 2.842, 4.07, 3.277, 1.623, 0.4523, 0.0635])
        #self.den = np.array([1, 4.06, 6.685, 5.709, 2.667, 0.6461, 0.0635])

        # wave is at 0.083 Hz or 0.52124 rad/s which is inside the band, but in the edge
        # exact fossens frequencies

        #considering wn = [0.52124 - 0.23, 0.52124, 0.52124 + 0.37]
```

```
        # self.num = np.array([1, 2.385, 2.868, 1.892, 0.758, 0.1659, 0.0183])
        # self.den = np.array([1, 3.407, 4.655, 3.255, 1.228, 0.237, 0.0183])

        # Fossen notch
        num = np.array([1, 2.385, 2.868, 1.892, 0.758, 0.1659, 0.0183])
        den = np.array([1, 3.407, 4.655, 3.255, 1.228, 0.237, 0.0183])
        z_s, p_s, k_s = signal.tf2zpk(num, den)
        z_z, p_z, k_z = signal.bilinear_zpk(z_s, p_s, k_s, 10)
        self.sos_notch_fossen = signal.zpk2sos(z_z, p_z, k_z)
        self.zi_notch_fossen = signal.sosfilt_zi(self.sos_notch_fossen)

        self.sos_notch_butter = signal.butter(6, [0.046352285679, 0.14184525164], 'bandstop', fs=10, output='
        self.zi_notch_butter = signal.sosfilt_zi(self.sos_notch_butter)

        # systems pole = 1/time constant is proxy for systems bandwith
        # systems pole is 0.0001592 Hz (based on my papaer, 63% of step response)
        # pole (resultant) of butterworth filter must be >> 0.0001592
        self.sos_lowpass_butter =  signal.butter(6, 0.10625, fs=10, output='sos')
        self.zi_lowpass_butter = signal.sosfilt_zi(self.sos_lowpass_butter)

        self.xf_msg = State()

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_simulated_state = self.create_subscription(
            State,
            '/simulated_state',
            self.callback_simulated_state,
            1)

        self.publisher_filtered_state = self.create_publisher(
            State,
            '/filtered_state',
            1)

    def callback_shutdown(self, _):
        sys.exit()

    def callback_simulated_state(self, msg):
        self.log_state(msg, 'subscriber')

        self.simulated_state_history[0].append(msg.position.x)
        self.simulated_state_history[1].append(msg.position.y)
        self.simulated_state_history[2].append(msg.position.theta)
        self.simulated_state_history[3].append(msg.velocity.u)
        self.simulated_state_history[4].append(msg.velocity.v)
        self.simulated_state_history[5].append(msg.velocity.r)

        filtered_state_msg = self.state_filter(msg.time)
        self.publisher_filtered_state.publish(filtered_state_msg)
        self.log_state(filtered_state_msg, 'publisher')

    def state_filter(self, t):
```

```
        # filters entire state

        # Wave filters
        # Butterworth notch filter
        state_history_filtered = map(lambda sig: signal.sosfilt(self.sos_notch_butter, sig, zi=sig[0]*self.zi_notch
        # Fossen noth filter
        # state_history_filtered = map(lambda sig: signal.sosfilt(self.sos_notch_fossen, sig, zi=sig[0]*self.zi_no

        # Noise filter (low pass): remove high freq noise from white noise added by gps_imu_simul
        # comment line below when gps_imu_simul is not activated
        state_history_filtered = map(lambda sig: signal.sosfilt(self.sos_lowpass_butter, sig, zi=sig[0]*self.zi_low

        state_current_filtered = [sig[-1] for sig in state_history_filtered]

        self.xf_msg.position.x = state_current_filtered[0]
        self.xf_msg.position.y = state_current_filtered[1]
        self.xf_msg.position.theta = state_current_filtered[2]
        self.xf_msg.velocity.u = state_current_filtered[3]
        self.xf_msg.velocity.v = state_current_filtered[4]
        self.xf_msg.velocity.r = state_current_filtered[5]
        self.xf_msg.time = t

        self.filtered_state_history[0].append(self.xf_msg.position.x)
        self.filtered_state_history[1].append(self.xf_msg.position.y)
        self.filtered_state_history[2].append(self.xf_msg.position.theta)
        self.filtered_state_history[3].append(self.xf_msg.velocity.u)
        self.filtered_state_history[4].append(self.xf_msg.velocity.v)
        self.filtered_state_history[5].append(self.xf_msg.velocity.r)

        return self.xf_msg


    def log_state(self, state, communicator):
        log_str = 'listened simulated' if communicator == 'subscriber' else 'published filtered'
        self.get_logger().info(
            '%s state: {position: {x: %f, y: %f, theta: %f}, velocity: {u: %f, v: %f, r: %f}, time: %f}'
            % (
                log_str,
                state.position.x,
                state.position.y,
                state.position.theta, # yaw angle
                state.velocity.u,
                state.velocity.v,
                state.velocity.r,
                state.time
            )
        )

    def generate_plots(self):
        #clean before
        files = glob.glob(os.path.join(self.plots_dir, 'simulatedState', '*.png'))
        for f in files:
            os.remove(f)

        params = {'mathtext.default': 'regular'}
        plt.rcParams.update(params)
```

```
        t = self.TIME_STEP*np.array(range(len(self.filtered_state_history[0])))
        fs_dir = "filteredState"
        filtered_state_props = [
            {
                "title": "Filtered Linear Position X",
                "ylabel": "x [m]",
                "file": "filteredLinearPositionX.png"
            },
            {
                "title": "Filtered Linear Position Y",
                "ylabel": "y [m]",
                "file": "filteredLinearPositionY.png"
            },
            {
                "title": "Filtered Angular Position Theta",
                "ylabel": r"$\theta\;[rad\;(from\;east\;counterclockwise)]$",
                "file": "filteredAngularPositionTheta.png"
            },
            {
                "title": "Filtered Linear Velocity U",
                "ylabel": "u [m/s]",
                "file": "filteredLinearVelocityU.png"
            },
            {
                "title": "Filtered Linear Velocity V",
                "ylabel": "v [m/s (port)]",
                "file": "filteredLinearVelocityV.png"
            },
            {
                "title": "Filtered Angular Velocity R",
                "ylabel": "r [rad/s (counterclockwise)]",
                "file": "filteredAngularVelocityR.png"
            },
        ]

        for i in range(len(self.filtered_state_history)):
            fig, ax = plt.subplots(1)
            ax.set_title(filtered_state_props[i]["title"])
            ax.plot(t, self.filtered_state_history[i])
            ax.set_xlabel(r"$t\;[s]$")
            ax.set_ylabel(filtered_state_props[i]["ylabel"])
            # ax.set_ylim([min(self.filtered_state_history[i]), max(self.filtered_state_history[i])])
            fig.savefig(os.path.join(self.plots_dir, fs_dir, filtered_state_props[i]["file"]))

        bode_dir = "bodePlots"

        filters = [
            {
                'dtf': self.sos_notch_fossen,
                'title': 'Wave filter - Frequency response',
                'file': 'notchFilterFossenBodePlot.png'
            },
            {
                'dtf': self.sos_lowpass_butter,
                'title': 'Sensor noise filter - Frequency response',
                'file': 'lowPassFilterBodePlot.png'
            },
```

```
    {
        'dtf': self.sos_notch_butter,
        'title': 'Wave filter - Frequency response',
        'file': 'NotchFilterButterBodePlot.png'
    }
]

#clean before
files = glob.glob(os.path.join(self.plots_dir, bode_dir, '*.png'))
for f in files:
    os.remove(f)

for filter in filters:
    # Bode plot
    fig, ax = plt.subplots(2)
    fig.suptitle(filter['title'], fontsize=16)
    w, h = sosfreqz(filter['dtf'], worN=8000, fs=10)

    ## Gain
    axGain = ax[0]
    db = 20*np.log10(np.maximum(np.abs(h), 1e-5))
    axGain.semilogx(w, db)
    axGain.set_title('Gain')
    # axGain.set_ylim(min(db), max(db))
    axGain.axes.get_xaxis().set_visible(False)
    axGain.set_ylabel("Gain [dB]")

    ## Phase
    axPhase = ax[1]
    negative_phase = [(-phase - 180) if phase > 0 else phase for phase in np.rad2deg(np.angle(h))]
    axPhase.semilogx(w, negative_phase)
    axPhase.set_title('Phase')
    # axPhase.set_ylim(min(negative_phase), 0)
    axPhase.set_xlabel("Frequency [Hz]")
    axPhase.set_ylabel("Phase [deg]")

    fig.savefig(os.path.join(self.plots_dir, bode_dir, filter['file']))

############## report plots
files = glob.glob(os.path.join(self.plots_dir, "reportPlots", "waveFilter", '*.png'))
for f in files:
    os.remove(f)

# u simulated and u filtered together
fig, ax = plt.subplots(1)
ax.set_title("Linear Velocity U")
ax.plot(t, self.simulated_state_history[3])
ax.plot(t, self.filtered_state_history[3])
ax.set_xlabel(r"$t\;[s]$")
ax.set_ylabel("u [m/s]")
ax.legend([r"$u$ from sensor", r"$u$ filtered (notch and low-pass)"])
fig.savefig(os.path.join(self.plots_dir, "reportPlots", "waveFilter", "surgeSimulated&Filtered.png"))

# theta simulated and theta filtered together
fig, ax = plt.subplots(1)
ax.set_title(r"Angular Position $\theta$")
ax.plot(t, self.simulated_state_history[2])
```

```
        ax.plot(t, self.filtered_state_history[2])
        ax.set_xlabel("t [s]")
        ax.set_ylabel(r"$\theta\;[rad]$")
        ax.legend([r"$\theta$ from sensor", r"$\theta$ filtered (notch and low-pass)"])
        fig.savefig(os.path.join(self.plots_dir, "reportPlots", "waveFilter", "yawSimulated&Filtered.png"))


def main(args=None):
    try:
        rclpy.init(args=args)
        wave_filter_node = WaveFilter()
        rclpy.spin(wave_filter_node)
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
        wave_filter_node.get_logger().info('Stopped with user interrupt')
    except SystemExit:
        pass
    except:
        print(traceback.format_exc())
    finally:
        wave_filter_node.generate_plots()
        wave_filter_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

E.8 yaw_controller.py

```
import sys
import os
import glob
import traceback

import numpy as np

import matplotlib.pyplot as plt

import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32
from std_msgs.msg import Bool
# custom interface
from path_following_interfaces.msg import Control, State
from path_following_interfaces.srv import InitValues

class YawController(Node):
    def __init__(self):
        super().__init__('yaw_controller_node')

        self.declare_parameter('plots_dir', './')
        self.plots_dir = self.get_parameter('plots_dir').get_parameter_value().string_value

        self.RUDDER_SAT = 0.610865 # 35 degrees
```

```
        self.TIME_STEP = 0.1

        self.rudder_angle_history = []

        self.last_rudder_angle = 0

        self.Kp = 1.6 # best: 1.6
        self.Kd = 65 # best: 65
        self.Ki = 0.00075  # best: 0.00075 # what i used when tuning surge controller:  0.000075 (antiwindup way),
        self.t_current_desired_yaw_angle = 0.1
        self.t_last_desired_yaw_angle = 0
        # for the integral action (acumulates error)
        self.theta_bar_int = 0
        self.integration_range = 0.1

        self.server_init_control = self.create_service(
            InitValues, '/init_yaw_control', self.callback_init_control
        )

        self.subscription_shutdown = self.create_subscription(
            Bool,
            '/shutdown',
            self.callback_shutdown,
            1)

        self.subscription_filtered_state = self.create_subscription(
            State,
            '/filtered_state',
            self.callback_filtered_state,
            1)

        self.subscription_desired_yaw_angle = self.create_subscription(
            Control,
            '/desired_yaw_angle',
            self.callback_desired_yaw_angle,
            1)

        self.publisher_rudder_angle = self.create_publisher(
            Float32,
            '/rudder_angle',
            1)

        self.rudder_msg = Float32()

    def callback_shutdown(self, _):
        sys.exit()

    def callback_init_control(self, req, res):
        self.waypoints = req.waypoints

        # format waypoints
        self.waypoints.position.x.insert(0, req.initial_state.position.x)
        self.waypoints.position.y.insert(0, req.initial_state.position.y)
        self.waypoints.velocity.insert(0, req.initial_state.velocity.u)

        self.desired_yaw_angle = req.yaw
```

```
        self.desired_yaw_angle_old = req.initial_state.position.theta
        rudder_msg = self.yaw_control(req.initial_state.position.theta, req.initial_state.velocity.r)
        res.yaw = rudder_msg.data
        self.last_rudder_angle = rudder_msg.data
        return res

    def callback_filtered_state(self, msg):
        self.get_logger().info('listened filtered yaw angle: %f' % msg.position.theta)
        self.publisher_rudder_angle.publish(self.rudder_msg)
        self.rudder_angle_history.append(self.rudder_msg.data)
        self.t = msg.time
        self.get_logger().info('time: %f' % msg.time)
        self.yaw_control(msg.position.theta, msg.velocity.r)
        self.get_logger().info('published rudder angle: %f' % self.rudder_msg.data)

    def callback_desired_yaw_angle(self, msg):
        self.get_logger().info('listened desired yaw angle: %f' % msg.desired_value)

        self.t_last_desired_yaw_angle = self.t_current_desired_yaw_angle
        self.t_current_desired_yaw_angle = self.t

        # fixes async issues
        # sometimes receives 2 desired yaw angles in sequence, without receiving filtered yaw angle
        if self.t_last_desired_yaw_angle == self.t_current_desired_yaw_angle:
           self.t_current_desired_yaw_angle += self.TIME_STEP

        if self.desired_yaw_angle != msg.desired_value:
            self.desired_yaw_angle_old = self.desired_yaw_angle
            self.desired_yaw_angle = msg.desired_value

    def pid(self, theta_bar, theta_bar_dot, integrator=True):
        if integrator:
            self.get_logger().info('self.theta_bar_int: %f' % self.theta_bar_int)
            rudder_angle = -self.Kp*theta_bar - self.Kd*theta_bar_dot - self.Ki*self.theta_bar_int
        else:
            rudder_angle = -self.Kp*theta_bar - self.Kd*theta_bar_dot

        self.theta_bar_int = self.theta_bar_int + theta_bar*self.TIME_STEP
        return rudder_angle

    def yaw_control(self, theta, r):
        # desired theta
        theta_des = self.desired_yaw_angle
        # last desired theta
        theta_des_old = self.desired_yaw_angle_old
        # error
        theta_bar = theta - theta_des
        if theta_bar > np.pi:
            theta_bar = -(np.pi - (theta%np.pi-theta_des%np.pi))
        elif theta_bar < - np.pi:
            theta_bar = np.pi + (theta%np.pi-theta_des%np.pi)

        theta_bar_dot = r - (theta_des - theta_des_old)/ \
            (self.t_current_desired_yaw_angle - self.t_last_desired_yaw_angle)
        self.get_logger().info('theta_bar_dot: %f' % theta_bar_dot)

        # antiwindup stategy 1
```

```
        if abs(theta_bar) > self.integration_range:
            self.get_logger().info('### not using integrator because of integration range')
            rudder_angle =  self.pid(theta_bar, theta_bar_dot, integrator=False)
        else:
            self.get_logger().info('### normal pid using integrator')
            rudder_angle = self.pid(theta_bar, theta_bar_dot)

        # rudder saturation (with 1% safety margin)
        # real sat is 35 degress
        rudder_angle = max(-self.RUDDER_SAT*0.99, min(rudder_angle, self.RUDDER_SAT*0.99))
        self.rudder_msg.data = rudder_angle
        self.last_rudder_angle = rudder_angle

        return self.rudder_msg

    def generate_plots(self):
        #clean before
        files = glob.glob(os.path.join(self.plots_dir, 'rudderAngle*.png'))
        for f in files:
            os.remove(f)

        params = {'mathtext.default': 'regular'}
        plt.rcParams.update(params)

        t = self.TIME_STEP*np.array(range(len(self.rudder_angle_history)))
        fig, ax = plt.subplots(1)
        ax.set_title("Rudder angle")
        ax.plot(t, self.rudder_angle_history)
        ax.set_xlabel("t [s]")
        ax.set_ylabel(r"$\delta\;[rad\;(from south clockwise)]$")
        # ax.set_ylim(min(self.rudder_angle_history), max(self.rudder_angle_history))

        fig.savefig(os.path.join(self.plots_dir, "rudderAngle.png"))

def main(args=None):
    try:
        rclpy.init(args=args)
        yaw_controller_node = YawController()
        rclpy.spin(yaw_controller_node)
    except KeyboardInterrupt:
        print('Stopped with user interrupt')
        yaw_controller_node.get_logger().info('Stopped with user interrupt')
    except SystemExit:
        pass
    except:
        print(traceback.format_exc())
    finally:
        yaw_controller_node.generate_plots()
        yaw_controller_node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

# GRAVEYARD:
# -  for antiwindup strategy 2 (may be used in ther scenario)
    # [useful snippet in other situation] for antiwindup strategy 2 only:
```

```
    #     # cumulative of the error (integral action)
    #     theta_bar_int = self.theta_bar_int + theta_bar*self.TIME_STEP
    #     # self.theta_bar_int = max(-self.ANTIWINDUP, min(self.theta_bar_int,self.ANTIWINDUP))
    #     self.get_logger().info('self.theta_bar_int: %f' % theta_bar_int)
    #     # control action
    #     rudder_angle = -self.Kp*self.K_tuning_factor*theta_bar - self.Kd*theta_bar_dot - self.Ki *theta_bar

    #     return rudder_angle, theta_bar*self.TIME_STEP

    # [useful snippet in other situation] antiwindup stategy 2 -> wasnt very succesfull in this case, but is
    # if ( # saturated
    #     self.last_rudder_angle > self.RUDDER_SAT*0.99 or self.last_rudder_angle < -self.RUDDER_SAT*0.99
    # ):
    #     # verify if the current control would increase the abs(self.theta_bar_int)
    #     self.get_logger().info('### doing antiwindup experiment with integrator')
    #     if self.last_rudder_angle*self.pid_using_integrator(theta_bar, theta_bar_dot, experiment=True)[1] >
    #         # dont consider integral action
    #         self.get_logger().info('### not using antiwindup because would "saturate more"')
    #         rudder_angle = self.pid_not_using_integrator(theta_bar, theta_bar_dot)
    #     else:
    #         self.get_logger().info('### normal pid using integrator')
    #         rudder_angle = self.pid_using_integrator(theta_bar, theta_bar_dot)[0]
    # else:
    #     self.get_logger().info('### normal pid using integrator')
    #     rudder_angle = self.pid_using_integrator(theta_bar, theta_bar_dot)[0]

# - autotune controller
    # def tune_controller(self, waypoints, initial_state):
    #     cases = []
    #     self.desired_steady_state_yaw_angles = []
    #     for i in range(1, len(waypoints.velocity)):
    #         print(waypoints.position.x[i])
    #         print(waypoints.position.y[i])
    #         print(waypoints.position.x[i-1])
    #         print(waypoints.position.y[i-1])
    #         distance = (
    #             (waypoints.position.x[i] - waypoints.position.x[i-1])**2 +
    #             (waypoints.position.y[i] - waypoints.position.y[i-1])**2
    #         )**0.5
    #         desired_steady_state_yaw_angle = math.atan2(
    #             waypoints.position.y[i]-waypoints.position.y[i-1],
    #             waypoints.position.x[i]-waypoints.position.x[i-1]
    #         )

    #         self.desired_steady_state_yaw_angles.append(desired_steady_state_yaw_angle)
    #         if i != 1:
    #             self.get_logger().info(' i != 1:')
    #             last_desired_steady_state_yaw_angle = math.atan2(
    #                 waypoints.position.y[i-1]-waypoints.position.y[i-2],
    #                 waypoints.position.x[i-1]-waypoints.position.x[i-2]
    #             )
    #         else:
    #             self.get_logger().info('i == 1:')
    #             last_desired_steady_state_yaw_angle = initial_state.position.theta
    #         delta_yaw_angle = (
    #             abs(desired_steady_state_yaw_angle - last_desired_steady_state_yaw_angle)
    #         )
```

```
    #         try:
    #             case = delta_yaw_angle/distance
    #         except ZeroDivisionError:
    #             case = delta_yaw_angle
    #         cases.append(case)
    #     worst_case = max(cases)
    #     # 0.0011107205 = math.radians(90 - 45)/sqrt(500^2 + 500^2)
    #     auto_tuning_factor = worst_case/0.0011107205
    #     self.get_logger().info('yaw controller autotuning factor: %f' % auto_tuning_factor)
    #     self.K_tuning_factor = self.K_tuning_factor*auto_tuning_factor
```

E.9 setup.py

```
import os
from glob import glob
from setuptools import setup
from glob import glob

package_name = 'path_following'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'logs', 'mylogs'), []),
        (os.path.join('share', package_name, 'logs', 'pydynalogs'), []),
        (os.path.join('share', package_name, 'logs', 'roslogs'), []),
        (os.path.join('share', package_name, 'db', 'rosbags'), []),
        (os.path.join('share', package_name, 'db', 'waypoints'), []),
        (os.path.join('share', package_name, 'plots', 'state'), []),
        (os.path.join('share', package_name, 'plots', 'simulatedState'), []),
        (os.path.join('share', package_name, 'plots', 'filteredState'), []),
        (os.path.join('share', package_name, 'plots', 'setpoints'), []),
        (os.path.join('share', package_name, 'plots', 'bodePlots'), []),
        (os.path.join('share', package_name, 'plots', 'errors'), []),
        (os.path.join('share', package_name, 'plots', 'reportPlots'), []),
        (os.path.join('share', package_name, 'plots', 'reportPlots', 'waveFilter'), []),
        (os.path.join('share', package_name, 'plots', 'reportPlots', 'gpsImuSimul'), []),
        (os.path.join('share', package_name, 'plots', 'reportPlots', 'losGuidance'), []),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='bruno',
    maintainer_email='bruno.c.scaglione@gmail.com',
    description='Path-Following Ship',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'yaw_controller = path_following.yaw_controller:main',
```

```
            'surge_controller = path_following.surge_controller:main',
            'backend = path_following.backend:main',
            'control_allocation = path_following.control_allocation:main',
            'gps_imu_simul = path_following.gps_imu_simul:main',
            'venus = path_following.venus:main',
            'wave_filter = path_following.wave_filter:main',
            'los_guidance = path_following.los_guidance:main',
            'kalman_filter = path_following.kalman_filter:main',
        ],
    },
)
```

E.10 package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XI
<package format="3">
  <name>path_following</name>
  <version>0.0.0</version>
  <description>Path-Following Ship</description>
  <maintainer email="bruno.c.scaglione@gmail.com">bruno</maintainer>
  <license>Apache License 2.0</license>

  <exec_depend>os</exec_depend>
  <exec_depend>glob</exec_depend>
  <exec_depend>sys</exec_depend>
  <exec_depend>traceback</exec_depend>
  <exec_depend>datetime</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>flask</exec_depend>
  <exec_depend>json</exec_depend>
  <exec_depend>numpy</exec_depend>
  <exec_depend>sympy</exec_depend>
  <exec_depend>scipy</exec_depend>
  <exec_depend>venus</exec_depend>

  <depend>path_following_interfaces</depend>
  <depend>rclpy</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>


  <export>
    <build_type>ament_python</build_type>
  </export>
</package>

\clearpage
```

E.11 path_following.launch.py

```
import os

from ament_index_python.packages import get_package_share_directory, get_package_prefix
from launch import LaunchDescription
from launch.actions import ExecuteProcess
from launch_ros.actions import Node

# obs: not exaclty shure where generate_launch_description is invoked
# however absolute paths should do, with get_package_share_directory

def generate_launch_description():
    P3D_FILES = [
        'TankerL186B32_T085.p3d',
        'NoWaves_TankerL186B32_T085.p3d',
        'NoCurrent&Wind_TankerL186B32_T085.p3d',
        'NoWaves&Current&Wind_TankerL186B32_T085.p3d'
    ]

    pkg_share_dir = get_package_share_directory('path_following')
    pkg_install_dir = get_package_prefix('path_following')
    pkg_dir = os.path.join(pkg_install_dir, 'lib', 'pydyna_simple')
    logs_dir = os.path.join(pkg_share_dir, 'logs')
    plots_dir = os.path.join(pkg_share_dir, 'plots')
    p3d_file = P3D_FILES[0] # change p3d here
    db_dir = os.path.join(pkg_share_dir, 'db')

    os.environ['ROS_LOG_DIR'] = os.path.join(logs_dir, 'roslogs')
    # Set LOG format
    os.environ['RCUTILS_CONSOLE_OUTPUT_FORMAT'] = '[{severity} {time}] [{name}]: {message} ({function_name}() at {

    ld = LaunchDescription()

    rosbag_record_all = ExecuteProcess(
        cmd=['ros2', 'bag', 'record', '/state'], # use '-a' if want to record all
        output='screen'
    )

    start_pydyna_simple_node = Node(
        package='pydyna_simple',
        executable='simul',
        name='pydyna_simple_node',
        output='screen',
        parameters=[
                {'pkg_share_dir': pkg_share_dir},
                {'pkg_dir': pkg_dir},
                {'p3d_file': p3d_file}
        ]
    )

    start_los_guidance_node = Node(
        package='path_following',
        executable='los_guidance',
        name='los_guidance_node',
        output='screen',
        parameters=[
```

```
            {'plots_dir': plots_dir}
        ]
    )

    start_surge_controller_node = Node(
        package='path_following',
        executable='surge_controller',
        name='surge_controller_node',
        output='screen',
        parameters=[
            {'plots_dir': plots_dir}
        ]
    )

    start_yaw_controller_node = Node(
        package='path_following',
        executable='yaw_controller',
        name='yaw_controller_node',
        output='screen',
        parameters=[
            {'plots_dir': plots_dir}
        ]
    )

    start_control_allocation_node = Node(
        package='path_following',
        executable='control_allocation',
        name='control_allocation_node',
        output='screen',
        parameters=[
            {'plots_dir': plots_dir}
        ]
    )

    start_gps_imu_simul_node = Node(
        package='path_following',
        executable='gps_imu_simul',
        name='gps_imu_simul_node',
        output='screen',
        parameters=[
            {'plots_dir': plots_dir}
        ]
    )

    start_wave_filter_node = Node(
        package='path_following',
        executable='wave_filter',
        name='wave_filter_node',
        output='screen',
        parameters=[
            {'plots_dir': plots_dir}
        ]
    )

    start_venus_node = Node(
        package='path_following',
        executable='venus',
```

```
        name='venus_node',
        output='screen'
    )

    start_backend_node = Node(
        package='path_following',
        executable='backend',
        name='backend_node',
        output='screen',
        parameters=[
                {'db_dir': db_dir}
        ]
    )

    ld.add_action(rosbag_record_all)
    ld.add_action(start_pydyna_simple_node)
    ld.add_action(start_los_guidance_node)
    ld.add_action(start_surge_controller_node)
    ld.add_action(start_yaw_controller_node)
    ld.add_action(start_control_allocation_node)
    ld.add_action(start_gps_imu_simul_node)
    ld.add_action(start_wave_filter_node)
    ld.add_action(start_venus_node)
    ld.add_action(start_backend_node)

    return ld
```

E.12 setup.cfg

```
[develop]
script_dir=$base/lib/path_following
[install]
install_scripts=$base/lib/path_following
```

APPENDIX

F

PATH_FOLLOWING_INTERFACES CODE

The files that compose the *path_following_interfaces* package can be found below.

F.1 Control.msg

```
float32 desired_value 0.0
float32 distance_waypoints 707.0
int32 current_waypoint 1
```

F.2 Position.msg

```
# positions (earth-fixed reference frame)
float32 x 0.0
float32 y 0.0
# 1.57079632679 radians = 90 degrees
float32 theta 1.57079632679
```

F.3 PositionsXY.msg

```
float32[] x [500.0, 1000.0, 1500.0, 2000.0, 2500.0]
float32[] y [500.0, 1000.0, 1500.0, 2000.0, 2500.0]
```

F.4 State.msg

```
# 3DOF state of the craft
Position position
Velocity velocity
float32 time 0.0
```

F.5 Velocity.msg

```
# velocities (craft-fixed reference frame)
float32 u 1.0
float32 v 0.0
float32 r 0.0
```

F.6 Waypoints.msg

```
PositionsXY position
float32[] velocity [3.0, 3.5, 4.0, 4.5, 5.0]
```

F.7 InitValues.srv

```
#request
State initial_state
Waypoints waypoints
float32 surge 0.0
float32 yaw 0.0
---
#response
float32 surge 0.0
float32 yaw 0.0
```

F.8 package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>path_following_interfaces</name>
  <version>0.0.0</version>
  <description>Interfaces for the path-following ship system</description>
  <maintainer email="bruno.c.scaglione@gmail.com">Bruno</maintainer>
  <license>Apache License 2.0</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <build_depend>rosidl_default_generators</build_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```