Implementation of control algorithms for an autonomous sailboat

Assistant Engineer Internship Report
FISE ROB 2024
April 2023 – August 2023

Harendra RANGARADJOU
ENSTA Bretagne - Autonomous Robotics
harendra.rangaradjou@ensta-bretagne.org

Supervisor:

Dr. Jian WAN
Aston University - College of Engineering and Physical Sciences
Mechanical, Biomedical & Design Engineering
j.wan3@aston.ac.uk

Acknowledgements

I would like to thank **Prof. Luc JAULIN** (referent professor) for having recommended me for this internship, as well as **Dr. Jian WAN** (internship tutor) for the assistance he provided throughout the entire project. I would also like to thank **Jean-Baptiste Souppez** (Senior Teaching Fellow at Aston University) for the words of advice he gave on useful naval architecture and navigation concepts for the project.

I would finally like to sincerely thanks **Catherine RIZK** and **Ludovic MUSTIÈRE**, who were both also assigned to this project. They worked wonders on their respective parts, and I would not have been able to do even a fraction of the work I got through without their precious help.

Abstract

As a part of my robotics engineering training at *ENSTA* Bretagne, I undertook a sixteen-week internship (from April 24–2023 to August 11–2023) in the Mechanical, Biomedical and Design Engineering Department of the College of Engineering and Physical Sciences at Aston University, Birmingham UK. During these four months, I performed a plethora of tasks revolving around the control of an autonomous sailboat, ranging from writing and testing sensor drivers to developing a dynamic simulation in python for the boat. The main mission I was assigned was to allow the boat to pass by every waypoint in a given list of arbitrary length.

In this report, the context is firstly established through a brief introduction on *ENSTA* Bretagne, Aston University and my motivations for undertaking this internship. The stakes and coherence of the latter as a part of my training are also discussed. Secondly, an in-depth presentation of the hardware and software used in the final version of the system is given, followed by a thorough review of the work done during the sixteen weeks which lead to the completion and testing of this final prototype. Thirdly, the results attained are analysed and discussed, along with the unfinished tasks and the potential improvements. Lastly, a summary of all the aforementioned key points is made and used to put my personal gains from this experience into perspective.

Résumé

Dans le cadre de ma formation d'ingénieur à l'*ENSTA* Bretagne, j'ai réalisé un stage de seize semaines (du 24 avril 2023 au 11 août 2023), dans le département d'Ingéniérie Mécanique, Biomédicale et de Conception du Collège d'Ingéniérie et de Sciences Physiques de l'Université d'Aston, à Birmingham, au Royaume-Uni. Pendant ces quatre mois, j'ai réalisé de nombreuses tâches ayant à trait au contrôle d'un voilier autonome, allant de l'écriture et du test de pilotes pour des capteurs, au développement d'une simulation dynamique sous python pour le bateau. La mission principale qui m'a été attribuée était de permettre au bateau de passer par toutes les balises fictives d'une liste donnée de taille arbitraire.

Ce rapport vise dans un premier temps à établir le contexte par une brève introduction sur l'*ENSTA* Bretagne, l'Université d'Aston et les motivations qui m'ont amenées à faire ce stage. Les enjeux et la cohérence de ce dernier dans le cadre de ma formation seront aussi traités. Dans un second temps, un portrait matériel et logiciel détaillé du système final est dressé, suivi d'un passage en revue minutieux du travail réalisé pendant les seizes semaines ayant conduit à la réalisation et aux test du prototype final. Dans un troisième temps, les résultats obtenus sont présentés et analysés, tout comme le travail restant et les pistes d'améliorations possibles. Enfin, un résumé de l'ensemble des points clés mentionnés ci-dessus est utilisé pour mettre en perspective les gains que j'ai pu tirer de cette expérience.

Keywords

Autonomous sailboat, Control algorithms, Kalman Filter, Python, Simulation, Hardware in the loop, Raspberry Pi 4, Navio2, Education, Internship

Contents

1 Introduction

1.1 Context

The robotics engineering training at *ENSTA* Bretagne is fundamentally multidisciplinary and covers a wide spectrum of application fields for autonomous mobile robotics. It relies on a balanced mix of theoretical and practical teachings, by combining regular theoretical classes with numerous practical team projects. These projects can range from programming an autonomous rover to working on a humanoid bipedal robots, as illustrated in Figure 1.

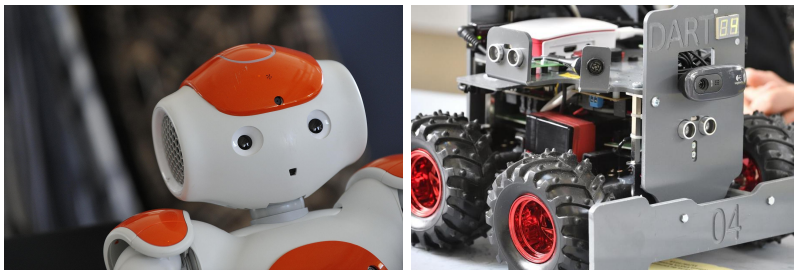


Figure 1: *Example of robots used in projects*

The school specialises in marine robotics. As a result, one of the major activities of the curriculum is the *DDboat* project, at the Guerlédan naval base. This project focuses on the implementation of control algorithms for model boats like the one seen in Figure 2. The goal for every group of students is to ensure, in just under a week, that their boat is capable of fulfilling a set of missions. These missions can range from simply making the boat follow a given heading for a given amount of time, to following a complex track, defined by *GNSS* waypoints, with an imposed deferred start from certain waypoints.

Figure 2: *A DDboat on the Guerlédan lake*

Despite specialising in marine robotics, sailboats are only briefly studied in the curriculum. This partially explains why my curiosity was peaked when I was presented with an opportunity to work on one, that and the great impression left by the *DDboat* project. The internship description mentioning machine learning also got me interested, as it is the field of robotics that I am the most interested in and one that we do not get to explore much until our last year of training. This is why I ended up choosing to go to Aston University to work on **Dr. WAN** 's autonomous sailboat project, along with **Catherine RIZK** and **Ludovic MUSTIÈRE**, two other students from *ENSTA* Bretagne.

Aston University, located in the city of Birmingham, United Kingdom, is a prestigious institution renowned for its commitment to excellence in education and research. Founded in 1966, Aston has since established itself as a leading university, ranking 446th in the 2024 QS World University Ranking, placing the University in the top 30% of global institutions, and 46th place nationally [1]. It offers a wide range of courses, particularly in business, engineering, and applied sciences [2] and supports cutting edge research activities in these same fields [3]. Research at Aston is spread across multiple Colleges and Schools [3]. The autonomous sailboat project I worked on, being supervised by **Dr. WAN**, is associated with the Mechanical, Biomedical and Design Engineering Department of the College of Engineering and Physical Sciences.

Figure 3: *Aston University, main building*

1.2 Goals and stakes

I had many goals with this project. First and foremost, it was my assistant engineer internship. As such, it had to allow me to confront my knowledge with the reality of a globalised research community and to put them in perspective in anticipation of my final year of training at *ENSTA* Bretagne
It ultimately achieved that goal by providing me with an opportunity to put my knowledge into practice in a foreign country with a different work culture than the one I had always been used to up until then.

Moreover, it is a part of my career project, similarly to the rest of my training, it had to allow me to broaden my knowledge of robotics either by working on an aspect of robotics which I had not touched upon yet or by significantly deepening my knowledge on an aspect I was already familiar with. The reason for this goal is the same that motivated my choice of a future career in robotics, *i.e.* to become as polyvalent of an engineer as possible.
It managed to meet this criterion by allowing me to manipulate a type of robot I had no prior experience with, that is *USV*s, and to experience all the difficulties that come with the design of such robots.

In addition, this project also has stakes for **Dr. WAN** who has been doing research on *USV*s and on multi-agent systems involving *USV*s and *UAV*s since 2016. Indeed, my internship was only a contribution to the autonomous sailboat project. For instance, the boat I worked on had already been modified the previous year to be able to complete the same objective I was tasked with ensuring it could achieve. In that sense, my goal was primarily to verify the reproducibility of the results obtained by the previous team of students.

This is probably the part I had the hardest time with, as much of the earlier work done on the boat was not readily reusable. This in turn led me to spend most my internship re-writing low-level code for the boat from scratch rather than being able to focus on the implementation of control algorithms, despite the latter being closest to the intended purpose of the platform. As a matter of fact, the autonomous sailboat project aims at developing an easy-to-build and easy-to-use open platform that could be used widely in education or in research. For example, it could be used as a means to teach students how to program *USV*s or as a quickly deployable platform to run basic tests of novel control algorithms. I like to think that I helped to get closer to this goal by making sure to produce readable, documented and ready-to-use code, that will undoubtedly help, if not future users, at least the next group of students that will play a part in this project.

2 Activities

2.1 System description

2.1.1 Hardware

The sailboat used is the "Raggaza™ 1-Meter Sailboat RTR", shown in Figure 4. It features a double rip-stop nylon sail connected *via* a winch, a single rudder, a fiberglass main hull, and a keel to improve the stability of the vessel.

Figure 4: *Raggaza™ 1-Meter Sailboat RTR*

Two positional high torque servomotors, shown in Figure 5, are used to control the sails and rudder separately. The Hitec HS-785HB is used to pilot the winch connected to the sails and the Hitec HS-5645MG is used to pilot the rudder. Operating under a 6V input voltage, they are capable of delivering a maximum torques 13.2 kg.cm and 12.1 kg.cm and reaching maximum speeds of $1.38s@60°$ and $0.18s@60°$ respectively [4, 5]

Figure 5: *Hitec HS-785HB (left) and Hitec HS-5645MG (right)*

Radio control is used to pilot the boat manually. The emitter used is a Hobbyking HK-T4A V2 2.4GHz and the receiver is a Hobbyking HK-TR6A V2 2.4GHz, both shown Figure 6. Communication between the transmitter and the Navio2 is established *via PPM* signals.

Figure 6: *Hobbyking HK-T4A transmitter and HK-TR6A V2 receiver*

The on-board computer is composed of a Raspberry Pi 4 Model B and a Navio2, shown in Figure 7. The Raspberry pi 4 is a highly performant and polyvalent microprocessor. It features a BCM2711 chip based on ARMv8 architecture and capable of top clock speeds of 1.5GHz [6]. The Navio2 board is a raspberry pi autopilot *HAT*. Compatible with ArduPilot and ROS, it features two *IMU*s, a *GNSS*, and a barometer. It can also handle *RC* inputs and *PWM* outputs [7].

Figure 7: *Raspberry Pi 4 Model B (left) and Navio2 (right)*

The observation process is enabled by multiple sensors, all shown in Figure 8. First, the state of the boat (see section *2.3.2. Model and simulation*) is determined using the Navio2 embedded sensors. The *GNSS*, a NEO M8N from ublox, is used to get the general position of the boat with a precision ranging from 2m to 4m, depending on the constellation used by the sensor [8]. The MPU9250 and LSM9DS1 *IMU*s, all-in-one accelerometer, gyro, and magnetometer, are used to provide the heading of the boat. Moreover, a Calypso ULTRASONIC Portable wind sensor is used to measure the wind direction and wind speed.

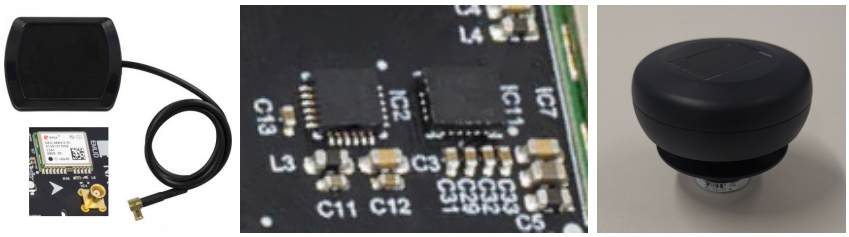


Figure 8: *NEO M8N (left), dual IMU (center), Calypso anemometer (right)*

The system was initially powered by a single 11.1V 3s 2200mAh or 14.8V 4s 1800mAh LiPo battery, combined with a *BEC* to power the servomotors and a step-down converter to power the onboard electronics. However, the use of separate batteries for the onboard computer and servomotors was seriously considered .

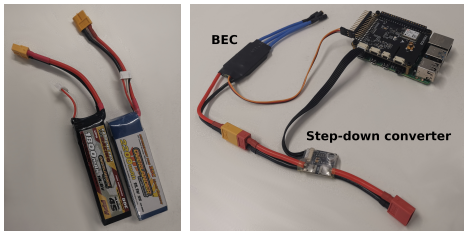


Figure 9: *LiPo batteries used (left), voltage regulation unit (right)*

An simple diagram of the hardware architecture of the system is presented in Figure 10 for the sake of clarity.

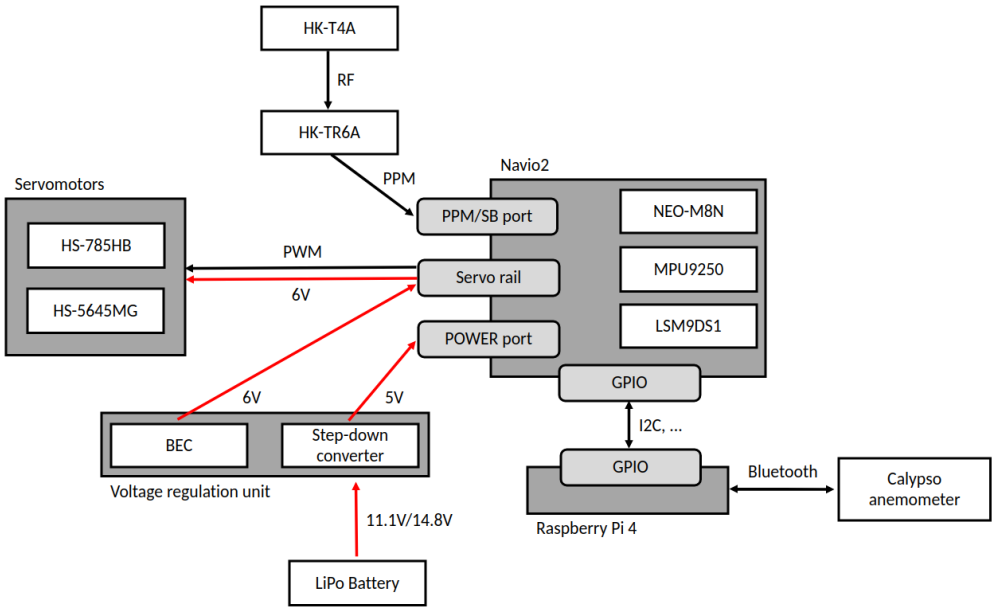


Figure 10: *Diagram of the on-board electronics*

2.1.2 Software

The OS used for the Raspberry Pi is a modified 32-bit version of the Raspberry Pi OS (formerly Raspbian), a debian based operating system maintained by Raspberry Pi Ltd. A modified system image containing significant adjustments to ensure full compatibility between the Raspberry Pi and Navio2 is provided by Emlid, the constructors of the Navio2 board. For instance, some changes were made to the *SPI* and *I2C* bus configurations, to enable the communication between the Raspberry Pi and the Navio2 sensors.

The main programs are all written using Python 3.11 with additional libraries listed in Table 1. A multithreading-based approach was used to design the architecture of the code. Bash and scripts were added to facilitate the launch of missions and to automate certain tasks like sensor calibration. These scripts are meant to be launched remotely on the on-board computer *via* a *SSH* connection. I used a *SSH* key pair to facilitate the connection process to the remote session hosted on the on-board computer.

Library	Version	Description
calypso-anemometer	0.6.0	Calypso anemometer driver and *API*
dill	0.3.7	Python's pickle module extension
matplotlib	3.7.2	Plotting library
navio2	1.0.0	Navio2 driver
numpy	1.25.1	Scientific computing library
pygubu	0.31	Tkinter-based *GUI* builder
PyProj	3.6.0	Cartographic projection library
PyYAML	6.0.1	*YAML* parser
scipy	1.11.1	Scientific computing library
SMBus	1.1.post2	I2C device interface
Spidev	3.6.*	SPI device interface
tkinter	8.6	*GUI* toolkit

Table 1: *List of python libraries used*

2.2 Preparing the boat

2.2.1 Onboard computer

The plan was at first to use a ROS2 architecture for the robot, exploiting the Navio2 ROS compatibility *via* a rosbridge. With this idea in mind, I started setting up a debian 22.04 (Jammy Jellyfish) installation on the Raspberry Pi. After the basic installation process of the *OS* and the initialisation of some useful parameters, like the network settings, I proceeded to install ROS 2 Humble. Despite successfully installation, I repeatedly failed to access the *IMU* from the Raspberry. I was aware that this was due to the fact that I was not using the custom system image provided by Emlid. Nevertheless, being a 32-bit image, it is severely incompatible with the default ROS2 installs, which is why I ignored it in the first place. Indeed, having already tried in vain to find a workaround for this issue, I assumed that trying to understand and to replicate the quirks of the Emlid image would be best.

Unfortunately, not too long after I started investigating these modifications in-depth, I realised that the extent and the subtlety of some of them meant that they could not be replicated in a reasonable amount of time. As a consequence, the whole team came to the conclusion that using the custom image along with a multithreaded program architecture in place of the initially planed ROS2 architecture was the next best thing to do.

2.2.2 Drivers

With the Raspberry Pi fully configured, I started working on the sensors. The first task at hand was to test the code written the previous year. Despite the drivers for all sensor drivers being functional, due to the non-modular and rather unpolished architecture of the main program and of the drivers themselves, it was decided to redo everything by salvaging as much of the code as possible. Nevertheless, because of the numerous unconventional solutions it relied on and of the lack of documentation, we ended up going mostly from scratch. An object-oriented approach was used in this project, it is covered in more details in the introduction of section *2.4. Controlling the boat*

All drivers share more or less the same structure. They work by collecting and storing the latest useful data from the sensor with a set frequency. This useful data is defined in the class constructor using attributes. The constructor is in charge of initialising the latter as well as useful objects and settings, like the measurement thread and associated mutex or the data update frequency. A getter property is associated with each data attribute. It calls an *_initialization()* method, used to configure and establish the communication with the sensor. The thread is started by using the *start()* method and is stopped by using the *stop()* method. Its target method is the *updateData()* method, which is in charge of retrieving the latest sensor data. Lastly, the *shutdown()* method is called to stop and disconnect the sensor.

I started by developing the *GNSS* driver. The useful navigation data retrieved from the sensor is the latitude, longitude, altitude and advancing direction of the boat, as well as the norm of its speed vector and of its components in the *NED* frame along with its ground speed *i.e.* its speed in the North/East plane. The matching the attributes are *_lat*, *_lon*, *_alt*, *_heading*, *_speed*, *_speed_north*, *_speed_east*, *_speed_down*and *_groundSpeed*. Note that *_heading* is not the actual heading of the boat, as the advancing direction takes drift into account. The status of the *GNSS* and the current date are also respectively stored under the *_satus* and *_date* attributes. The sensor provides these information to the driver in the form of ublox messages, detailed in Table 2.

Message name	Associated attributes
MSG_NAV_POSLLH	*_lat*, *_lon*, *_alt*
MSG_NAV_STATUS	*_satus*
MSG_NAV_TIMEUTC	*_date*
MSG_NAV_VELNED	*_speed*, *_speed_north*, *_speed_east*, *_speed_down*and *_groundSpeed*

Table 2: *List of messages sent by the GNSS and associated attributes*

I then worked on the calypso driver. A proof of concept had already been proposed by **Catherine RIZK**, which could not however run parallel to the main program as is. Moreover, persistent unknown issues, which later turned out to be simple battery issues, led to the creation of three drivers for the anemometer. Both drivers collect the observed wind speed and wind direction in the frame of the boat (see section *2.3.2. Model and simulation*) and both of them also allow the user to choose which sensor to connect to, by specifying the *MAC* address of its bluetooth interface. It is of interest to note that this anemometer can be paired with a compass and/or *IMU* to centralise measurements.

The first driver is based on the proof of concept. It uses the calypso-anemometer library (see Table 1) to communicate with the sensor. However, this library uses asynchronous programming, *via* the AsyncIO library, which proved to be an issue for many reasons. Firstly, Python class constructors cannot handle asynchronous code and yet the *_initialization()* method of the anemometer driver uses asynchronous methods from the calypso-anemometer library, which meant this driver could not keep the same structure as the others. Secondly, I needed to cleanly restrict the use of AsyncIO to the driver, as failing to do so would have required to switch the entire logic and syntax of the project to the AsyncIO format, which was not a viable option given the volume of code already produced at the time. For example, the instantiation of the driver needed to be synchronous to avoid the need to analyse and eventually restructure the code to prevent event loop conflicts.
The solution I ended up going with to overcome these difficulties could be called a "synchronous factory method" approach. This approach is based on the "asynchronous factory method" technique, which consists in using a class method to call the constructor and subsequently performing the desired asynchronous operations on the instantiated object before returning it. The main difference is that no coroutines were directly called in the class method, instead the event loop in charge of awaiting the results of the asynchronous tasks is ran. The event loop is created on the first call to the factory method and is reused in following calls. I also implemented basic quality of life features, like automatic reconnection attempts and an adjustable (re)connection attempt counter before raising an error.

The second and third drivers use the same method as the one made the previous year, that is scrapping low level code from library directly and bypassing the use of the AsyncIO library. They can be considered as two versions of the same driver developed separately. I wrote one of the two and was ultimately in charge of merging both versions. As opposed to the calypso-anemometer library approach, The merged driver uses a simple Bluepy bluetooth client to connect to the sensor. Indeed, the former involves a Bleak bluetooth client wrapped in the *CalypsoAPI* class which is supposed to be the main interface (*API*) between the user code and the sensor. The two main benefits of this approach is that the sensor is much responsive and that I could reuse the exact same structure as the one used by other sensor drivers, without having to resort to the workarounds imposed by the use of the AsyncIO Library. The sensor data was obtained by accessing and eventually modifying certain registries of the sensor, detailed in [9]. The main registries accessed and the corresponding values/services are listed in Table 3

Registry *UUID*	**Associated value/service**
0x180A	Device information service
0x180D	Data service
0x2A39	Sensor characteristics
0xA001	Measurement status
0xA002	Data rate
0xA003	Compass status

Table 3: *List of calypso anemometer registry UUIDs and associated values/service*

The driver for the *IMU*s was written by **Ludovic MUSTIÈRE**. It comes with a calibration utility for the accelerometer and for the magnetometer, based on the method presented in appendices B and C of [10] It provides access to the pitch, yaw and roll measured by the sensors. He also developed the driver for the sail and rudder servomotors. This driver allows the user to control an arbitrary number of servomotor through a single class by using configurable profiles.

A generic moving median filter was later implemented to clean up the noisy sensor data. Filters were ultimately only used on the *IMU* and on the anemometer. Standard test programs capable of testing any sensor using a driver respecting the template defined earlier, with or without filters, and servomotors were conjointly writen by **Ludovic MUSTIÈRE** and me.

2.3 Simulating the boat

2.3.1 Modular simulation program

I started working on the simulation after having finished the first functional versions of each driver I was assigned to develop. Like with the drivers, I started by testing the previous code and trying to but ended up discarding and restarting from scratch.
The purpose of the simulation was initially to get a basic understanding of the physics of the sailboat, but I wanted it to be much more than that. I actually wanted it to be a core part of the bundle I aimed at creating. Indeed, because the true goal of the autonomous sailboat project is eventually to be used widely, I wanted to create an all-in-one solution that covered everything from simulation to real testing to optimise the user experience. Consequently, I started thinking about what features should be implemented alongside the baseline simulation and settled mainly on *HIL*, log related functionalities.
I wanted this program to be both exhaustive and adaptable. As a matter of fact, while an exhaustive but rather basic simulation could suffice for educational purposes, adaptability would allow the higher complexity demanded by research to be implementable. The adaptability of the code relies on two key ideas, modularity and flexibility.

On the one hand, the code is modular. First, the implementation of all modules, like control algorithms, boat models and so on, is standardised. For example, all controllers are stored in the same file and are all derived from an abstract template BaseController which ensures their compatibility with the main boat class (see section *2.3. Virtual boat*). The file is scanned by the simulation program on launch and all controllers are identified and added as options to the control algorithm selection menu. The exact same reasoning also applies for supervisors and observers, which makes the implementation and subsequent testing of new variations of these components both simple and intuitive. A similar procedure is also partially implemented for boat classes, except the template parent class is not abstract and is more restrictive. It could easily be implemented for filters, which already respect the proper file structure.

On the other hand, the flexibility of the simulation program shows through the settings of the program. Settings can be easily modified, saved and reloaded at a later time by using configuration profiles stored in *.yaml* files. Settings menus allow the user to tweak the simulation configuration profiles and to save interesting ones easily in an intuitive fashion. Configuration files are useful because they are in fact easier to understand and to work with than raw code implementations (see *Appendix B: Sample configuration file*) The MACRO_CASE syntax convention used in the *.yaml* files for easier processing.

Configurations are dynamic objects storing settings as instance attributes. They are implemented *via* the *ConfigurationOverlay* class which, as suggested by its name, is merely an overlay for the *Configuration* class which is itself the template for all configuration objects. The latter acts as a *.yaml* file handler with basic editing features while the former acts as an interface between the configuration file and the simulation program. At first, these two classes were supposed to be merged but a recurring issue led to their separation before seemingly resolving itself (see section *3.2. Notable issues*) However, they still have a symbolic purpose. On the one hand, *Config* contains all the necessary methods and attributes to load, edit and save changes made to a configuration file. On the other hand, *ConfigurationOverlay* is used store settings as instance attributes, dynamically initialising them based on the data fields of the configuration file loaded, *via* the python *setattr()* method.

2.3.2 Model and simulation

The model used for the boat is introduced in [11]. Let $O(x_0, y_0)$ and $M(x, y)$ be the origin of the selected reference frame and the barycenter of the boat respectively. Let further $\mathbf{x}$ and $\mathbf{u}$ be the state vector and system inputs respectively, with $\mathbf{x} = (x, y, \theta, v, \omega)^\top$ and $\mathbf{u} = (\delta_r, \delta_{s,max})^\top$, where the notations listed in Table 4 and illustrated in Figure 11

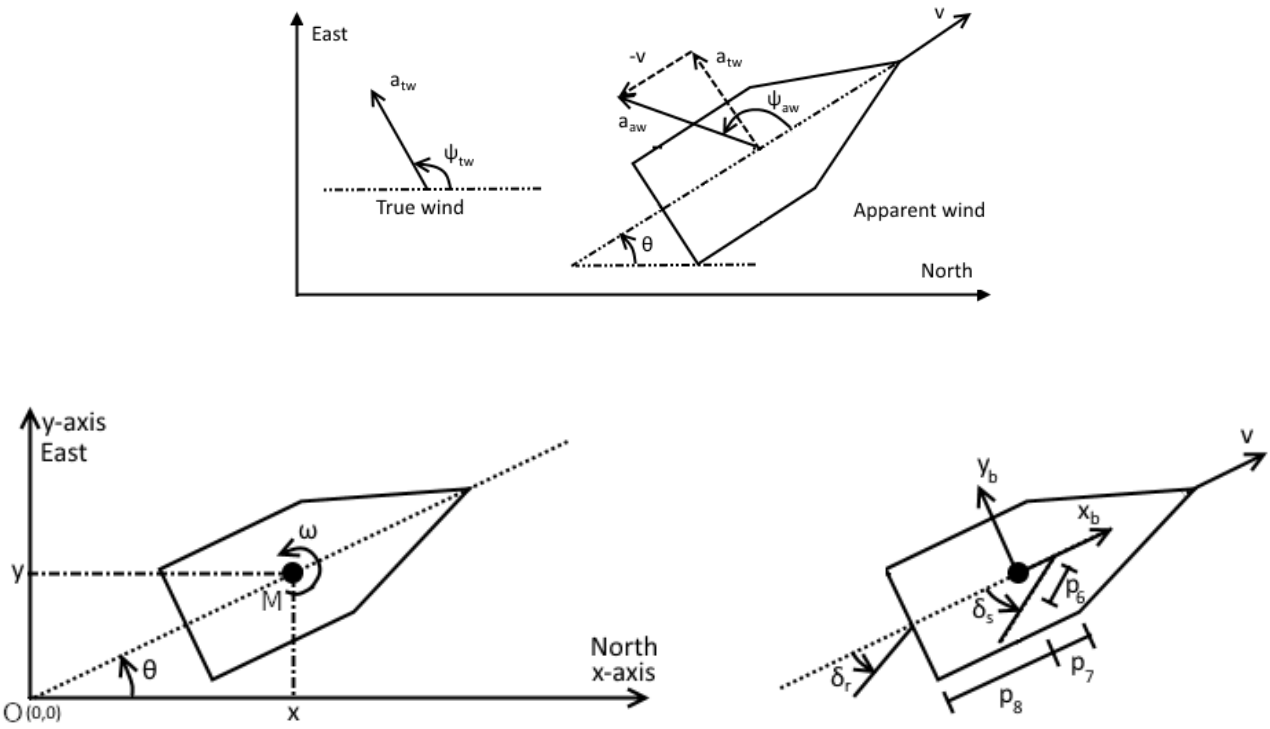


Figure 11: *Illustrations of the model*

we use the following state equation:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} v\cos(\theta) + p_1 a_{tw}\cos(\psi_{tw}) \\ v\sin(\theta) + p_1 a_{tw}\sin(\psi_{tw}) \\ \omega \\ (g_s\sin(\delta_s) - g_r p_{11}\sin(\delta_r) - p_2 v^2)/p_9 \\ (g_s(p_6 - p_7\cos(\delta_s)) - g_r p_8\cos(\delta_r) - p_3\omega v)/p_{10} \end{pmatrix} \tag{1}$$

Notations			
Symbol	**Value**	**Symbol**	**Value**
x	x coordinate of the boat in the *NED* frame	$\delta_{r,max}$	maximum rudder angle
y	y coordinate of the boat in the *NED* frame	$\delta_{s,max}$	maximum sail angle
θ	heading of the boat	g_r	force applied to the rudder
v	speed of the boat	g_s	force applied to the sail
ω	angular speed	a_{tw}	true wind norm
δ_r	rudder angle	ψ_{tw}	true wind direction
δ_s	sail angle	a_{aw}	apparent wind norm
$\delta_{r,min}$	minimum rudder angle	ψ_{aw}	apparent wind direction
$\delta_{s,min}$	minimum sail angle		

Model parameters			
Symbol	**Value**	**Symbol**	**Value**
p_1	drift coefficient	p_7	distance to mast
p_2	tangential friction	p_8	distance to rudder
p_3	angular friction	p_9	mass of the boat
p_4	sail lift	p_{10}	moment of inertia
p_5	rudder lift	p_{11}	rudder break coefficient
p_6	distance to sail center of effort		

Table 4: *Notations and model parameters*

a_{tw} and ψ_{tw} can be expressed as functions of a_{aw}, ψ_{aw} and θ which are known variables. The forces exerted on the rudder and sail can be expressed as functions of model parameters and system inputs. The reader is invited to refer to [11] for details on these assertions.

Note that, in reality, we do not directly control the angle of the sail δ_s but rather its maximum possible angle $\delta_{s,max}$. Moreover, $\delta_{s,min}$ is introduced for the sake of convenience but is not inherently useful. Indeed, whereas $\delta_{r,min}$ and $\delta_{r,max}$ are used to actively control the range of motion of the rudder, $\delta_{s,min}$ is 0 by definition.

The simulation is object-oriented. The main *Simulation* class has attributes for key objects, like thread, mutexes, drivers, data classes, and settings. It includes many methods associated with its different key features, the main ones being *_simCallBack()*, *_logCallback()* and *_setupWizard()*. The two first ones are responsible for running the main control loop of the simulation and handling the log replaying functionality respectively while the latter is in charge of the settings menu.

At the heart of the simulation is my *_modCallback()* method. I implemented a Runge-Kutta 2^{nd} order numerical integration method to integrate the equations of the model. It runs in its own thread *_modThread*, which is exclusively in charge of computing the state of the simulated boat. A major advantage of this structure is that changing the integration method is as simple as replacing the *_modCallback()* method with the implementation of the desired replacement method. The replacement method simply has to match the expected output format specified in the documentation.

2.3.3 Virtual boat

While I was developing the simulation, one of my main concerns was the transposition of the implementation of control algorithms from the simulated system to the real system. We had came to the conclusion that implementing a main boat class to centralise the control algorithms, drivers, so on and so forth, was the best approach. I then took it upon myself to adjust the *Sailboat* class, implemented by **Ludovic MUSTIÈRE**, and to create a child class derived from it called *VirtualSailboat*.

The *Sailboat* class stores the drivers, filters and control algorithms used by the boat as attributes. It has methods to start, stop and shutdown all the aforementioned drivers. It also includes methods to update sensor measurements and actuator commands.

The *VirtualSailboat* class contains utility methods to allow the main control loop of the simulation to be as close as possible to the one used for real mission programs (see *Appendix F: Algorithms*). It stores the dynamic model of the boat as method, used by the simulation core method *_modCallback()* (see section *2.3.2. Model and simulation*), and includes attributes to keep track of the evolution of the model. It also features attributes used to track the nature of the drivers used, *i.e.* real or virtual.

Indeed, in addition to the *VirtualSailboat*, I also implemented virtual component drivers. They allow to feed data to the *Sailboat* class with the same format as the corresponding real driver. One downside is that they absolutely need to be paired with a *Simulation* object to function properly. Virtual sensor drivers are all derived from the abstract parent class *VirtualSensor*. The latter features the same *start()*, *stop()* and *shutdown()* methods as real sensor drivers. They also use the same property system to access useful values and the same thread system used to update them, with the difference that getter properties are used to noise the otherwise perfect measurements. Nevertheless, the main difference lies in the measurement thread target, which here is the *_measure()* method. This method is a basic while loop calling the *_updateData()* abstract method on each iteration, while the driver is running. *_updateData()* is meant to retrieve the values of the simulated boat state matching the desired sensor values.
the *VirtualServo* and *VirtualServomotors* classes are basically one to one matches of the *Servo* and *Servomotors* classes implemented by **Ludovic MUSTIÈRE**. The main difference is that instead of sending a *PWM* command to the servomotors, they directly change the values of the system inputs stored in the *Sailboat* class.

2.3.4 Hardware in the loop

I made a basic utility, based on the machine library, capable of identifying whether the program is running on the on-board computer or on another device. It relies on the fact that the *machine.platform()* function returns *'aarch64'* when running on the Raspberry Pi 4 amrv8 architecture. This utility is used, among other things, to load specific libraries on specific devices only, *e.g.* to load Spidev and SMBus (see Table 1) on the Raspberry only.

I initially planned the *HIL* functionality to be a part of a broader "Real-time simulation" (*RTS*) feature. Following the all-in-one bundle logic, *RTS* would have been used to launch missions on the real boat from the same program that allowed to run simulations. However, because it turned out to be as simple as adding another control loop, I discarded the feature some time after implementing it. I was quickly made understood that it was more reasonable to have only one control loop structure, shared for the simulation and real missions (see section *2.3.2. Model and simulation*). Making a separate mission launcher program matter-of-factly reduced the complexity of the code while still allowing us to attain the desired end result.

In the end, *HIL* was passively merged with the main simulation feature. Thanks to the way I implemented virtual drivers, it indeed turned out that implementing *HIL* was as simple as switching virtual drivers and real drivers. To enable this, I added a components menu, under the main settings menu, that allows the user to select between virtual and real state for the components. I also implemented extensive checks to ensure that real drivers could only be picked to real when the corresponding real component was actually accessible by the program.

2.3.5 Logs and replay

One of the most important aspects of testing is the analysis of the results. For this reason, I implemented a log replay feature to the program which made reviewing logged data a lot simpler. Although initially destined for real mission, the implementation of the *VirtualSailboat* class meant that logs of simulated missions could also be recorded. This incidentally made testing and debugging the logging process a lot easier.

While **Ludovic MUSTIÈRE** developed the logger class, I defined the log file format. Log files are *.csv* files divided in two parts, metadata stored in a header and data stored in the main body of the file (see *Appendix C: Sample log file*) I also implemented a log reader utility capable of extracting both the data and metadata from a log file and of storing it in instance attributes where the log replay method could easily access them.

2.3.6 User interface

The *HMI* is a key component of the user experience, hence the special care it received. Two user interfaces were made, I worked on the *CLI* with **Ludovic MUSTIÈRE** and brought a minor contribution to the *GUI* made by **Catherine RIZK**. A diagram of the global structure of the *HMI*, including the differences between the *CLI* and *GUI* structures, is presented in *Appendix D: HMI structure*

The *CLI* was designed for the Raspberry, so that we could run *HIL* among other things. It was made using bash scripts and the python argparse library. It consists of a succession of interconnected menu screens.

The bash scripts are in charge of two things. First, they ensure the proper initialisation of the program, *e.g.* by pre-running servomotor drivers to give the main program access privileges for the necessary files. Then, they define useful aliases to make using certain functionalities of the program easier. For instance, launching sensor or actuator tests with default settings for a specific sensor is as simple as using the command *testComponent*, where 'Component' is to be replaced by the name of the component of choice, *e.g. testIMU* or *testCalypso*.

While aliases are used to define shortcuts, argparse is used to define the main command. Using the *ArgumentParser* class, I was able to retrieve and handle the command line arguments. I was also able to standardise and document the command.

The *CLI* is mostly composed of python based interfaces, built using the default python functions *input()* and *print()* and nested while loops. I made a significant effort to make the parts of the interface I worked on foolproof. All the settings menus I developed are capable of basic input syntax verification, accepting only existing options with eventual valid matching arguments. Some screenshots of the *CLI* can be found in *Appendix E: CLI screenshots*.

The *GUI* made using pygubu and tkinter. It was added to improve the user experience when running the simulation is running on any other device than the on-board computer. I developed a class to handle the simulation window, *i.e.* the window containing the plot used to display either a live simulation or a log file replay. It was initially part of the *CLI* before we decided to implement a full *GUI*, at which point the class was renamed to *SimulationPlotter* and relocated to the *GUI* file.

The *SimulationPlotter* class has instance attributes for tkinter settings, matplotlib settings, threads and mutexes. Most importantly it can access the instance of the *Simulation* class currently running as one of its instance attributes. It uses a thread named *__drawSimuThread*, targeting the *__drawSimuCallback()* method, and a couple other utility methods and functions to update the plot with data from *Simulation*. Moreover, it uses the tkinter main loop to run a keylogger, which allows user to interact with simulation window shown in Figure 12. The user can pause and resume the simulation by pressing the 'p' key and quit the simulation, completed or not, by pressing the escape key. At first, the pause feature was meant to allow easier frame by frame analysis of the data but the data display was never implemented to the simulation window for lack of time. They can also move the camera by using arrow keys, zoom in or out by using keypad plus and minus keys, and change of camera with the 'c' key. The camera settings can be reset by using the 'r' key. Note that resetting the camera only affects the currently selected camera. Last but not least, the user can also switch from camera mode to control mode, if a compatible manual controller is used by the boat. This feature was initially intended to simulate serial control (see section *2.4.2.2. Serial*).

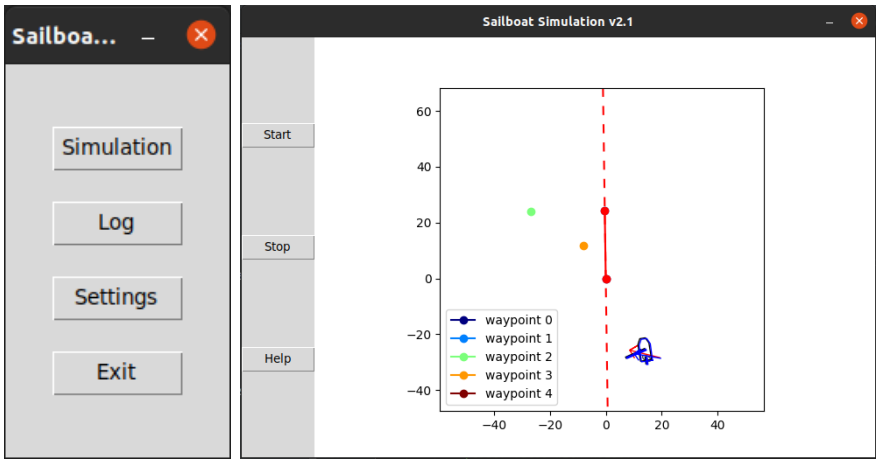


Figure 12: *Screenshots of the GUI: main menu (left) and simulation window (right)*

2.4 Controlling the boat

2.4.1 Autonomous operation

The main algorithm used for autonomous operation is based on a simple control loop (see section *Appendix F: Algorithms*). There were initially multiple control loops made for different contexts, *e.g.* for the simulation, for real missions, so on and so forth, which were later all replaced by a single loop. All control algorithm implementations were made respecting a certain protocol, which significantly facilitates the addition of new algorithms to the simulation (see section *2.3.1. Modular simulation program*).

2.4.1.1 Supervision

In our program, supervisors are the algorithms that define the behaviour of the boat, *i.e.* the mission it tries to accomplish. They define targets based on the state vector of the boat. the main missions we tested the boat on were line following, station keeping and path following. In the two first scenarios mentioned, the targets are respectively a line between two waypoints and a single waypoint.

I was in charge of the implementation of all supervisors. Supervisors are all derived from the abstract parent class *BaseSupervisor*. The latter has attributes for the waypoints list loaded, the target index i_{targ} which uniquely identifies the target, and the color map used to draw targets on the simulation window plot whenever the simulation running. Setter and getter properties for the target index were also added to make them wrap around at the end of the list, *i.e.* the index value is defined modulo the length of the waypoints list.
It features two main abstract methods, namely *get_target()* and *draw_target()*. These methods are respectively used to compute and return the target of the boat, and draw all the known waypoints as well as the current target on the simulation window plot whenever it is possible. A total of three supervisors based on BaseSupervisor were implemented.

The *SupervisorLineFollowingDefault* is in charge of the line following mission mentioned earlier. It targets the line between the waypoints at indices i_{targ} and $i_{targ}+1$ and returns it as a list of these two waypoints. Given that it is a basic test algorithm, the target is set once before launch, at instantiation, and remains constant until the end of the test.

The *SupervisorStationKeepingDefault* is in charge of the station keeping mission mentioned earlier. It targets and returns the waypoint at target index i_{targ}. Given that it is a basic test algorithm too, the target is also set once before launch and remains constant until the end of the test

The *SupervisorPathFollowingDefault* is in charge of path following missions. Let us call a path a closed polygonal chain of length $n \in \mathbb{N}$. The specificity of the considered path is that it is defined by the waypoints file loaded, that is the ordered list of its vertices as *GNSS* waypoints. The path following mission simply consists in following the considered path from start to finish and to station next to its first waypoint after having completed a lap.
Practically speaking, it is simply a combination of the two previous supervisors, *i.e.* we follow successively the different segments making up the path and finish by stationing next to the starting waypoint after having reached target index $i_{targ} = n$. As such it returns the waypoint at target index i_{targ} or a list of the waypoints at indices i_{targ} and $i_{targ}+1$, depending on the mission progress. The condition to change the currently targeted segment is reaching the half-plane delimited by line perpendicular to it and passing by its end point, which does not contain the starting waypoint of the target segment. Let $W_{i_{targ}}$ and $W_{i_{targ}+1}$ be the waypoints at indices i_{targ} and $i_{targ}+1$ respectively. The aforementioned condition is then simply equivalent to the following basic inequation, *i.e.* we increase the value of i_{targ} by 1 as soon as this inequality is verified.

$$\overrightarrow{W_{i_{targ}}W_{i_{targ}+1}} \cdot \overrightarrow{W_{i_{targ}+1}M} > 0 \tag{2}$$

2.4.1.2 Control

In our program, controllers are used to ensure that the targets issued by supervisors are actually reached. They do so by defining target values for δ_r and δ_s, based on the boat state vector and the target defined by the supervisor, and by ensuring that these target values are actually respected by the servomotors. The commands determined by the controllers are in radians and are converted by the servomotor driver to *PWM* commands. Because the commands depend on the target returned by the supervisor, and the format of said target by proxy, certain controllers will only be compatible with certain supervisors.

I took it upon myself to standardise the implementation of all controllers, which was taken care of by **Catherine RIZK** and **Ludovic MUSTIÈRE** in the first place. I also fully implemented the path following controller. Controllers are all derived from the abstract parent class *BaseController*. The latter has attributes for the configuration file used by the boat which contains all the controller parameters.
It features one main abstract method, namely *get_cmd()*. This method is used to compute and return the commands as a 2d vertical numpy array. A total of four supervisors based on *BaseSupervisor* were implemented.

The *ControllerLineFollowing* is the default controller associated with the supervisor *SupervisorLineFollowingDefault*, it is based on the algorithm presented in [12]. The *ControllerStationKeeping* is default controller associated with supervisor *SupervisorStationKeepingDefault*, inspired by various algorithms presented in [12, 13, 11]. The *ControllerPathFollowing* is the default controller associated with the supervisor *SupervisorPathFollowingDefault*. It is simply a fusion of the implementation of the two previous controllers, using the appropriate behaviour to handle the format of the target provided by the supervisor at any given time. Lastly, the *ControllerViel* is a station keeping controller based on the algorithm presented in [14]. The algorithms for every controller are detailed in *appendix F: Algorithms*.

2.4.1.3 Observation

In our program, observers are used to get an estimate of the state vector of the boat at a given time. They use sensor measurements and/or prediction equations to compute this estimate. It is that estimate which is used to feed the supervisor and controller.

I was in charge of the implementation of the observers. Observers are all derived from the abstract parent class *BaseObserver*. The latter has attributes for the estimation of the boat state.
It features three main abstract methods, namely *updateEstimate()*, *drawEstimate()*, *reset()* These methods are used to compute the estimated state of the boat, to draw the estimated state of the boat on the simulation window plot whenever it is possible, and to reset the state estimation. A single observer based on BaseSupervisor was implemented.

The *ObserverEKF* is an extended kalman filter implementation. It has attributes for the initial values of the state estimate. One of its key feature is its usage symbolic computation, *via* the *lambdaKalman()* method (see *appendix G: Kalman filter implementation*), to compute the required mathematical functions. It is capable of pickling the resulting lambda functions to lambdaKalman file, the dill library. This is meant to allow the Raspberry Pi to bypass the symbolic calculation step, as it is not powerful enough to quickly get through it. I also added an option to forcefully rerun the symbolic calculations on instantiation. Another interesting feature of this observer is the way it performs the prediction step of the extended kalman filter. It relies on basic assumptions regarding the wind evolution to use a Runge-Kutta 2^{nd} order numerical integration method (see *Appendix G: Kalman filter implementation*). In summary, let dt be the temporal sampling step used for discretization, we assume that the true wind intensity $a_{tw,n}$ at time $n \cdot dt$ can be approximated by $a_{tw,n} = a_{tw,n} + (a_{tw,n} - a_{tw,n-1})$, where $a_{tw,n-1}$ is the true wind intensity at time $(n+1) \cdot dt$. Lastly, note that the measurement vector used by the filter is $\mathbf{y} = (x_{gnss}, y_{gnss}, theta_{imu}, v_{gnss})^\top$.

2.4.2 Manual control

Manual control was implemented in the main loop in a way that allows the user to take over the control algorithms used for autonomous operation by overriding commands from the controller. Two manual control methods were initially planned, *RC* and serial, though *RC* was the only one ready to use by the end of the project.

Whereas the *RC* control relies on a HK-T4A emitter and a HK-TR6A receiver (see *2.1.1 Hardware*), the serial control would have relied on a pair of XBEE modules, most likely the XBEE Pro series 1 modules used the year prior. The distinction between the two control methods using the names *RC* and serial is made for the sake of convenience but both of these methods use radio frequencies to communicate with the boat. The priority in of the command chain is defined as follows: *RC* overrides serial which overrides control algorithms.

2.4.2.1 RC

I wrote the *RC* driver. It includes attributes for the command readings, remote status, useful threads and mutexes along with getter and setter properties for the reading attribute. Readings are stored *via* the utility data class *RCReading*, used to store all four channel inputs from the receiver (2 axes times 2 joysticks) and the corresponding rudder and sail commands.
Like others divers, it has *start()* and *stop()* methods with a thread, targeting the *getReading()* method, which to update the reading, but has no initialisation nor shutdown method, Furthermore, it runs a thread named *_statusThread*, targeting the *updateStatus()* method, to continuously monitor the status of the remote using the utility method *isIdle()*. This thread differs from the *_readingThread* thread targeting the *getReading()* method, which continuously retrieves inputs from receiver module.
Its main specificity is that I built it so that it would behave like a controller in the eye of boat. This translates into the addition of a *get_cmd()* method which returns the desired sail and rudder commands in the same format as a controller would.

The class also features class attributes for the recorded minimum, idle and maximum value of each channel detailed in Table 5. They can be adjusted using the test program, to calibrate the controller inputs. In our case, we use channel 2 to control the sail and channel 0 to control the rudder, which translates to right joystick throttle and left joystick steering. Let $ch_{i,min}$, $ch_{i,idle}$ and $ch_{i,max}$ be the minimum, idle and maximum recorded inputs for the i^{th} channel and ch_i be the current input value of the i^{th} channel. The corresponding target angles is obtained using the basic formula, using the notations defined above and in Table 4:

$$\delta_r = \frac{\delta_{r,max} - \delta_{r,min}}{ch_{0,max} - ch_{0,min}}(ch_0 - ch_{0,idle}) \tag{3.1}$$

$$\delta_{s,max} = \frac{\delta_{s,max} - \delta_{s,min}}{ch_{2,max} - ch_{2,min}}(ch_2 - ch_{2,min}) \tag{3.2}$$

These channel values are also used to detect when the *RC* emitter is online. The *updateStatus()* method is in charge of updating status attribute of driver. Whenever *RC* emitter is offline, inputs on channel all channels are at idle values with a maximum observed deviation of 3 points. We consider that *RC* enabled as soon as we leave this interval on even one of the channels. However we wait for a couple status thread iterations with idle channels before switching the *RC* status back to offline. This delay is here to avoid a scenario where the servomotors receive controller commands in between two *RC* commands because of a signal loss, which could cause an erratic behaviour.

Channel n^o	Minimum value	Idle value	Maximum value
0	1075	1499	1945
1	1071	1499	1883
2	1150	899	1818
3	1081	1499	1930

Table 5: *Recorded channel input values*

2.4.2.2 Serial

The serial control method was supposed to allow keyboard control of the boat from the simulation program before we decided to create a separate mission launcher. It was also intended to provide real time feedback from the on-board computer to a connected remote computer. This would have been used, among other things, to pilot a simulation running on the on-board computer using the *GUI* on a remote computer. In the end it was not implemented for a lack of time.

2.5 Testing the boat

Live testing took place at the Bournville Radio Sailing and Model Boat Club. It is a large and shallow artificial lake offering ample space to test a model sailboat. Waypoints were arbitrarily placed on the lake using Google Maps, both sets of waypoints used for testing are shown in Figure 13. As mentioned previously, the missions tested include line following, station keeping and path following (see section *2.4. Controlling the boat*).

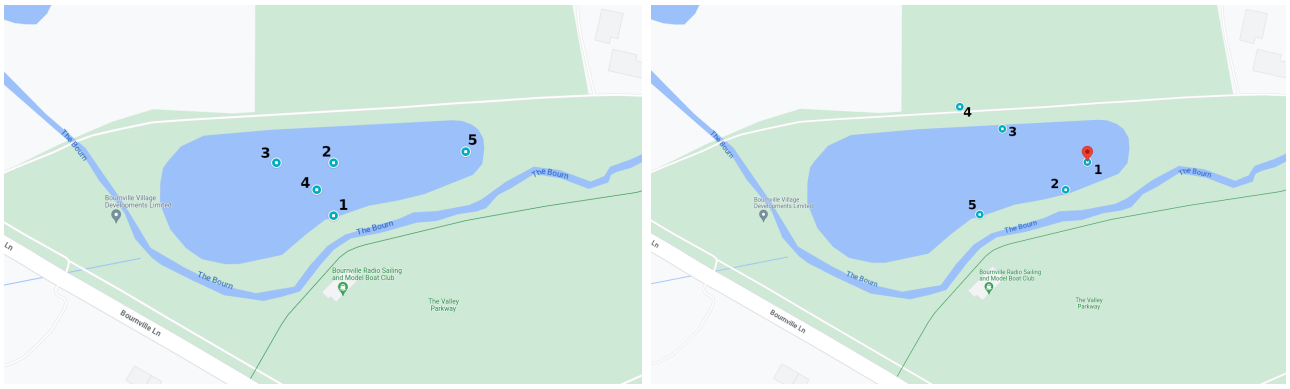


Figure 13: *Sets of waypoints used*

The procedure for starting the boat is fairly simple. Firstly, all the on-board components are connected and powered on. Secondly, the access point is turned on and the *SSH* communication between the on-board and remote computers is tested. Thirdly, because the on-board electronics are not in a waterproof case, the boat hull needs to be sealed with waterproof tape. Fourthly, the desired mission is selected and launched from the remote computer. Lastly, the boat is put on the lake and kept in place until the initialisation process is complete, at which point it is released and starts to operate autonomously.

We came across many issues during live boat tests. One of our main sources of troubles was the bad weather. Although we had good wind on the lake, we also rarely had any sunlight, which made using the solar anemometer a real chore as it would almost always be in low power mode, causing innumerable connection issues. Another issue we faced was with the servomotors, it is explained in details in section *3.2. Notable issues*. Another problem we had was with the recorded boat position. For some reason, we realised while on the lake that the real evolution of the position of the boat did not match the one depicted in the log files. It turned out that multiple successive inversions of the x and y axes, which would end up correcting themselves in the log file but not in the boat main control loop, were spread throughout the code

In spite of all these challenges, we were able to produce some experimental results. The best results achieved were attained during a station keeping mission. This mission can be divided in four phases. Firstly, the boat is on stand by and is being held in place while it initializes itself. Secondly, it is released facing the general direction of the target waypoint, to avoid any collision with the shore, and it eventually manages to follow a line straight to it. Thirdly, once on the waypoint, it tries to stay near the target but eventually drifts away before subsequently trying to no avail to get back in the target zone. Lastly, the boat is seen drifting sideways for a while, as we assume control of it, before we bring it back to the shore. The full replay of the associated log can be found here (or by following this link: `https://youtu.be/BWaITWV6-Yc`).

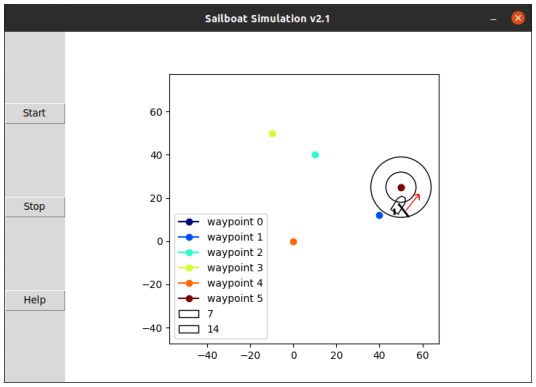


Figure 14: *Screenshot of the successful mission replay*

The log file generated shows that the boat had successfully reached and stationed next to the target waypoint but the recorded heading, which was initially correctly, seemed completely off by the time we took control of the boat. Given the progressive nature of the appearance of the discrepancies, I felt like this would only come from a physical issue with the boat and indeed we found that the *IMU* was the responsible for this behaviour. As a matter of fact the *IMU* kept repeatedly uncalibrating itself after working seemingly fine for a while once recalibrated. We were not able to pinpoint the specific cause or causes of this issue for lack of time, although I suspect the proximity of the sensor to both of the high torque servomotors to be the main culprit. Moreover, in spite of the projection algorithm used to get a better value for the heading of the boat, the high roll angle values naturally experienced by a sailboat during navigation may also contribute to the inaccuracy of the measurements. In any case this explains why the boat was capable of station only up until a certain point, the point past which *IMU* measurements were to erroneous for the algorithms to work.

2.6 Documenting the boat

One of my key goals with the project was to ensure its reusability and to make it easily deployable. Therefore, It was only natural that a gitlab was set up for the project [15]. Furthermore, two other measures were taken to that effect, namely having a fully documented code and exhaustive readme files on the git. On the one hand, the documentation of every object, function and important variable is meant to allow future developers that would need to tweak the existing code to do so more easily. On the other hand, the readme files were made to describe every important folder of the git and to simplify overall the inner-workings of the project (see *appendix H: README files*) They act as user quick-start guides and are oriented towards students who would be using the platform in a pedagogical context.

The documentation process was spread over two sessions, one halfway through the project and one at the end. The first one was more focused on the git. The whole team settled on the final folder architecture of the project and proceeded to sort the files accordingly. The majority of the readme files were also added and the existing code almost entirely documented, leaving only work-in-progress sections untouched The second session was more centred on the code, being almost exclusively dedicated to the documentation, with the only exception being the addition of the few missing readme files.

3 Results

3.1 Work accomplished

By the end of the Internship, I was ultimately unable to reproduce the results obtained by the previous team in charge of the project. The boat was only ever capable of partially clearing any provided missions, as explained in section *2.5. Testing the boat*. However, this final result is far from representative of the work done.

Indeed, a lot of has been done on the software side of the project. I wrote two functional drivers for the anemometer and one for the *GNSS*. I implemented a significant portion of the features of the simulation program, including but not limited to the core simulation loop, the main control loop, the log replay functionality and multiple settings menus. I also implemented control algorithms in the form of basic supervisors as well as an extended kalman filter, all successfully tested in the simulation, and standardised the implementation of controllers. Lastly, I contributed to documenting the code.

The main added value over what had already been done last year is that these tasks were all accomplished with the idea of reusability in mind. With this notion at the heart of my approach for this project, I ensured that future developers and users could easily reuse and/or improve the code.

3.2 Notable issues

I encountered many challenges throughout the project, from basic ubuntu errors to obscure python issues or even hardware troubles. The most notable ones that I was able to overcome are listed below. They are characterised by their complexity and the interesting solution they required to be solved, which in my opinion made them worthy of this special attention.

The first interesting issue I was faced with was the implementation of the *Config* class for the simulation program. Initially the configuration classes were going to follow same logic as drivers, one class per configuration profile with all the classes derived from an abstract parent *Config*. However, I did not like this idea as it seemed very limiting and tedious to work with given that configurations were bound to change with the evolution of the program. As a consequence, I came up with the idea of creating a single configuration class with dynamically updated attributes, initialised using the contents of *.yaml* configuration files (see *Appendix B: Sample configuration file*). The main advantage of this method is that configuration files are more flexible and understandable for end-users.
The method I initially selected was to make only two classes. The main parent class would contain all the non-dynamic methods and attributes and the child overlay class would use the default python function *setattr()* to dynamically modify its instance attributes on constructor call (see section *2.3.1. Modular simulation program*). The problem I had with this implementation was a recurrent error message: *AttributeError: 'Resource' object has no attribute '__dict__'*, apparently generated because the object I was using *setattr()* on had no *__dict__* attribute, making the function inoperative on said object. After some research, it turned out that only classes had *__dict__* attributes and not instances. For this reason, I implemented an *InitConfig* method to modify the *Config* class *__dict__* before instantiating it, and subsequently resetting the *Config* class *__dict__*, before ultimately returning the instance with modified attributes. This solution worked flawlessly but some time after its implementation new tests revealed that problem had disappeared. As a result, I decided to go back to the initial method which was less susceptible to be affected by python updates.

The most complex challenge I had to face was development of the first anemometer driver. The main issue was to limit the use of the AsyncIO library, used by the calypso-anemometer library, to the driver (see section *2.2.2. Drivers*). Even without any prior experience with the library, I quickly came across the concept of factory method for performing asynchronous operations when instantiating an object. I was then quickly capable of adapting and integrating this method to our project. The real difficulty was to handle the measurement acquisition from the sensor, as it was using async coroutines. The first solution I used was to run an event loop in the measurement thread target method *_updateData()*, in order to wait for the results on each iteration of the thread loop.
The calypso-anemometer library provides methods to generate both real and dummy sensor readings. While this first solution worked with the fake readings produced by, it would crash when using real measurements. Testing would systematically result in the error message *RuntimeError: Task got Future <Future pending> attached to a different loop*, stating that an event wheel received a task it had not started which it cannot handle. Upon preliminary investigation, it turned out that the loop used by the message dbus driver of the Bleak client used by calypso-anemometer library was using a closed loop to run events, while the loop used by my driver was running. More specifically, this might actually have been an issue with the *_bus* attribute of the *BleakClientBlueZDBus* backend used by the Bleak client, of which the initialisation is handled within an async *events.get_running_loop* context manager, in the *BleakClientBlueZDBus.connect()* method. The fact that this method is used only by the real *API* and not the fake one also explained the difference in behaviour observed. This was the first clear indication that the issue was AsyncIO related and not multithreading related as initially suspected given that a proof of concept driver for the calypso had already been made. Screenshots of the key error message and of the event loop identification test are available in *Appendix I: Anemometer driver investigation*.

I then hypothesized that the *get_running_loop()* method from AsyncIO, which I used to get the event loop handling the async coroutines in the measurement thread, was returning a different loop at the time of the thread execution than at the time of the initialisation of the Bleak client. Upon further investigation, it turned out to be true. A temporary loop is used during the initialisation of the *MessageBus* instance used by the Bleak client and subsequently closed after the initialisation. It can be accessed *via* as the instance attribute *_loop* of the *MessageBus* class. This loop is explicitly shutdown on coroutine *call()* execution, and the boolean *_loop._asyncgens_shutdown_called* is consequently set to True.
After many unsuccessful attempts at trying to find a workaround either using dedicated AsyncIO methods to run the measurement thread with *_updateData()* being an async coroutine or even trying to modify the loop declarations in the library directly, I ended up being able to go back to the initial method by isolating and centralising all AsyncIO access points in our code. I did this by forcing the creation and use of a new event loop on the first instantiation of the calypso driver, defining it as the new main event loop, and reusing it in eventual following instantiations instead of recreating it.

The final issue I would like to discuss is a hardware related one. A repeated issue we observed during the initial hardware test was that, when a new target was provided to the servomotor before it had the time to reach the previous one, the Raspberry Pi would often end up crashing. This problem was only ever observed in the servomotor testing phase and although it was very easily reproducible, it never seemed to occur in *HIL* simulated missions. Nevertheless, I started noticing unprompted ill-timed Raspberry crashes throughout the later half of July, whenever we would go for live tests of the boat.
The fact that the servomotors were never a source of troubles until then and that the mechanical part of the system had allegedly been dimensioned and tested the previous year meant that it took a while before I started considering them as potential culprits. Nonetheless, it turned out after further investigation that what we mistook for an outlying failure scenario was actually just a consistent setup to generate current consumption spikes. The issues we had all along were coming from using same battery/*BEC* pair for the motors and the on-board electronics, which could not sustain power delivery in the case of a current spike. This issue was temporarily fixed by attaching a second battery to the Raspberry Pi USB-c power input. The more permanent solution of using two separate battery/*BEC* pairs, one providing power to the electronics and the other to the motors, was later considered.

3.3 Work left

Even if a lot was done, there is even more left to do, as evidenced by the experimental results attained. The first step would be of course to fix the only remaining known unsolved issue, *i.e.* the *IMU* calibration problem. However, there are many improvements that could be made beyond that.

The first major improvement would be to switch from a multithreading-based program structure to multiprocessing-based one. Indeed, in later stages of project, we actually started hitting computational limitations of using only one physical processor core. The main problem we faced was the inability to lower the main control loop frequency to a satisfying level because of the number of threads having to run on a single core.

A related improvement would be to add a cooling solution to the on-board computer. As a matter of fact, in later stages of project, we started facing thermal throttling issues partly due to absence of cooling solutions for the Raspberry Pi 4 processor beyond the integrated heat spreader. These troubles having arisen when using a single core, multiprocessing would almost certainly require an active cooling solution to be implemented to the build.

Another significant upgrade would be the proper implementation of serial communication to and from the boat. The template driver for the XBEE module which was used the year prior to enable this very mode of communication is already in place and only needs to be complete. Serial communication could be used for manual control, connecting the on-board computer to a remote computer or using the latter as a remote. It could also be used for real time feedback applications, like displaying the evolution of the boat in real time on a remote computer or adding a remote display option for the simulation and log replay when it is running on the Raspberry Pi.

One minor upgrade would be to switch the entire code to the CamelCase syntax and to *DRY* it. I particularly would have liked to do this before the end of the project, as it would have made the code slightly more intuitive to modify. Indeed, in its current state, the code uses a mix of CamelCase and snake_case which would make it annoying to settle on a syntax convention when modifying the code in the future.

4 Conclusion

During my internship at Aston University, I was able to work on an autonomous sailboat and to acquire experience with this type of drone. Despite being unable to reproduce last year's results, I ensured that the next students who will take on this endeavour will have much easier time seeing through it. Ultimately, my real contribution to this project was to get it closer to its final goal of offering an easily deployable and highly adaptable platform for testing and educational purposes, by laying down clean foundations for the software side. As a result, I was able to get familiar with the importance of standardisation in an academic context, especially in order to facilitate the reproducibility of experimental results which is one of the key steps of research and of the scientific method in general.

This internship also gave me an opportunity to immerse myself in a new work culture. This gets me closer to one of my personal goals of getting accustomed to as many work ethics and work cultures as possible in my early career, especially european ones, as mentioned in my first-year career plan report. The fact that it was in an academic context also aligns with my career project, as I intend to pursue a *Ph.D.* after graduation from *ENSTA* Bretagne.

I am grateful for the opportunities and learning experiences provided by my internship at Aston University. Indeed, when putting the latter in perspective with my career plan, it becomes clear that this experience was valuable both from a technical and cultural standpoint. Moreover, the gap-year I am currently taking will definitely give me time to draw even more insights from said experience. It will undoubtedly serve me well in my academic and professional journey and I look forward to continuing to build upon them.

Glossary

DDboat A differential drive model boat.

s@60° A servo speed measurement unit based on the amount of time, in seconds, it takes a servo arm to sweep left or right through a 60 degree arc at either 4.8 volts or 6 volts.

Acronyms

API Application Programming Interface.
BEC Battery Elimination Circuit.
CLI Command Line Interface.
DRY Don't Repeat Yourself.
ENSTA École Nationale Supérieure de Techniques Avancées.
GNSS Global Navigation Satellite System.
GUI Graphical User Interface.
HAT Hardware Attached on Top.
HIL Hardware In the Loop.
HMI Human Machine Interface.
I2C Inter-Integrated Circuit.
IMU Inertial Motion Unit.
MAC Media Access Control.
NED North East Down.
OS Operating System.
PPM Pulse Position Modulation.
PWM Pulse Width Modulation.
RC Radio Control.
RTS "Real-time simulation".
SPI Serial Peripheral Interface.
SSH Secure Socket Shell.
UAV Unmaned Aerial Vessel.
USV Unmaned Surface Vessel.
UUID Universally Unique Identifier.
YAML YAML Ain't Markup Language.

List of Figures

List of Tables

References

[1] *Rankings | Aston University*. URL: https://www.aston.ac.uk/undergraduate/why-aston/rankings (visited on 09/14/2023).

[2] *Courses | Aston University*. URL: https://www.aston.ac.uk/courses (visited on 09/14/2023).

[3] *Research | Aston University*. URL: https://www.aston.ac.uk/research (visited on 09/14/2023).

[4] *HS-785HB Karbonite, 3.5 Turn Winch Servo | HITEC RCD USA*. URL: https://hitecrcd.com/products/servos/analog/boat-analog/hs-785hb-3.5/product (visited on 09/14/2023).

[5] *HS-5645MG High Torque, Metal Gear Digital Sport Servo | HITEC RCD USA*. URL: https://hitecrcd.com/products/servos/sport-servos/digital-sport-servos/hs-5645mg/product (visited on 09/14/2023).

[6] *Raspberry Pi Documentation - Raspberry Pi hardware*. URL: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html (visited on 09/14/2023).

[7] *Introduction | Navio2*. URL: https://emlid.com//navio2/ (visited on 09/14/2023).

[8] *NEO-M8 – u-blox M8 concurrent GNSS modules*. 74HC4051. Rev. 12. u-blox.

[9] *CALYPSO ULTRASONIC Portable Solar & Mini – Developer Manual*. English version 1.0. CALYPSO Instruments. Mar. 2023.

[10] *Application Note AN3192 – Using LSM303DLH for a tilt compensated electronic compass*. 17353. Rev. 1. STMicroelectronics. Aug. 2010, pp. 24–31.

[11] Jon Melin. "Modeling, control and state-estimation for an autonomous sailboat". In: (2015). ISSN: 1401-5757. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-261553 (visited on 09/19/2023).

[12] Luc Jaulin and Fabrice Le Bars. "An Interval Approach for Stability Analysis: Application to Sailboat Robotics". In: *Robotics, IEEE Transactions on* 29 (Feb. 1, 2013), pp. 282–287. DOI: 10.1109/TRO.2012.2217794.

[13] Roland Stelzer. "Autonomous Sailboat Navigation: Novel Algorithms and Experimental Demonstration". In: (2013). URL: https://api.semanticscholar.org/CorpusID:64183049.

[14] Christophe Viel et al. "Position keeping control of an autonomous sailboat". In: *IFAC-PapersOnLine* 51.29 (2018), pp. 14–19. ISSN: 24058963. DOI: 10.1016/j.ifacol.2018.09.462. URL: https://linkinghub.elsevier.com/retrieve/pii/S2405896318321517 (visited on 09/26/2023).

[15] C. Rizk, L. Mustière, and H. Rangaradjou. *Aston Autonomous Sailboat*. https://gitlab.ensta-bretagne.fr/rangarha/aston-autonomous-sailboat-2023. 2023.

Appendices

Appendix A: Assessment Report

RAPPORT D'EVALUATION
ASSESSMENT REPORT

Merci de retourner ce rapport par courrier ou par voie électronique en fin du stage à :
At the end of the internship, please return this report via mail or email to:

ENSTA Bretagne – Bureau des stages - 2 rue François Verny - 29806 BREST cedex 9 – FRANCE
☎ 00.33 (0) 2.98.34.87.70 / stages@ensta-bretagne.fr

I - ORGANISME / *HOST ORGANISATION*

NOM / *Name* Aston University

Adresse / *Address* Aston Street, Birmingham B4 7ET

Tél / *Phone (including country and area code)* +44 07475104087

Nom du superviseur / *Name of internship supervisor*
Jian Wan

Fonction / *Function* Lecturer in Mechatronics and Robotics

Adresse e-mail / *E-mail address* wanj3@aston.ac.uk

Nom du stagiaire accueilli / *Name of intern* Harendra Rangaradjou

II - EVALUATION / *ASSESSMENT*

Veuillez attribuer une note, en encerclant la lettre appropriée, pour chacune des caractéristiques suivantes. Cette note devra se situer entre **A (très bien)** et **F (très faible)**
*Please attribute a mark from **A (excellent)** to **F (very weak).***

MISSION / *TASK*

- La mission de départ a-t-elle été remplie ? ☑A B C D E F
  Was the initial contract carried out to your satisfaction?
- Manquait-il au stagiaire des connaissances ? ☐ oui/*yes* ☑ non/*no*
  Was the intern lacking skills?

  Si oui, lesquelles ? / *If so, which skills?* _____

ESPRIT D'EQUIPE / *TEAM SPIRIT*

- Le stagiaire s'est-il bien intégré dans l'organisme d'accueil (disponible, sérieux, s'est adapté au travail en groupe) / *Did the intern easily integrate the host organisation? (flexible, conscientious, adapted to team work)*
  ☑A B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here*_____

COMPORTEMENT AU TRAVAIL / *BEHAVIOUR TOWARDS WORK*

Le comportement du stagiaire était-il conforme à vos attentes (Ponctuel, ordonné, respectueux, soucieux de participer et d'acquérir de nouvelles connaissances) ?

7

Version du 05/04/2019

Did the intern live up to expectations? (*Punctual, methodical, responsive to management instructions, attentive to quality, concerned with acquiring new skills)?*

A B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* can be more proactive to solve any issue that occurs

INITIATIVE – AUTONOMIE / *INITIATIVE – AUTONOMY*

Le stagiaire s'est –il rapidement adapté à de nouvelles situations ? A B C D E F
(Proposition de solutions aux problèmes rencontrés, autonomie dans le travail, etc.)

Did the intern adapt well to new situations? A B C D E F
(eg. suggested solutions to problems encountered, demonstrated autonomy in his/her job, etc.)

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* time management can be more efficient

CULTUREL – COMMUNICATION / *CULTURAL – COMMUNICATION*

Le stagiaire était-il ouvert, d'une manière générale, à la communication ? A B C D E F
Was the intern open to listening and expressing himself /herself?

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here*

OPINION GLOBALE / *OVERALL ASSESSMENT*

❖ La valeur technique du stagiaire était : A B C D E F
Please evaluate the technical skills of the intern:

III - PARTENARIAT FUTUR / *FUTURE PARTNERSHIP*

❖ Etes-vous prêt à accueillir un autre stagiaire l'an prochain ?

Would you be willing to host another intern next year? ☑ oui/*yes* ☐ non/*no*

Fait à _____, le _____
In Birmingham, on 05/09/2023

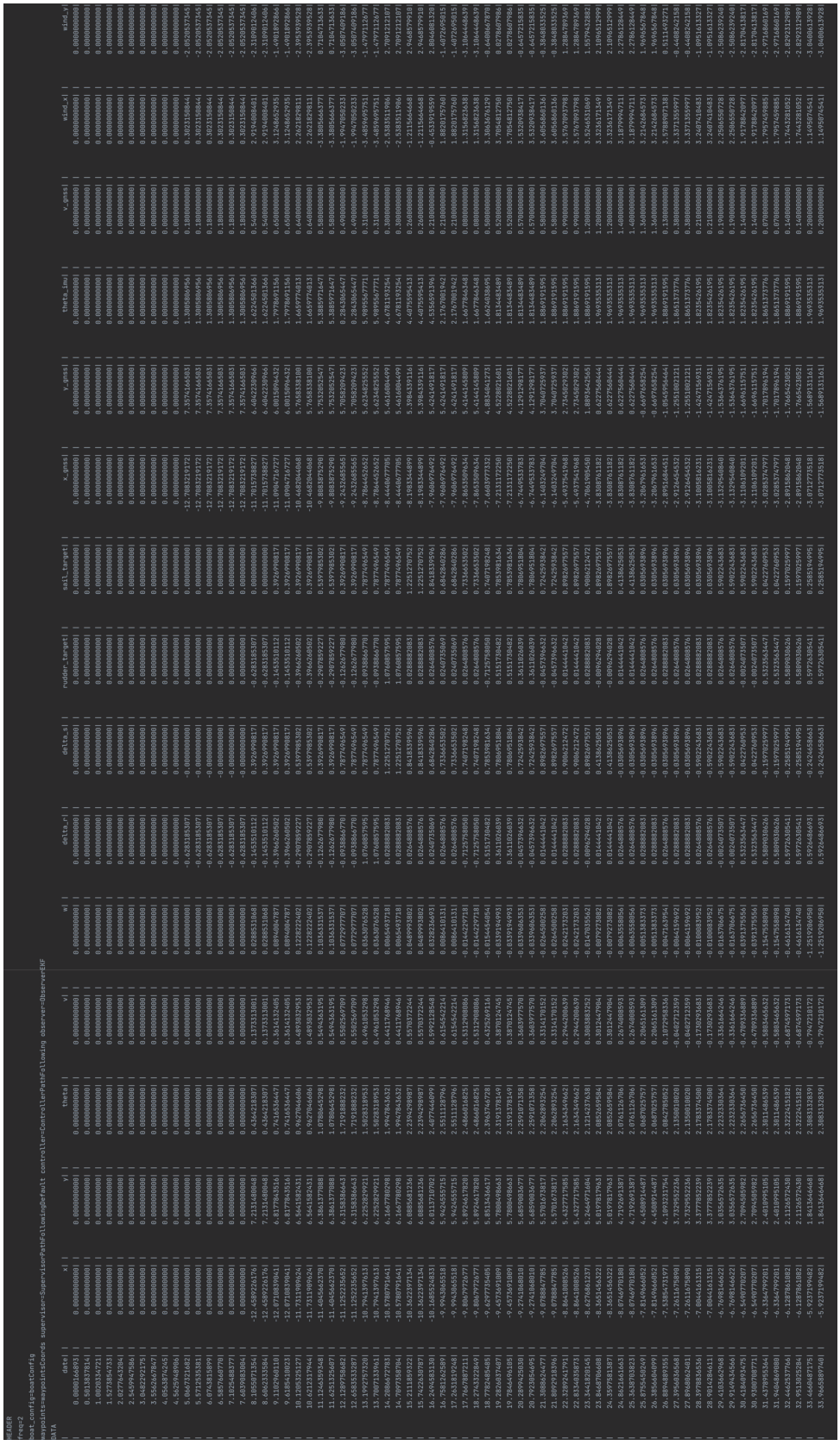
Signature Entreprise _____ Signature stagiaire
Company stamp _____ *Intern's signature*

Merci pour votre coopération
We thank you very much for your cooperation

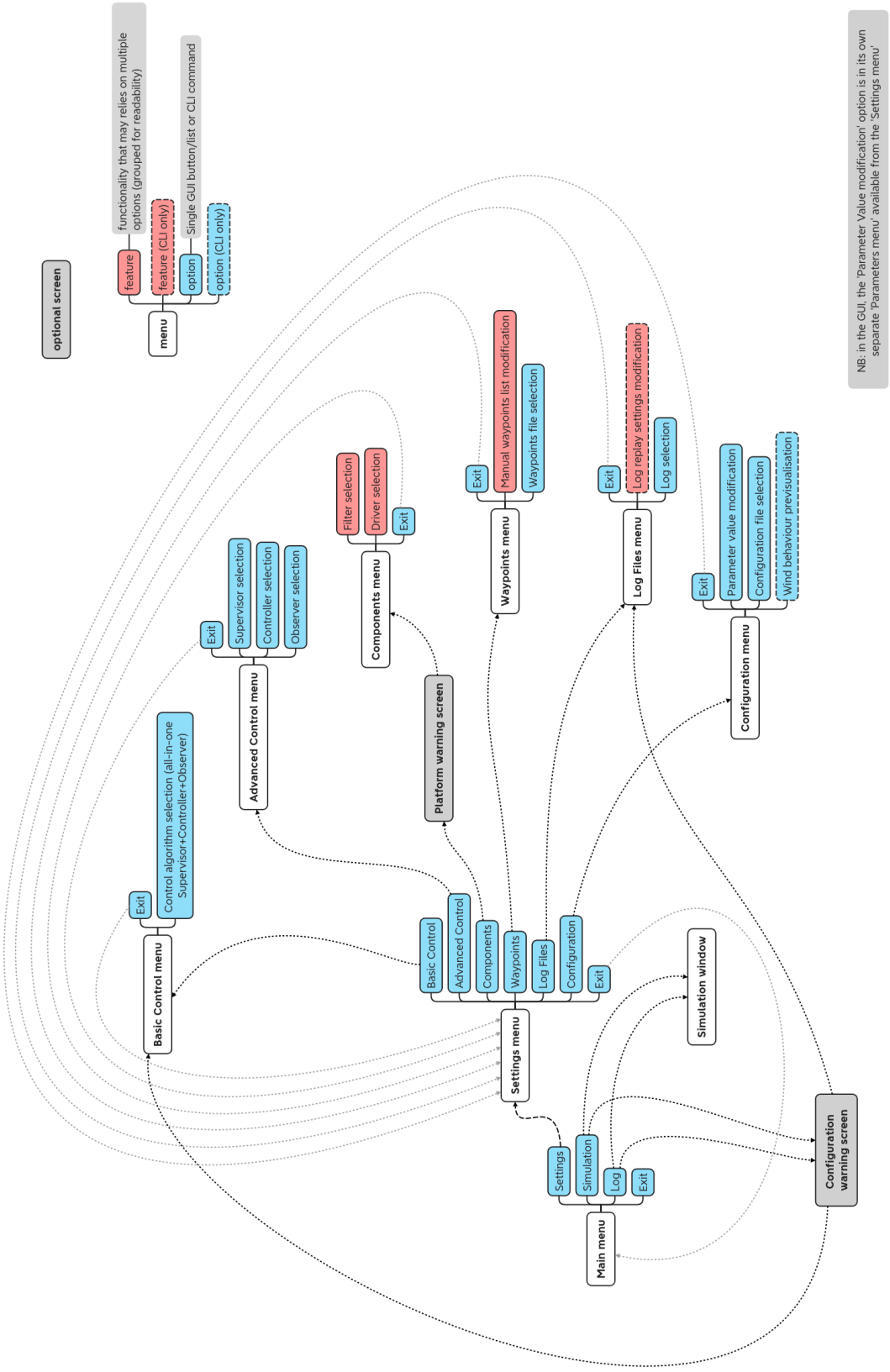
Appendix B: Sample configuration file

```yaml
# Default simulation configuration file (can act both as a simulation configuration file and a boat configuration file)

# Simulation configuration data
SEED_SPAWN: 42                  # Integer used to spawn the seed used for random number generation (defaults to 42)
STATIC: False                   # Whether the wind is static (defaults to False)
AVG_DUR: 40                     # Average duration for which the wind vector should remain sensibly the same (defaults to 40 s)
STD_DUR: 10                     # Standard deviation of the characteristic wind vector evolution duration (defaults to 10 s)
AVG_WIND_NORM: 2                # Average dynamic wind force or static wind force (defaults to 2)
STD_WIND_NORM: 1                # Standard deviation of the wind force (defaults to 1 m/s)
WIND_NORM_INTERP: 'cubic'       # Interpolation method for the norm of the wind vector (defaults to "cubic")
PSI: 2.356194490192345          # Static wind orientation (defaults to 3*pi/4)
STD_PSI: 1.0472                 # Standard deviation of the wind orientation (defaults to pi/3 rad)
PSI_INTERP: 'cubic'             # Interpolation method for the orientation of the wind vector (defaults to "cubic")
BOAT_CONFIG: 'boatConfig'       # Name of the boat configuration file without the extension
AVG_IMU: 0                      # Average noise for the IMU
STD_IMU: 0.3                    # Standard deviation for the IMU
AVG_GNSS_POS: 0                 # Average noise for the position read by the GNSS
STD_GNSS_POS: 0.00001           # Standard deviation for the position read by the GNSS
AVG_GNSS_SPEED: 0               # Average noise for the speed read by the GNSS
STD_GNSS_SPEED: 0.1             # Standard deviation for the speed read by the GNSS
AVG_CALYPSO_WIND: 0             # Average noise for the wind data read by the Calypso
STD_CALYPSO_WIND: 0.2           # Standard deviation for the wind data read by the Calypso
AVG_CALYPSO_TEMPERATURE: 0      # Average noise for the temperature data read by the Calypso
STD_CALYPSO_TEMPERATURE: 1.5    # Standard deviation for the temperature data read by the Calypso
AVG_CALYPSO_POS: 0              # Average noise for the position data read by the Calypso
STD_CALYPSO_POS: 0.3            # Standard deviation for the position data read by the Calypso
X: 10                           # 1st coordinate of the position, 1st component of the state vector
Y: -40                          # 2nd coordinate of the position, 2nd component of the state vector
THETA: 0                        # Heading, 3rd component of the state vector
V: 0                            # Speed, 4th component of the state vector
W: 0                            # Speed rotation, 5th component of the state vector
DELTA_R0: 0                     # Angle for the rudder
DELTA_S0: 0                     # Angle for the sail
T_MAX: 300                      # Maximal duration for the simulation
MODEL_DT: 0.05                  # Time interval used to advance the simulation of the boat model from one iteration to the next
ALGO_DT: 0.1                    # Time step used in the simulation loop to control the rate at which the simulation runs
TIME: 0                         # Initial time of the simulation

# Sailboat configuration data
P0: 0.03                        # Drift coefficient (defaults to 0.03)
P1: 40                          # Tangential friction (defaults to 40)
P2: 6000                        # Angular friction (defaults to 6000)
P3: 200                         # Sail lift (defaults to 200)
P4: 1500                        # Rudder lift (defaults to 1500)
P5: 0.5                         # Distance to sail center of effort (defaults to 0.5)
P6: 0.5                         # Distance to mast (defaults to 0.5)
P7: 2                           # Distance to rudder (defaults to 2)
P8: 300                         # Mass of the boat (defaults to 300)
P9: 400                         # Moment of inertia (defaults to 400)
P10: 0.2                        # Rudder break coefficient (defaults to 0.2)
DELTA_RMAX: 0.6283185307179586  # Maximum angle for the rudder (defaults to pi/5)
BETA: 0.3                       # Angle of the sail in crosswind (defaults to 0.3)
R: 40                           # Cutoff distance (defaults to 40)
INNER_RADIUS: 7                 # Inner radius of the station keeping target (defaults to 7)
OUTER_RADIUS: 14                # Outer radius of the station keeping target (defaults to 14)
XSI: 1.0471975511965976         # Close-hauled angle (defaults to pi/3)
GAMMA : 0.7853981633974483      # Incidence angle (defaults to pi/4)
```

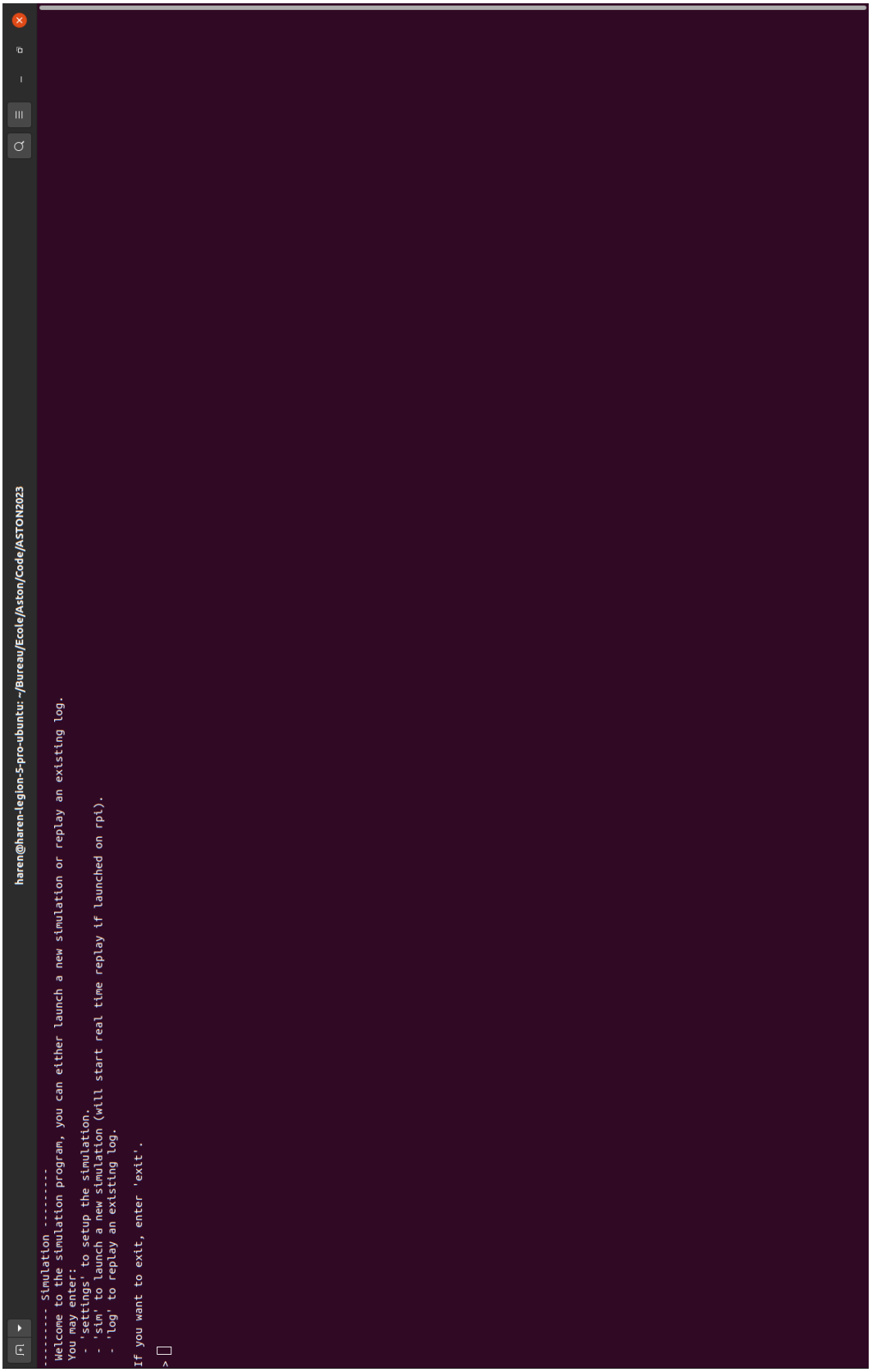
Appendix C: Sample log file



Appendix D: *HMI* structure



Appendix E: Some *CLI* screenshots



haren@haren-legion-5-pro-ubuntu: ~/Bureau/Ecole/Aston/Code/ASTON2023
-------- Components --------
Current component states:
- anemometer: Virtual
- gnss: Virtual
- imu: Virtual
- servomotors: Virtual
- xbee: Virtual
- rc: Virtual
To change a component you can:
- Enter the name of a component to toggle its state
To access the settings of a component:
- Use command '[component] -s [command]' to access the corresponding sensor settings. Possible commands are :
-> 'filter -l' to get information about the filter used by a sensor.
-> 'filter [filter_name]' command to set the filter for a sensor (Use 'filter none' to disable filters).
If you want to exit, enter 'exit'.
>

haren@haren-legion-5-pro-ubuntu: ~/Bureau/Ecole/Aston/Code/ASTON2023
-------- Parameters --------
Current simulation parameter values:
algo_dt: 0.1
avg_calypso_pos: 0
avg_calypso_temperature: 0
avg_calypso_wind: 0
avg_dur: 40
avg_gnss_pos: 0
avg_gnss_speed: 0
avg_imu: 0
avg_wind_norm: 2
boat_config: boatConfig
delta_r0: 0
delta_s0: 0
model_dt: 0.05
psi: 2.3561944901923345
psi_interp: cubic
seed_spawn: 42
static: False
std_calypso_pos: 0.3
std_calypso_temperature: 1.5
std_calypso_wind: 0.2
std_dur: 10
std_gnss_pos: 1e-05
std_gnss_speed: 0.1
std_imu: 0.3
std_psi: 1.0472
std_wind_norm: 1
t_max: 300
theta: 0
time: 0
v: 0
w: 0
wind_norm_interp: cubic
x: 10
y: -40
Current boat parameter values:
beta: 0.3
delta_rmax: 0.6283185307179586
gamma: 0.7853981633974483
inner_radius: 7
outer_radius: 14
p0: 0.03
p1: 40
p10: 0.2
p2: 6000
p3: 200
p4: 1500
p5: 0.5
p6: 0.5
p7: 2
p8: 300
p9: 400
r: 40
xsi: 1.0471975511965976
To modify the configuration you can:
- Load a simulation config file from the following list: ['simConfig']
- Use command 'help {sim, boat}' to get more information about parameters and parameter values
- Use command 'set param=value' to modify configuration parameter values

haren@haren-legion-5-pro-ubuntu: ~/Bureau/Ecole/Aston/Code/ASTON2023
Launching Boat
Starting measurements
Enter 'exit' if you want to return to the menu
Starting INU measurements
Starting GNSS measurements
Starting Calypso measurements
Sensors started
Starting servomotors
Started virtual servo 0
Started virtual servo 1
Servomotors started
Boat ready
0/300 (0%)
3.05/300 (1%)
6.05/300 (2%)

Appendix F: Algorithms

Main algorithm

Algorithm 1 Main control loop algorithm

Require: t_{max}, dt_{loop}

Begin
  while elapsed() $< t_{max}$ **do** ▷ elapsed() returns the time elapsed since launch
    $t_{0,loop} \leftarrow$ elapsed()
    ▷ updateCmd(i) updates actuator commands from the source i
    if rc() **then** ▷ rc() returns whether the rc controller is online
      updateCmd(rc)
    else if serial() **then** ▷ serial() returns whether the serial controller is online
      updateCmd($serial$)
    else
      updateCmd($algo$)
    end if
    updateMeasures() ▷ updateMeasures() updates sensor measurements
    updateEstimate() ▷ updateEstimate() updates the observer state estimate

    $t_{sleep} \leftarrow$ elapsed() $- t_{0,loop}$
    if $dt_{loop} > t_{sleep}$ **then**
      sleep($dt_{loop} - t_{sleep}$) ▷ sleep() pauses the program for the given amount of time
    end if
  end while
End

Controller algorithms

We introduce the parameters used by the various controllers in the table below.

Symbol	Value
r_{lim}	Tacking corridor width (defaults to 40)
r_{in}	Inner radius of the station keeping target (defaults to 7m)
r_{out}	Outer radius of the station keeping target (defaults to 14m)
ξ	Close-hauled angle (defaults to pi/3)
γ	Incidence angle (defaults to pi/4)
q	Hysteresis (Line following only: indicates on which side of the line the boat currently is when tacking)
l	Boolean status flag (Station keeping only: indicates whether the boat is currently following a line to the target (True) or station keeping (False))

Algorithm 2 ControllerLineFollowing algorithm

In: $M, \theta, v, W_1, W_2, a_{tw}, \psi_{tw}$ ▷ The target is the segment $[W_1W_2]$, starting at point W_1
Out: $\delta_{s,max}, \delta_r$
Begin
  $\overrightarrow{W_1M} \leftarrow M - W_1$
  $\overrightarrow{W_1W_2} \leftarrow W_2 - W_1$
  $e \leftarrow \det\left(\frac{\overrightarrow{W_1W_2}}{||\overrightarrow{W_1W_2}||}, \overrightarrow{W_1M}\right)$
  $\theta_{targ} \leftarrow \arg(\overrightarrow{W_1W_2})$
  $\theta_{nom} \leftarrow \theta_{targ} - \frac{2\gamma}{\pi}\arctan2(e, r_{lim})$
  $\psi_{aw} \leftarrow \arctan2(a_{tw}\sin(\psi_{tw} - \theta), a_{tw}\cos(\psi_{tw} - \theta) - v)$
  if $|e| > \frac{r_{lim}}{2}$ **then**
    $q \leftarrow \text{sign}(e)$
  end if
  if $\cos(\psi_{tw} - \theta_{nom}) + \cos(\xi) < 0$ **or** [$|e| < r_{lim}$ **and** $\cos(\psi_{tw} - \theta_{targ}) + \cos(\xi) < 0$] **then**
    $\theta_{real} \leftarrow \pi + \psi_{tw} - q\xi$ ▷ headwind $\implies$ tacking
  else
    $\theta_{real} \leftarrow \theta_{nom}$ ▷ tailwind
  end if
  if $\cos(\theta - \theta_{real}) \geq 0$ **then**
    $\delta_r \leftarrow \delta_{r,max}\sin(\theta - \theta_{real})$
  else
    $\delta_r \leftarrow \delta_{r,max}\,\text{sign}(\sin(\theta - \theta_{real}))$
  end if
  $\delta_{s,max} \leftarrow -\text{sign}(\psi_{ap})\min\left(|\pi - |\psi_{ap}||, \frac{\pi}{4}(\cos(\psi_{tw} - \theta_{real}) + 1)\right)$
End

Algorithm 3 ControllerStationKeeping algorithm

In: $M, \theta, v, W, a_{tw}, \psi_{tw}$ ▷ The target is waypoint W
Out: $\delta_{s,max}, \delta_r$
Begin
  $\overrightarrow{WM} \leftarrow M - W$
  $\theta_{targ} \leftarrow \arg(\overrightarrow{WM})$
  $d_{targ} \leftarrow ||\overrightarrow{WM}||$
  $\psi_{aw} \leftarrow \arctan2(a_{tw}\sin(\psi_{tw} - \theta), a_{tw}\cos(\psi_{tw} - \theta) - v)$
  if $d_{targ} > r_{out}$ **then**
    if not l **then**
      $A \leftarrow M$
      $l \leftarrow True$
      $\delta_r, \delta_{s,max} \leftarrow ControllerLineFollowing(M, \theta, v, A, W, a_{tw}, \psi_{tw})$
    end if

else
  if l **then**
    $l \leftarrow False$
  end if
  if $d_{targ} > r_{in}$ **then**
    if $\cos(\theta_{targ} - \psi_{tw}) > 0$ **then**
      if $\cos(\pi + \theta_{targ} - \frac{\pi}{2} - \psi_{tw}) > \cos(\pi + \theta_{targ} + \frac{\pi}{2} - \psi_{tw})$ **then**
        $\theta_{targ} \leftarrow \theta_{targ} + \frac{\pi}{2}$
      else
        $\theta_{targ} \leftarrow \theta_{targ} - \frac{\pi}{2}$
      end if
    end if
    if $\cos(\psi_{tw} - \theta_{targ}) + \cos(\xi) < 0$ **then**
      if $\sin(\psi_{tw} - \theta_{targ}) > 0$ **then**
        $q \leftarrow 1$
      else
        $q \leftarrow -1$
      end if
      $\theta_{real} \leftarrow \psi_{tw} + \pi + q\xi$ ▷ motion heading including drift
    else
      $\theta_{real} \leftarrow \theta_{targ}$
    end if
  else
    if $\cos(\psi_{tw} - \theta_{targ}) + \cos(\xi) > 0$ **then**
      if $\cos(\theta_{targ} - \psi_{tw} - \pi + \xi) > \cos(\theta_{targ} - \psi_{tw} - \pi - \xi)$ **then**
        $q \leftarrow -1$
      else
        $q \leftarrow 1$
      end if
      $\theta_{real} \leftarrow \psi_{tw} + \pi + q\xi$
    else
      $\theta_{real} \leftarrow \theta_{targ}$
    end if
  end if
  if $\cos(\theta - \theta_{real}) \geq 0$ **then**
    $\delta_r \leftarrow \delta_{r,max} \sin(\theta - \theta_{real})$
  else
    $\delta_r \leftarrow \delta_{r,max} \operatorname{sign}(\sin(\theta - \theta_{real}))$
  end if
  $\delta_{s,max} \leftarrow -\operatorname{sign}(\psi_{ap}) \min\left(|\pi - |\psi_{ap}||, \frac{\pi}{4}(\cos(\psi_{tw} - \theta_{real}) + 1)\right)$
end if
End

Algorithm 4 ControllerViel algorithm

In: $M, \theta, v, W, a_{tw}, \psi_{tw}$ ▷ The target is waypoint W
Out: $\delta_{s,max}, \delta_r$
Begin
  $\overrightarrow{WM} \leftarrow M - W$
  $\theta_{targ} \leftarrow \arg(\overrightarrow{WM})$
  $d_{targ} \leftarrow ||\overrightarrow{WM}||$
  $\psi_{aw} \leftarrow \arctan2(a_{tw}\sin(\psi_{tw} - \theta), a_{tw}\cos(\psi_{tw} - \theta) - v)$
  if $d_{targ} > r_{out}$ **then**
    if not l **then**
      $A \leftarrow M$
      $l \leftarrow True$
      $\delta_r, \delta_{s,max} \leftarrow ControllerLineFollowing(M, \theta, v, A, W, a_{tw}, \psi_{tw})$
    end if
  else
    if l **then**
      $l \leftarrow False$
    end if
    if $d_{targ} \geq r_{in}$ **then**
      if $\cos(\theta_{targ} - \psi_{tw}) > 0$ **then**
        if $\cos(\theta_{targ} - (\psi_{tw} + \frac{\pi}{2})) > \cos(\theta_{targ} - (\psi_{tw} - \frac{\pi}{2}))$ **then**
          $\theta_{targ} \leftarrow \theta_{targ} + \frac{\pi}{2}$
        else
          $\theta_{targ} \leftarrow \theta_{targ} - \frac{\pi}{2}$
        end if
      end if
      if $\cos(\psi_{tw} - \theta_{targ}) + \cos(\xi) > 0$ **then**
        if $\cos(\theta_{targ} - (\psi_{tw} + \pi - \xi)) > \cos(\theta_{targ} - (\psi_{tw} + \pi + \xi))$ **then**
          $q \leftarrow 1$
        else
          $q \leftarrow -1$
        end if
        $\theta_{real} \leftarrow \psi_{tw} + \pi + q\xi$ ▷ motion heading including drift
      else
        $\theta_{real} \leftarrow \theta_{targ}$
      end if
      $\delta_{s,max} \leftarrow -\text{sign}(\psi_{ap}) \min\left(|\pi - |\psi_{ap}||, \frac{\pi}{4}(\cos(\psi_{tw} - \theta_{real}) + 1)\right)$
    else if $v > 0$ **then**
      $\theta_{real} \leftarrow \psi_{tw} + \pi$
      $\delta_{s,max} \leftarrow -\text{sign}(\psi_{ap})|\pi - |\psi_{ap}||$
    else
      $\theta_{real} \leftarrow \psi_{tw} + \pi - \xi\,\text{sign}(v)$
      $\varepsilon_{\delta_s} \leftarrow 5$
      $\delta_{s,opti} \leftarrow \frac{\pi}{4}(\cos(\psi_{tw} - \theta_{real}) + 1)$

$\delta_{s,M} \leftarrow \min(|\pi - |\psi_{ap}||, \frac{\pi}{2})$
$\delta_{s,lim} \leftarrow \max(\delta_{s,M} - \varepsilon_{\delta_s}, 0)$
$\delta_{s,max} \leftarrow -\operatorname{sign}(\psi_{ap}) \min(\delta_{s,lim} - |\delta_{s,opti}|)$
end if
if $\cos(\theta - \theta_{real}) > 0$ **then**
$\delta_r \leftarrow \delta_{r,max} \sin(\theta - \theta_{real})$
else
$\delta_r \leftarrow \delta_{r,max} \operatorname{sign}(\sin(\theta - \theta_{real}))$
end if
end if
End

Appendix G: Kalman filter implementation

```
def _buildLambda(self, forceCompute: Optional[bool] = False):
    """Initializes all the functions used by the observer. Computes the evolution matrix using symbolic calculus or
    by unpickling previously computed results.

    Args:
        forceCompute (bool): Whether to forcefully run/rerun symbolic calculations
    """

    if not forceCompute:
        print("Loading lambda functions")
    try:
        with open(abspath(getcwd()) + "/Config/lambdaKalman", "rb") as file:
            self.aMatrix = load(file)
        if not forceCompute:
            print("Successfully retrieved pickled data")

    except (FileNotFoundError, IOError, EOFError):
        if not forceCompute:
            print("Failed to retrieve pickled data. Rebuilding missing objects")
        x, y, theta, v, w = symbols("x y theta v w")
        aWind, psiWind, delta_r, delta_s = symbols("aWind psiWind delta_r delta_s")
        X = Matrix([x,
                    y,
                    theta,
                    v,
                    w])
        u = Matrix([delta_r,
                    delta_s])

        w_ap = Matrix([aWind * sym_cos(psiWind - theta) - v,
                       aWind * sym_sin(psiWind - theta)])
        psi_ap = sym_atan2( *args: w_ap[1, 0], w_ap[0, 0])
        a_ap = (w_ap[1, 0] ** 2 + w_ap[0, 0] ** 2) ** 0.5

        gr_ = self.boat.config.p4 * v * v * sym_sin(delta_r)
        gs_ = self.boat.config.p3 * a_ap * sym_sin(delta_s - psi_ap)

        dX = X + self.dt * Matrix([v * sym_cos(theta) + self.boat.config.p0 * aWind * sym_cos(psiWind),
                                   v * sym_sin(theta) + self.boat.config.p0 * aWind * sym_sin(psiWind),
                                   w,
                                   (gs_ * sym_sin(delta_s) - gr_ * self.boat.config.p10 * sym_sin(
                                       delta_r) - self.boat.config.p1 * v * v) / self.boat.config.p8,
                                   (gs_ * (self.boat.config.p5 - self.boat.config.p6 * sym_cos(
                                       delta_s)) - self.boat.config.p7 * gr_ * sym_cos(
                                       delta_r) - self.boat.config.p2 * w * v) / self.boat.config.p9])

        aMatrix = dX.jacobian(X)
        aMatrix.simplify()
        self.aMatrix = lambdify( args: [X, u, aWind, psiWind], aMatrix, modules=sympy)
        if not forceCompute:
            print("Done rebuilding")

        if not forceCompute:
            print("Dumping newly built objects")
        dill_Settings['recurse'] = True
        with open(abspath(getcwd()) + "/Config/lambdaKalman", "wb") as file:
            dump(self.aMatrix, file)
        if not forceCompute:
            print("Successfully dumped newly built objects")
```

```python
def updateEstimate(self, u: ndarray, wind: ndarray, predict: Optional[bool] = True):
    """Updates the estimated state vector of the boat

    Args:
        u (ndarray): System command vector
        wind (ndarray): Apparent wind vector
        predict (bool): whether to estimate the boat state by using prediction and correction or simply by merging sensor data
    """
    try:
        if predict:
            self.boat.u = u
            aMatrix = array(self.aMatrix(self.xHat.flatten(), u.flatten(), norm(wind), arg(wind))).astype(float64)

            k1 = self.model(wind=self.boat.wind)

            x_ = self.boat.x + self.dt * k1
            w_ = self.boat.supervisor.get_target(x_)
            wind_ = self.boat.wind + (self.boat.wind - self.boat.windMemory)
            u_ = self.boat.controller.get_cmd(x_, w_, wind_)
            k2 = self.model(x=x_, u=u_, wind=wind_)

            v = self.xHat + self.dt * (0.7 * k1 + 0.3 * k2) - aMatrix @ self.xHat

            self.xHat, self.gammaHat = kalmanPredict(self.xHat, self.gammaHat, v, self.gammaAlpha, aMatrix)

            # Overflow protection
            if any([isnan(x) for x in self.xHat]):
                raise OverflowError("Overflow encountered in the kalman filter prediction")
            elif norm(self.xHat) > 1e9:
                raise OverflowError("Overflow protection triggered by the kalman filter prediction")
        else:
            self.gammaHat = array([[1e3, 0, 0, 0, 0],
                                   [0, 1e3, 0, 0, 0],
                                   [0, 0, 10, 0, 0],
                                   [0, 0, 0, 100, 0],
                                   [0, 0, 0, 0, 10]])

        self.xHat, self.gammaHat = kalmanCorrect(self.xHat, self.gammaHat, deepcopy(self.boat.y), self.gammaBeta,
                                                 self.cMatrix)
    except LinAlgError:
        warning("Encountered singular matrix in observer")

    except OverflowError as e:
        if "kalman filter" in e.args[0]:
            warning(f"Encountered error: '{e}'.\nPerforming emergency observer reset")
            self.reset(build=False)
        else:
            print(e.args)
            raise
```

Appendix H: README files

README.md

ASTON University Autonomous Sailboat 2023

Requirements

You need to use :

- Python 3.7+

Python Libraries 📚

In order launch and use this project, you need these python libraries. Please follow the setup instructions :

- Navio2
- Numpy (1.*.*)
- Matplotlib (3.*.*)
- Calypso anemometer (0.6.0)

❗ **Navio2 library** was built for Python2. But it can be easily **fixed** 🔨

You just need to **follow** those steps :

- You first need the access to modify the library. Open a terminal and use this command:

```
sudo chown -R username path
```

It should look something like that:

```
sudo chown -R pi /usr/local/lib/python3.7/dist-packages/navio2/
```

- Open the library in you favorite IDE and find the file called `lsm9ds1.py`
- Use Ctrl + F and search for `xrange`. Now replace all its occurences by `range`.

Congrats 👏 You are now ready to work with navio2 library.

Files

Hardware Setup

Informations

RaspberryPi Username : pi

RaspberryPi Password : raspberry

Setup SSH connection between the computer and the RaspberryPi

In order to **communicate** between your computer and the RasperryBi, you need to set up an SSH connection between them. Here is how it can be achieved:

1. Connect your computer and the RaspberryPi to the same **Wireless** LAN. It can be a phone Wifi Hotspot.
2. Find the RaspberryPi IP Adress. You can either scan the Wifi with on your computer with:

```
nmap -F -n -sS 127.0.0.1
```

Or get the IP adress from your RaspberryPi with:

```
hostname -I
```

3. Start the connection. You then need to enter the **password** of the RaspberryPi.

```
ssh pi@192.168.43.178
```

Tips 🍾

Useful fixes

Use your IDE on raspberryPi with SSH

VSCode

You first need to install two extensions from the marketplace:

- Remote - SSH
- Remote - SSH: Editing Configuration Files

You can then follow this tutorial to setup it correctly (It's very easy and last less than 5 minutes 😉)

It also allows you to install any extension you would like directly on the pi.

Pycharm (Professional Edition)

Follow this tutorial : https://www.jetbrains.com/help/pycharm/configuring-remote-interpreters-via-ssh.html

Roadmap and Ideas

[] Create a config file for the servomotors (.json or .yaml) [] Use Mutex with 'with' syntax [] input non blocking simulation [] Choose between different config files [] Save config files [] Return to default config file [] Add security to interface

Bluetooth info

sudo visudo

https://wiki.archlinux.org/title/bluetooth

load btusb module Start/enable bluetooth.service (sans utiliser sudo)

Launch bluetooth : systemctl start bluetooth Check status : sudo systemctl status bluetooth

Bluetooth issues

When we encountered the issues described in this post: https://raspberrypi.stackexchange.com/questions/47045/hcitool-scan-says-no-such-device, (i.e. no bluetooth controller detected), editing the /boot/config.txt and commenting the line:

```
dtoverlay=pi3-disable-bt
```

and rebooting the RPi solved our problems

Ou lancer les prog

Tout lancer depuis ASTON2023 cf aliases

```
. .
```

and rebooting the RPi solved our problems

Ou lancer les prog

Tout lancer depuis ASTON2023 cf aliases

README.md

Algorithms

This folder contains the programs which define the missions that the boat can undertake as well as the behaviours that allow it must adopt to complete them.

Supervision

Supervisors are the highest level form of control the program has over the boat. They define the mission to accomplish and set abstract instrumental targets to achieve it.

Implemented supervisors:

- SupervisorLineFollowingDefault
- SupervisorStationKeepingDefault
- SupervisorPathFollowing

Control

Controllers are a lower level form of control the program has over the boat. They act upon individual actuators and ensure that they behave as expected to reach the target set by the supervisor. In our case the controllers pilot the servomotor angles of the rudder and the sail.

Implemented supervisors:

- ControllerLineFollowing
- ControllerStationKeeping
- ControllerPathFollowing
- ControllerViel

Observation

Observers are used by the boat during missions to estimate its state (e.g. using the feedback from its sensors or a predictive approach).

Implemented observers:

- ObserverEKF

Filtering

The Filter class can be used by any sensor, it is a generic moving median filter capable of filtering the evolution of one variable at the time, meaning that as many filter as there are variables to filter need to be declared.

README.md

CONFIG

This folder contains the configuration files for the project. It allows the user to change the parameters of the project without having to change the code. It is especially useful when you are testing at the lake.

boatConfig.yaml

This file contains the contains the configuration of the **real** sailboat. You can find inside the close-hauled angle or cutoff distance.

simConfig.yaml

This file contains the contains the configuration of the **simulated** sailboat. You can find inside the random generated seed, the noise parameters for each sensor and the wind parameters.

.defaultConfig.yaml

This file contains the default configuration of the project. It is used to generate the `boatConfig.yaml` and `simConfig.yaml` files. You should not modify this file. It should also not be used by the project and only serve as a backup.

Waypoints

This folder contains two files dedicated to the waypoints. The first one is the `waypoints.csv` file. It contains the waypoints that the boat will have to follow in cartesian coordinates. The seccond one is the `waypointsCoords.csv` file. It contains the waypoints that the boat will have to follow in latitude and longitude.

README.md

Drivers

The folder driver is designed to be used as a **package**. It contains all the **sensor** inside the sailboat and test files for each of them. Furthermore, it also contains the **Sailboat class**. This class was created to easily use all sensors simultaneously.

Project Architecture

```
Drivers
├── Calypso
│   ├── README.md
│   ├── __init__.py
│   ├── calypso.py
│   └── test_calypso.py
├── Gnss
│   ├── README.md
│   ├── __init__.py
│   ├── gnss.py
│   └── test_GNSS.py
├── Imu
│   ├── README.md
│   ├── __init__.py
│   ├── accelerometer_calibration.py
│   ├── imu.py
│   ├── magnetometer_calibration.py
│   ├── magnetometer_data_3d.csv
│   ├── magnetometer_data_circle.csv
│   ├── magnetometer_get_data.py
│   ├── magnetometer_plot.py
│   └── test_imu.py
├── Servomotors
│   ├── README.md
│   ├── __init__.py
│   ├── servomotors.py
│   └── test_servomotors.py
├── Xbee
│   ├── README.md
│   ├── __init__.py
│   ├── test_XBEE.py
│   └── xbee.py
├── README.md
└── __init__.py
```

Imports

The use of `__init__.py` file allows us to use this folder as a package (see Python documentation for more informations). That way, imports are shortened and much clearer.

README.md

SCRIPTS

This folder contains all the **bash/zsh scripts** needed to launch the programs. It also contains a small script that you can run to modify the aliases and add shortcuts to the Raspberry Pi.

Aliases 🖊️

This file contains aliases that works as shortcuts for the Raspberry Pi. To add them, you have to run run the script once with:

```
./home/pi/ASTON2023/Scripts/add_aliases.bash
```

Then, you **NEED** to reboot the Raspberry Pi to make the changes effective.

Sailboat 🚤

This file contains the script that launches the sailboat program. It starts the **bluetooth service** and initialize the permissions for the **PWM pins**. Then, it launches the program with the arguments given to the script.

Test 🧪

This file contains the script that launches the test program. It starts the bluetooth service and initialize the permissions for the PWM pins. Then, it launches the program with the arguments given to the script.

Shortcuts 📌

Here is the list of all shortcuts that you can use with the current aliases:

- *testCalypso* : Test the Calypso and prints its data
- *testIMU* : Test the IMU and prints its data
- *testGNSS* : Test the GNSS and prints its data
- *testServo* : Test the servo and moves it
- *testIMUFilter* : Test the IMU filter and prints its data
- *testXBEE* : Unused for now
- *testRC* : Unused for now
- *launch* : Launch the sailboat program
- *calibAcc* : Calibrate the accelerometer
- *calibMag* : Calibrate the magnetometer

README.md

SIMULATION

This folder contains the simulation program. It is mainly used to **test the controller** we want to use but we also added a log replay functionnality and a way to test the sensor while being in the simulation.

Imports

This code is using the following libraries :

- Numpy (1..)
- Scipy
- Tkinter
- Pygubu
- Matplotlib (3..)

GUI

We added a **user interface** to the simulation for a more friendly use. For more information, read this README.md.

Simulation

The simulation is based on a 2D model of the sailboat. It uses the simConfig.yaml file. This file contains all the parameters of the simulation. The model only lacks the filters functionnality.

Model

The model is based on the following equations in Modeling, control and state-estimation for an autonomous sailboat by **Jon Melin**.

README.md

Utils

This folder contains useful functions and utilities used throughout the entire code.

config.py

The Config and ConfigOverlay classes can be used in conjunction with .yaml configuration files to create configuration templates. The .yaml files must be placed in the Config folder to be detected.

logs.py

The Log class can produce mission log for both real and simulated missions. The log files are output in the Log folder.

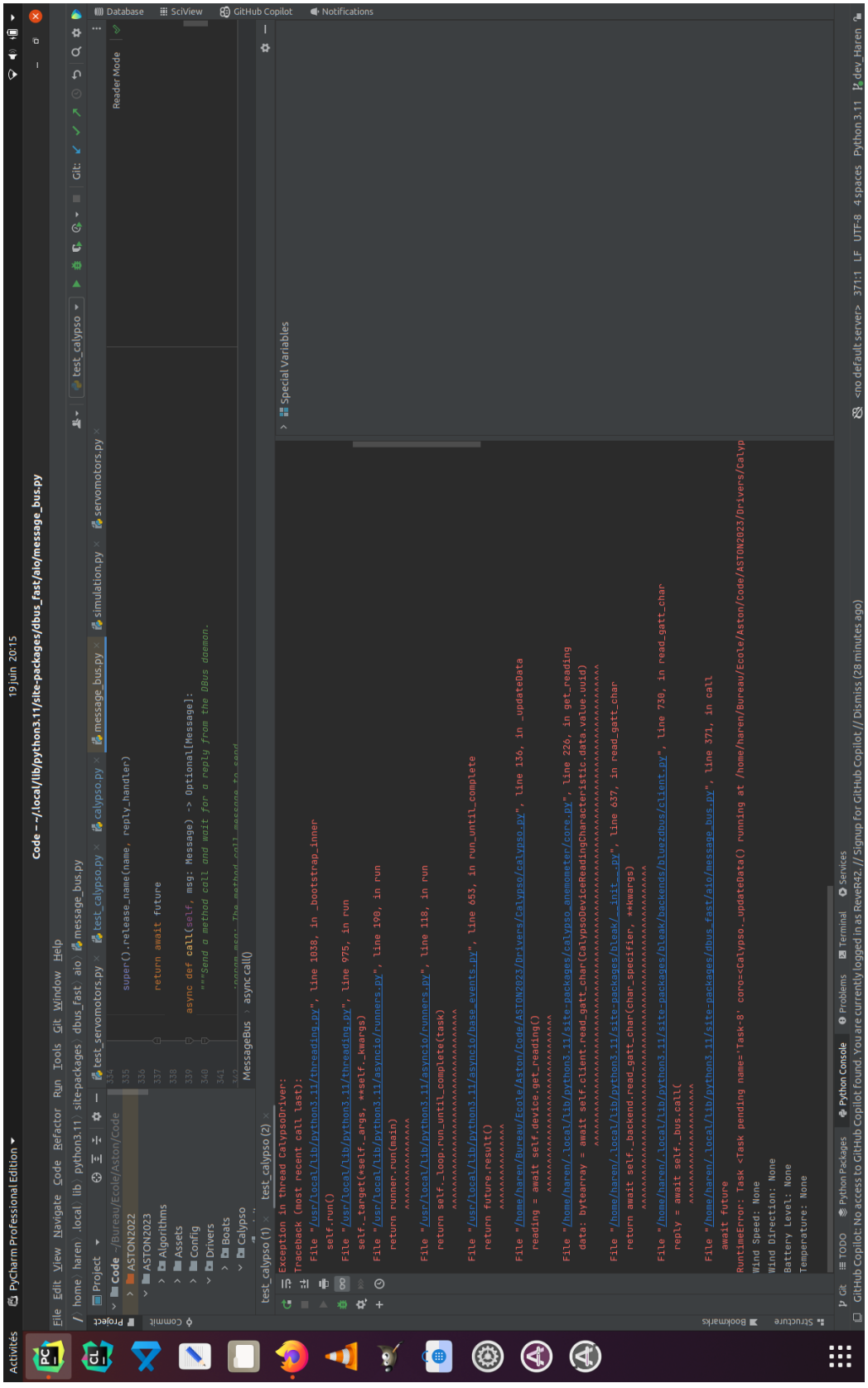
utils.py

This file centralizes all the miscellaneous utilities. It contains useful roblib functions, matplotlib utilities, waypoint/log file processing utilities, useful settings and coordinate projection utilities.

autocomplete.py

This is a work in progress helper for the cli. It is meant to provide a custom user prompting interface that could be used instead of the default python 'input()' function and which would allow for real-time autocompletion/help

Appendix I: Anemometer driver investigation



Code – ~/.local/lib/python3.11/site-packages/dbus_fast/aio/message_bus.py
19 juin 21:03
PyCharm Professional Edition
File Edit View Navigate Code Refactor Run Tools Git Window Help
Database SciView GitHub Copilot Notifications
test_calypso
test_servomotors.py
calypso.py
pydevconsole.py
core.py
__init__.py
client.py
message_bus.py
simulation.py
servomotors.py
Reader Mode
Project
Code ~/Bureau/Ecole/Aston/Code
ASTON2022
ASTON2023
Algorithms
Assets
Config
Drivers
Boats
Calypso
__init__.py
calypso.py
README.md
test_calypso.py
Gnss
Imu
Servomotors
Virtual
Xbee
__init__.py
Logs
Scripts
Simulation
__init__.py
paramMF.mat
Special Variables
MessageBus async call()
test_calypso (8) test_calypso (9)
Creating new event loop
True
Starting measurements...
Connection established Calypso
Calypso anemometer initialised.
Battery Level: None
Temperature: None
Roll: None
Pitch: None
Heading: None
Exception in thread CalypsoDriver:
Wind Speed: None
Wind Direction: None
Python Console
Problems
Terminal
Services
Python Packages
TODO
Git
Structure
Bookmarks
GitHub Copilot: No access to GitHub Copilot found. You are currently logged in as ReveR42. //Signup for GitHub Copilot //Dismiss (today 19:47)
<no default server> 360:1 LF UTF-8 4 spaces Python 3.11 dev_Haren