Master Systèmes Dynamiques et Signaux

Mémoire

Development of Localization & Control Algorithms for an Autonomous ROV

Auteur :
M. Hugo Yverneau

Jury :
Pr. L. Hardouin
Pr. L. Jaulin

Version du
24 août 2023

Résumé

Le processus d'automatisation d'un Remotely Operated underwater Vehicle (ROV) repose sur deux blocs logiciels : la localisation et le contrôle. D'une part, le bloc logiciel de localisation est chargé de filtrer toutes les informations pour déterminer l'état du robot, tandis que le bloc de commande est chargé d'utiliser l'état estimé du robot pour contrôler ses actionneurs. Pour répondre à ces questions, ce rapport de Projet de Fin d'Étude (PFE) sur le développement d'algorithmes de localisation et de contrôle pour un ROV autonome se penche sur différents aspects du problème. Les questions de localisation sont abordées avec la mise en œuvre d'un algorithme de fusion de données de haut niveau utilisant un graphe de contraintes et avec le développement de bas niveau d'un pilote de sonar. Les aspects de contrôle sont abordés en modifiant l'architecture des contrôleurs du ROV.

Abstract

The automation process for a Remotely Operated underwater Vehicle (ROV) is based on two software blocks: localization and control. On the one hand, the localisation software block is responsible for filtering all the information to determine the state of the robot, while the control block is responsible for using the estimated state of the robot to control its actuators. To address these issues, this end-of-study project report on the development of localisation and control algorithms for an autonomous ROV delves into different aspects of the problem. Localization issues are addressed with the implementation of a high-level data fusion algorithm using a constraint graph and with the low-level development of a sonar driver. Control aspects are addressed by changing the architecture of the ROV controllers.

Acknowledgements

I would like to thank Gautier Dreyfus for making this great work-study opportunity possible; David Barral, Mathieu Mege and the whole team at Forssea in Sète for their warm welcome, their availability and their good humour; my tutor Auguste Bourgois for his guidance and his time. My thanks also go to Pierre Narvor for its great help on the sonar comprehension and for Paris Forssea team for their occasional but no less useful assistance.

I can't finish without specific thanks to Mathieu Mege for his outspokenness, to Victor Buchet for his crazy lunch plans and Felix Nilius for his teaching of electricity, electronics and, of course, speleology.

Keywords

ROV, sonar, sensor fusion, constraint graph optimization, ROS, chained controllers

Contents

1 Introduction

PFE at ENSTA Bretagne

Projet de Fin d'Étude (PFE) is the graduation project performed in a company or laboratory during the last semester of ENSTA Bretagne engineering course. In my case, this project has a strong research component, as part of the research master's degree in dynamic systems and signals at the University of Angers. This report and the associated defense constitute the final assessments leading to the granting of both diplomas.

The format of the one-year alternance study allowed me to precede my full seven-month period in the company with five months of part-time work on a Forssea-related project. This first project was an acoustic Simultaneous Localization and Mapping (SLAM). As this was no longer a priority for Forssea, I did not continue to work on this project except indirectly within the sonar driver context. I will therefore not develop this work further in this report.

Forssea Robotics

Forssea Robotics is a small French tech company with the ambition of providing autonomous industrial subsea robotics solutions for the offshore industry. Founded in 2016, Forssea now has around 25 employees, the company has a workshop in Sète (Hérault) and research and development is based in both Sète and Paris. Forssea mainly designs, develops and manufactures two products, NavCam[1] and Argos[2]. The NavCam is a subsea relative positioning camera with QR code. And Argos is an Remotely Operated underwarter Vehicle (ROV) designed to provide piloting assistance and autonomy features. Forssea's customers are mainly offshore energy industrialists, for example in the oil and wind power sectors companies (such as Perenco, DeepOcean or EDF), but also organizations with more specific applications such as deep-sea scientific research or mine clearance (such as Ifremer or Geomines).

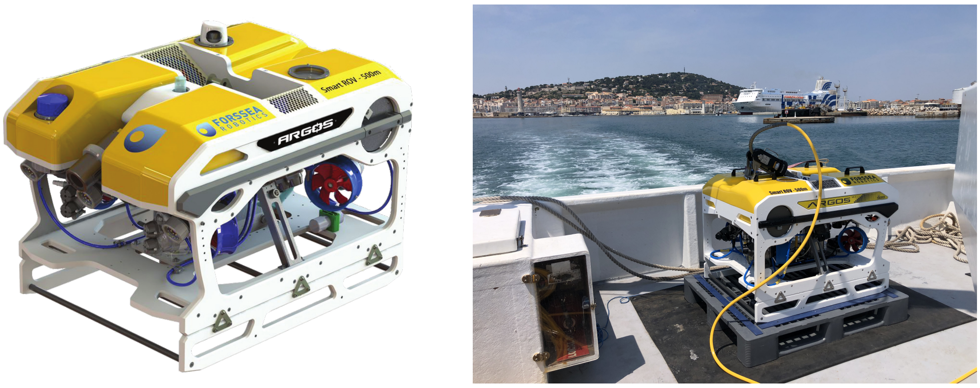


Figure 1.1: Argos

[1] Further information about NavCam can be found at https://forssea-robotics.fr/smart-cameras

[2] Further information about Argos can be found at https://forssea-robotics.fr/smart-rov

Subsea Robotics

Subsea robotics is a very specific field of robotics. Indeed, the underwater world is a highly constraining environment. Alongside the mechanical constraints of pressure resistance and watertightness, due to the strong dispersion of electromagnetic waves, communication with robots and accurate positioning are major challenges at the heart of today's scientific research.

To solve the communication issue, submarine robotics has seen widespread use of ROVs for several decades, without any significant evolution. However, umbilical cable management is a major problem during ROV operations. In addition to hindering the ROV's movements, the umbilical cable represents a major risk of jamming and thus blocking the ROV. This can result in damage to the ROV itself or to surrounding equipment. While umbilical cable integrity is also vital for recovering the ROV at the end of the operation.

As Global Navigation Satellite System (GNSS) satellite constellations are not reachable, localization in subsea robotics is mostly based on inertial localization with Fiber Optic Gyroscopes (FOG) Inertial Naviagation System (INS). INS are costly sensors that offer high short-term accuracy but, like all dead reconing methods, suffer drift issues over time. In the main applications, sensor fusio (e.g. FOG INS, Doppler Velocity Log (DVL), pressure sensor and GNSS at the surface) is performed in the INS-integrated software by highly efficient Kalman filter-based methods.

At present, ROVs are the unavoidable protagonists of underwater robotics. However, we can see AUVs gaining in maturity and attempting to position themselves as the future of the sector.

PFE Unfolding

This end-of-study project report focuses on the control and localization problems of Forssea's ROV: Argos. At the beginning of my project, the ROV was in a functional state, so my mission was to participate in the design and implementation of new functionality. My work was divided into three main areas, the implementation of a ROS driver for a sonar so that it could be used to perform an acoustic SLAM, the optimisation of the structure of the control blocks and the implementation of a localization system using a constraint graph to filter the sensor data.

Sonar As a result of the work carried out on creating an acoustic SLAM, it became apparent that the ROS 2 driver used was not fully functional. I therefore worked on completing this driver, making sure that it met Forssea's requirements. Tests were also carried out to ensure that the sonar worked with its driver.

Chained Controllers In parallel with the sonar work, I worked on the Forssea control framework to change the architecture of the control blocks. I optimised the framework based on the ROS 2 Control chainable controller style. This means that each controller can now be changed independently, which is of direct interest for the Research and Development (R&D) phases but also increases the efficiency and reliability of the code.

Sensors Filter The last and most important part of my work was to use a constraint graph to filter sensor data in order to determine the state of the ROV. Forssea doesn't currently do any filtering. I picked up Forssea's work that had been put on hold because of day-to-day priorities. It was decided to adapt a constraint graph framework to Forssea's needs. My first task was to debug the code that had been passed to me to allow the data pipeline to run. I then implemented a simple 1D constraint generated by the ROV's pressure sensor. Next, a 3D constraint induced by the DVL. I was able to rely on constraints on the evolution of physical systems that were already implemented in our use case.

Lie Theory A small part of my work was also to obtain equations to get the covariance matrix modified by an odometry correction. This work having been done at the end of the writing of this report and not having succeeded, I mainly brought myself up to the required level by learning Lie theory [9, 15].

2 Constraint Graph Optimization

To estimate the robot localization various algorithms exist in the state of the art: Kalman filter based methode (as Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF)), particles filter or constraints graph approch (as set-membership method) are the more common methods used. No method has outperformed the others, they all have their assets and their drawbacks. For now, at Forssea, we rely on the INS's Kalman filter based fusion software, if the INS is mounted on the ROV, otherwise the data of the embeded sensors are used without any fusion. The aim of performing a fusion is to integrate every sensor without being tied by the INS capability (for example intergrate home made sensor as NavCam) and provide more numerous and more accurate data on our FOG INS agnostic ROVs.

All the following explanations of the constraint graph are within the frame of the Forssea use case. This means that this report will not cover the generic constraints graph case, thus it will not describe some other specific constraints graph use case (in particular for set-membership networks which is a very different approch but still categorized under constraints graph method).

I'm now going to explain in more detail how constraint graph optimisation works and then we'll look in more detail at the Fuse framework that was used to implement it.

2.1 Constraints, Variables and Graph

2.1.1 Variables

A variable correspond to a wanted quantity at a given time, as the component of the state vector of robot, included its position. Variables are the vertices of the graph. The optimization algorithm works on these variables to adjust their values. Variables are the output of the fusion algorithm. The the variable is composed of a header (with IDs and time stamp) and the values (or the value in the case of 1D variables).

```
fuse_variables::Fixed1DVariable:
uuid: 0ab0345e-c696-5b0e-920f-f23f06c6182b
stamp: 159900000000
device_id: 209fef7c-6c31-53a7-bc16-c5e131b0c160
size: 1
data:
- x: -2.86535
```

Figure 2.1: Example of a Water Surface Variable Representation in Fuse.

In Figure 2.1, `Fixed1DVariable` is the type of the variable. The `device_id` is the ID of the concerned physical system, in this case the water surface. `stamp` is the time stamp of this variable, express in number of seconds since the Epoch. The `uuid` is the unique ID of the specific variable, is calculated pseudo-randomly with the seeds `device_id` and `stamp`. `size` corresponds to the size of the vector of the values. And `data/x` is the value of the fisrt dimension (in this case the only dimension) of the vairable, in this case height of the water surface above the WGS84 ellipsoid over the robot.

Notice the absence of data quality information over the quality of the data in variables. At the opposit of set-membership method, data accuracy is not contained in the variables but in the constraints. It is also important to note the creation of a new variable (with a new time stamp) when the algorithm gets or computes a new value. The values of a variable only change during the optimization, whereas all the meta data (`device_id`, `stamp`, etc.) can not be modify.

2.1.2 Constraints

A constraint corresponds to a mathematical relationship between variables at a given time, for example the physical law between the height of the water surface and the height of the robot given by the pressure sensor: $P_{rov}(t) = \rho\ g\ \ (z_{ws}(t) - z_{rov}(t))$. The constraints are the edges of the graph. The optimizer uses the constraints to determine the values of the variables. In Fuse, a constraint consists of a header (with IDs and timestamp), a list of variable Universally Unique Identifier (UUID)s and a cost function to calculate the values during optimization.

```
fuse_models::ConstantFixed1DStateKinematicConstraint
source: water_surface
uuid: 27477e05-b246-4402-ba71-6be8129476d2
variable 1: 970ad388-15de-533e-aa5a-3c29abd07850
variable 2: d24befa6-1583-5c0a-bdc4-e4d188449b5f
dt: 0.1
sqrt_info: 316.228
```

Figure 2.2: Example of a Motion Model Water Surface Constraint Representation in Fuse.

In FIGURE 2.2, `ConstantFixed1DStateKinematicConstraint` is the type of the constraint. `source` is the name of the physical system generating the constraint, in this case the water surface. `uuid` is the unique identifier of the specific constraint, randomly generated. `variable 1` corresponds to the UUID of the first involved variable. `dt` is a specific field `ConstantFixed1DStateKinematicConstraint` which is not in generic constraint, it is the time difference between the two variables timestamps. `dt` is only used in the cost function. And `sqrt_info` is the information matrix, in this case a square matrix of size 1 because there are two 1D variables involved.

An important point to note is the absence of time stamps in the constraints. Time information is contained in the variables. Moreover, the time stamps of the variables involved are not necessarily close; for example, in FIGURE 2.2 the time stamps are different. It's also important to note that the algorithm creates new constraints to handle each new variable. A constraint cannot change even during optimization, and no field can be modified.

There is no limit to the number of variables involved in a constraint, meaning that in the case of a constraint with only 1 or 2 variables, we can refer to a graph, but with a constraint with 3 or more variables, the use of the term graph is inappropriate. The exact mathematical structure is the hypergraph, where the variables are the vertices and the constraints the hyperedges. As this difference is not important for understanding, we take the liberty to use the term graph for hypergraph and the term edge for hyperedge.

In Fuse, there are two types of constraints: sensor model constraints and motion model constraints. Sensor model constraints are induced by new sensor data, while motion model constraints are generated to take the evolution of a physical system into account without any new information. Motion model constraints are generated when necessary, i.e. when requested by a sensor model. While defining sensor models, we must define which physical system is to be updated (i.e. which motion model constraints are to be created during callback) when new sensor data are received.

2.1.3 Graph

A hypergraph is an abstract structure in which objects, named vertices, may be related to each other. Other objects, called hyperedges, establish connections between several vertices (or a single vertex), without any number limit. A graph is a specific hypergraph in which hyperedges, called edges, connect vertices in pairs (or singly). In our constraint graph application, we use a hypergraph structure, although, by habit, we make the confusion of graph (resp. edge) with hypergraph (resp. hyperedges). Formally, a hypergraph is a pair (V, E) where V is a set of elements named vertices (or nodes, points, elements) and E is a subset of $\mathcal{P}(V)$ such that $\varnothing \notin E$. In this application of graph constraints, we consider variables as vertices and we attach information (such as IDs or cost functions) to the edges of the constraint frame, as previously discussed.

As we regularly (at the frequencies of the sensors) add variables and constraints to the graph, we need to remove old variables and constraints to avoid memory usage and computation time exploding. This removal is called marginalisation. Fuse keeps all constraints and variables between the current start time and the current time. The current time is the time of last time stamp awareness and the current start time is `current_start_time` = max(`first_time`, `current_time` − `lag_duration`) ; where `lag_duration` is a Fuse parameterisable value. To do this, all the variables to be marginalised are grouped together into an artificial variable, time stamped at the current start time, which can still be optimised, but does not represent a concrete quantity. The aim of this is to maintain the influence of history by re-processing all variables during optimization.

2.2 Non-Linear Optimization

2.2.1 Constraint Graph Optimization

The optimization phase is the heart of the constraint graph based fusion. The graph is the useful structure enabling a minimizer to optimize the values of variables in order to reduce a loss generated by the computation of cost functions of all constraints. As minimization is a recurrent issue in various fields of engineering, there is a wide spectrum of methods and implementations, and no optimizer specific to robotics. As the graph structure is widely independent of the optimization method used, this part is often perceived as a black box by roboticists. The main points to bear in mind are that the computation can be relatively heavy and is not always compatible with real-time processing, and that the result can be downright inaccurate under certain conditions if the algorithm used converges to a local minimum. In this case, choosing another optimizer (or changing the settings of the optimizer in use) may be a solution for finding a global minimum despite an unfavorable cost function.

2.2.2 Non-Linear Least Squares Optimizer

Fuse uses a non-linear least squares [7] optimizer called Ceres [5]. Each constraint has an associated cost function. There are two types of inputs to the cost function: some constants[3] and variables to be optimised. The output of the cost function is named residual. A residual is a vector that characterises the error weighted by the information matrix associated with the constraint. If the error vector of a constraint is $err(x)$, and A is the associated information matrix (meaning the noise is assumed to be Gaussian), the residual is:

$$res(x) := A\ err(x). \tag{1}$$

During optimisation, Ceres performs optimisation steps until the loss is below a threshold. At each step, all the residuals are calculated and transformed into a residual block:

$$RB_i := \rho_i \left(\|res_i\|^2 \right) \tag{2}$$

Next, the optimizer can compute the loss :

$$loss := \sum_i RB_i = \sum_i \rho_i \left(\|res_i\|^2 \right) \tag{3}$$

If the loss is below the threshold, the optimisation stops, otherwise Ceres starts another optimisation step of the Levenberg-Marquardt algorithm [7] to find new values for the variables and minimise the loss. For the sake of clarification and simplification, we will consider from now on that the output of a cost function is the error err instead of the residual res.

[3] The term *constant* is relative to Fuse. This type of inputs are constant for a given constraint and do not vary during optimisation. But they will change over time and take on different values in different constraints. Bear in mind that constraints are time stamped.

2.3 Fuse

2.3.1 Constraint Graph Implementation

Fuse [2] is an opensource (BSD License) ROS framework created by Locus Robotics to allow a customisable implementation of constraint graph optimization base on Ceres solver. Once in hand, Fuse allows to use constraint graph only by developing constraints and variables (or using existing variables) and calling them up in a configuration file.

An important asset of Fuse is to provide a completly asychronous framework. This allows to add multiple constraints without concern for the frequency of the input sensors.

2.3.2 ID Based Graph Architecture

A particular feature of the Fuse architecture is that each device (a device corresponds to a physical system), variable and constraint is identified by a unique UUID. A UUID is a standard 128-bit format. The probability of two random UUID being identical is low enough ($\frac{1}{2^{128}} \approx 10^{-39}$) to be considered negligible, allowing a unique identifier to be created without the need for a centralised authority. On the one hand, Fuse uses a classical random generator to create the device and the UUID constraint. On the other hand, for variable devices UUID Fuse uses a specific random generator which uses two seeds to generate the variable UUID. The special feature of this two-seed random generator is that it is possible to invert the generator to recover the seeds from the UUID output. The first seed is a device UUID and the second is a time stamp. This specificity will be used to find a desired variable on the graph.

The links between variables, constraints and devices are managed by the UUID system. Each model (motion model or sensor model) is associated with a unique device by its UUID. A variable UUID is generated with the UUID of the model that created it and the time stamp of the sensor data or the current time stamp. To update the graph, Fuse uses a transaction, which is a temporary sub-graph containing the new variables and constraints that will be merged into the global graph. To generate a transaction, Fuse must link the variables or transactions concerned. To do this, Fuse finds the existing variables on the graph using the variable UUID and the variable type for the search, and creates the new variables required. Then Fuse creates a new constraint with a random UUID and links it to the variables by their .

2.3.3 Sequence Diagram

Figure 2.3 shows the sequential Fuse calculation caused by a sensor input. When a sensor driver sends a new ROS message, the sensor model creates the necessary new variables and constraints (for the record, all constraints implement a cost function). The optimizer then updates the appropriate motion models to generate the variables and constraints corresponding to the evolution of these models. The motion models to be updated are defined in each sensor model configuration. The graph then contains all

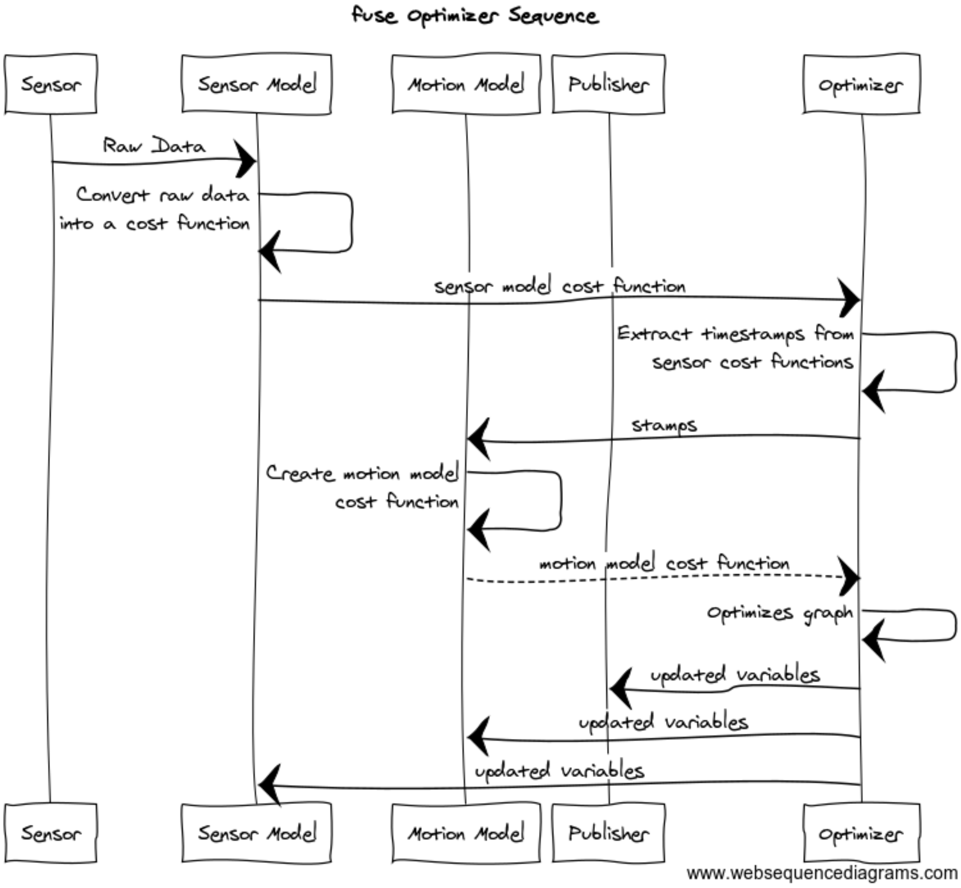


Figure 2.3: Fuse Sequence Diagram

the cost functions for calculating the optimisation. Once the optimisation is complete, the new variable values are sent to the various instances that need them.

2.4 Implementation

2.4.1 Notations

I am going to use the Forssea [11] notations, which are themselves inspired by marine and robotic notations, including the ROS [13, 16, 19, 22] standard. I'm allowing myself a few variations on this notation, mainly to clarify certain sub-entendres in order to bring greater clarity to this report. Let $\{a\}$ and $\{b\}$ be two frames, we will named the associate basis by the same name ($\{a\}_{frame} = (p_a, \{a\}_{basis})$). We note the vectors that compose the basis $(u_{ax}, u_{ay}, u_{az}) := \{a\}_{basis} = \{a\}$. And we only use orthonomal direct bases.

x^a: Coordinates of the vector x express in the basis $\{a\}$. Or coordinates of point x express in the frame $\{a\}$.

$x^a{}_{|i}$ i-th coordinate of the vector x express in the basis $\{a\}$. Or i-th coordinate of the point x express in the frame $\{a\}$.

p_a: Point of origine of the frame $\{a\}$. As the is no reference frame, p_a refers to the point rather than its coordinates.

$p_{b/a}$: Vector of origine of the frame $\{b\}$ with respect to the frame $\{a\}$, i.e. $p_{b/a} = p_b - p_a = \overrightarrow{p_a p_b}$. As there is no reference basis, $p_{b/a}$ refers to the vector rather than its coordinates.

(λ, μ, h): Geographical coordinates latitude, logitude, height above the ellipsoid. Unless otherwise specified, the WGS84 ellipsoid is used.

d_p: Represents the depth of the point p with respect to the water surface above p. It is measured positive downwards.

h_p: Represents the height of the point p with respect to the surface of the ellipsoid. It is measured positive upwards.

a_p: Represents the altitude of the point p with respect to the surface of the ground below p. It is measured positive upwards. A distinction must be made between the notations a_p for the altitude and $a_{p,a/b}$ or $a_{a/b}$ for acceleration.

Θ_{ab}: Tait-Bryan angles, or yaw, pitch, roll of the rotation of the frame $\{b\}$ with respect to the frame $\{a\}$. $\Theta = (\phi, \theta, \psi)$.

q_{ab}: Quaternion of the rotation of frame $\{b\}$ with respect to the frame $\{a\}$. $q_{ab} = \left(q_{ab\ |w}, q_{ab\ |x}, q_{ab\ |y}, q_{ab\ |z}\right)$.

R^a_b: Rotation matrix of the rotation frame $\{b\}$ with respect to the frame $\{a\}$. $R^a_b = R(\Theta_{ab}) = R(q_{ab})$.

$v_{p,b/a}$: Linear velocity of the point p fixed in the frame $\{b\}$ with respect to the frame $\{a\}$. Let us note $v_{b/a} = v_{p_b,b/a}$.

$\omega_{b/a}$: Angular velocity of the frame $\{b\}$ with respect to the frame $\{a\}$.

$a_{p,b/a}$: Linear acceleration of the point p fixed in the frame $\{b\}$ with respect to the frame $\{a\}$. Let us note $a_{p_b,b/a} = a_{b/a}$. A distinction must be made between the notations $a_{p,a/b}$ or $a_{a/b}$ for acceleration and a_p for the altitude.

$\wedge$: Cross product 3D-vector operator.

If we note x a physical quantity, we will note $\hat{x}$ the Fuse estimate and $\tilde{x}$ a sensor input, subject to existence. We also note the error $err(x) = \tilde{x} - \hat{x}$. For example $v_{b/e}$ is a physical quatity of the linear velocity of the body with respect to ECEF frame $\{e\}$, $\hat{v}^{b}_{b/e}$ is a Fuse estimate of the physical quatity $v^{b}_{b/e}$ and $\tilde{v}^{dvl}_{dvl/e}$ is the physical measurement of the physical quatity $v^{dvl}_{dvl/e}$ made by the DVL sensor. We may use $\hat{x}$ to refer to a calculated value compute from fuse variable. For example $\hat{v}^{dvl}_{dvl/e}$ is the expected DVL measurement calculated using the Fuse variables. We can use other notations such as $\mathring{x}$ to differentiate a calculated variable (e.g. a quantity calculated with the combination of sensor and fuse data) from the associated physical quantity, the use of this notation requires case-by-case definition.

$\{e\}$ The ECEF frame is a fixed reference frame relative to the Earth, with the Earth's centre of mass as the origin, u_{ez} points North, and u_{ex} points towards the intersection of IRM[4] with the equator [1, 8].

$\{m\}$ The map frame, fixed to the Earth, is defined by the user as the local frame of the working space. The origin is placed on the surface of the reference ellipsoid, u_{my} points North and u_{mz} is the normal to the ellipsoid.

$\{b\}$ The body frame is the ROV's fixed frame. The origin is named center of origin, u_{bx} goes forward and u_{by} goes left.

$\{n\}$ The navigation frame has the centre of origin of the ROV for origin, u_{bx} points East and u_{by} points North. This frame is useful for considering the robot's position without concern for its orientation.

$\{o\}$ The odometry frame is a frame in which data from odometry sensor will remain continous. In the event of a jump in the known $\{n\}$ due to a new absolute data from a reliable sensor, the odometry frame will jump to allow transparent use of odometry sensors [19].

$\{ws\}$ The water surface frame is the projection of $\{n\}$ on the water surface. This corresponds to a translation $d_{p_b} u_{nz}$ of the origin of $\{n\}$. This frame is used only for the altitude of the water surface.

$\{gd\}$ The ground frame is the projection of $\{n\}$ on the ground. This corresponds to a translation $-a_{p_b} u_{nz}$ of the origin of $\{n\}$. This frame is used only for the altitude of the ground.

$\{s\}$ Each sensor has a frame which the measurements are taken. This frame is fixed to $\{b\}$ and known by mesurement before operations on the SolidWorkws CAD model, see FIGURE A.7.

$\{t_i\}$ The i-th thruster frame is fixed to $\{b\}$. It is used for thrust allocation calculations.

Most of the time, conversions are performed using the ROS TF tree [12], which is the standard way to use frames with ROS. The TF tree is constructed from ROS messages containing the transformation from one frame to another. A Listener echoes the `tf_static` and `tf` topics for transform messages and, when a transform between two frames is queried, the Listener module walks up through the tree until a common parent is found. Then the found path is used to calculate the desired transform. All transformation messages are time stamped. Messages on `tf_static` are long-lived, corresponding to fixed transformations that are not expected to change and are

[4] The IRM is close to Greenwich meridian.

occasionally republished. Messages on `tf` are regularly updated and can support dynamic movements. At Forssea, TFs are published by the localisation module.

The fusion system used is the composition of three physical systems: Argos, the water surface above Argos, the ground below Argos. This gives us the state vector: $\left(p^e_{b/e}, q_{eb}, v^b_{b/e}, \omega^b_{b/e}, a^b_{b/e}, d_{p_b}, a_{p_b}\right)$. This makes us a state vector with 18 scalar variables with 17 degrees of freedom (due to the 3 degrees of freedom of the quaternion q_{eb}).

2.4.2 Depth Constraint

In underwater robotics, the pressure sensor is a highly interesting sensor. It provides direct and reliable information on the robot's depth, without any integration and at little cost.

We will now develop the equations induced by the pressure sensor. Please refer to 2.4.1 for more information on the notations used. The hydrostatic equation gives $P(p_b) = \rho_w \; g u_{nz} \cdot (p_{ws} - p_b)$ where $P(p_b)$ is the pressure at the position p_b; ρ_w is the density of the water, assumed to be constant because variations due to salinity, temperature or pressure are sufficiently small (on the order of 0.1% at most) ; $-g u_{nz}$ is the gravitational acceleration, also assumed to be constant (variations due to the earth's rotation, its non-sphericity and altitude are of the order of 10^{-4}% at most) and d_{p_b} is the depth at p_b. This gives us

$$d_{p_b} := (p_{ws/e} - p_{b/e}) \cdot u_{nz} = \frac{P(p_b)}{\rho_w \; g} \tag{4}$$

We now need to calculate the errors of the constraint. The cost function inputs are composed of pressure sensor data: $\tilde{d}_{p_b} := \dfrac{\tilde{P}(p_b)}{\tilde{\rho}_{sea} \; \tilde{g}}$ and Fuse variables : $\hat{p}^b_{b/e}$ and $\hat{p}^b_{ws/e}$. The cost function output is the error : $err(d_{p_b}) := \tilde{d}_{p_b} - \hat{d}_{p_b}$ where $\hat{d}_{p_b} = \hat{p}^b_{ws/e \; |3} - \hat{p}^b_{b/e \; |3}$. Finaly

$$err(d_{p_b}) = \tilde{d}_{p_b} - \hat{p}^b_{ws/e \; |3} + \hat{p}^b_{b/e \; |3} \tag{5}$$

2.4.3 DVL Constraint

A DVL measures the linear velocity of the ground relative to the sensor (i.e. the robot relative to the ground with one inversion). The measurement is made by sending acoustic pulses, the variation in frequency of the reflected wave is used to calculate the velocity using the Doppler effect. The advantage of this sensor is to provide linear velocity values with no integration error, giving a drift-free linear velocity data.

We are now going to develop the equations induced by the DVL sensor. Please refer to 2.4.1 for more information on the notations used. The velocity composition gives us: $v_{b/e} = v_{b/s} + v_{p_b,s/e}$. And the Varignon's relation: $v_{p_b,s/e} = v_{s/e} + p_{b/s} \wedge \omega_{s/e}$. As the frames $\{b\}$ and $\{s\}$ are rigidly attached, we have $v_{b/s} = 0$ and $\omega_{s/e} = \omega_{b/e}$. This is given us

$$v_{b/e} = v_{s/e} + p_{b/s} \wedge \omega_{b/e} \tag{6}$$

We now need to compute the error of the constraint, where each coordinate of the error must be expressed as a function of the inputs. The inputs to the cost function consist of DVL data: $\tilde{v}^s_{s/e}$ and Fuse variables: $\hat{v}^b_{b/e}$, $\hat{\omega}^b_{b/e}$ and $\hat{q}_{eb}$. We also need fixed constants[5]. These are $\tilde{p}_{s/b}$ and $\tilde{q}_{bs}$. The output of the cost function is the error vector: $err(v^b_{b/e}) := err(v^b_{b/e}) = \mathring{v}^b_{b/e} - \hat{v}^b_{b/e}$ where $\mathring{v}^b_{b/e}$ is a transformation of the DVL data $\tilde{v}^s_{s/e}$ to represent the velocity of the body express in the body basis. It is important to note that $\mathring{v}^b_{b/e}$ is not direct sensor data and that Fuse variables are used for its calculation.

Let us now express the relation (6) in the body frame $\{b\}$ with the intputs and outputs of the problem in order to write the cost function. We have $v^b_{s/e} = R^b_s \cdot v^s_{s/e}$ and $R^b_s = R(q_{bs})$

$$\begin{aligned} v^b_{b/e} &= v^b_{s/e} + p^b_{b/s} \wedge \omega^b_{b/e} \\ &= R(q_{bs}) \cdot v^s_{s/e} + p^b_{b/s} \wedge \omega^b_{b/e} \\ \mathring{v}^b_{b/e} &= R(\tilde{q}_{bs}) \cdot \tilde{v}^s_{s/e} + \tilde{p}^b_{s/b} \wedge \hat{\omega}^b_{b/e} \end{aligned}$$

$$err(v^b_{b/e}) := \mathring{v}^b_{b/e} - \hat{v}^b_{b/e} = R(\tilde{q}_{bs}) \cdot \tilde{v}^s_{s/e} + \tilde{p}^b_{s/b} \wedge \hat{\omega}^b_{b/e} - \hat{v}^b_{b/e} \tag{7}$$

2.5 Assets and Drawbacks

It is too early in the Fuse implementation at Forssea to provide relevant results to assess the effectiveness of Fuse. For the moment, the Fuse model with the Argos gazebo simulation is giving very good results compared with the simulation truth, but this is mainly due to the simplicity of the Fuse system used and the noise configuration of the gazebo simulation. There is no point in pushing the evaluation any further until the Fuse implementation is more complete. I will therefore draw some advantages from the disadvantages of the constraint graph and the use of Fuse in a qualitative way, without carrying out a quantitative study in relation to the algorithm currently used and in relation to the simulation and the ground truth.

2.5.1 Assets

The first advantage of Fuse is its modularity: it can integrate as many sensors or evolution constraints as required, and it is easy to switch from one sensors configuration to another. This means that Fuse can be used in various contexts, such as for SLAM. This enables Fuse to be used in different products such as INS ROVs and INS agnostic ROVs, but also in Forssea's smart cameras. Modularity is relevant from the point of view of the user and the developer, but it is also relevant for the implementation of Fuse.

[5] $\tilde{p}_{s/b}$ and $\tilde{q}_{bs}$ are measured before the operation on the SolidWorkws CAD model.

Indeed, Fuse provides a completely asynchronous framework, which allows parallelism and independence of certain processes during code execution.

Furthermore, Fuse provides a higher level framework. This means that it is not necessary to master the nonlinear minimisation algebra nor Ceres to develop a new constraint, neither is it necessary to master Fuse to use Fuse and to fine-tune the Fuse configuration. Its handling of non-linear constraints and its way of preserving history with marginalisation are also interesting features. Fuse uses a Google open source library for optimisation, which gives it high reliability, an extensive Application Programming Interface (API) and a good community. In addition, Fuse is based on ROS, which allows natural integration with the ROS style in the Forssea environment.

2.5.2 Drawbacks

Like all localization methods, Fuse is not perfect and has its disadvantages. The first drawback I'd like to point out is the one that has been a major difficulty in my work: the heaviness of the Fuse implementation. Indeed, the code is complex and the fact that execution is highly asynchronous makes understanding the Fuse code non-trivial. As I started working with a version of Fuse modified by Forssea, which didn't run, this difficulty in understanding the Fuse code shouldn't be as important in the future Forssea development as it was for me.

Besides the accuracy of results, all real-time localisation methods need to be executed at a sufficiently high frequency to be reliable. For the moment, Fuse's consumption on the CPU is low, but the computation of the Ceres optimisation will increase as the number of constraints increases. This increase of CPU consumption has no reason to grow non-linearly, on the contrary it can grow very quickly. In terms of CPU efficiency and reliability, the use of ROS is no guarantee of quality. Even if the use of ROS 2 improves efficiency and reliability, ROS remains a heavy framework and bugs still occur. We can add to the above disadvantages the small community of Fuse users compared with more popular methods such as Kalman libraries.

2.5.3 Justification of the Method

As we have seen above, good use of Fuse, properly tuned for Argos localization, will be a major improvement for the Forssea ROV and for future development. As the choice to use Fuse is not a trivial one, a full quantitative study will be required in terms of accuracy, robustness and CPU consumption.

3 Frontal Sonar

To use a sonar in an acoustic SLAM for a ROV, we need to interface the sonar's embedded software with the robot's software ecosystem. To do so, I developed a ROS driver for an oculus sonar from BluePrint Subsea.

3.1 Forward-Looking Sonar

3.1.1 Sonar Presentation

A multi-beam forward-looking sonar is an underwater composed of tree parts. Firstly, an acoustic transmitter that sends sound pulses into the water, which means that sonar is an active sensor (i.e. a sensor that alters its environment). Secondly, an acoustic receiver, which can sometimes be the same component as the transmitter. Thirdly, a software layer that takes into account the date the pulse was sent, and the date, orientation and intensity of the corresponding received echo, for data calculation and returns to the user. The term *multi-beam* refers to the directionality of the receiver: the receiver is able to discriminate the angle of the echo beams, as explained in the FIGURE 3.2. The forward-looking term refers more to the use of sonar than to its intrinsic characteristics. An Forward Looking Sonar (FLS) is intended to provide information ahead of the boat or robot, often for navigation, obstacle avoidance and/or environmental observation.

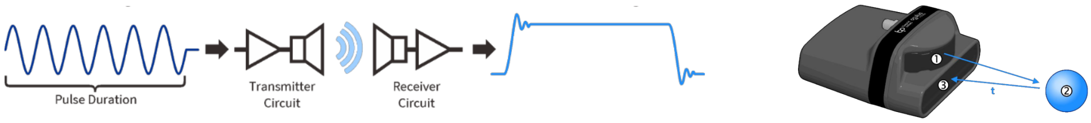


Figure 3.1: Operating Principle of the Oculus Sonar from BluePrint Subsea Documentation.

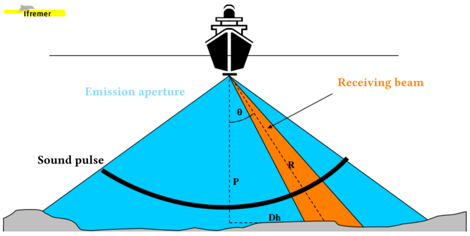


Figure 3.2: Crossed Beams Principle for Multibeams Sonar from Ifremer.

Compared with electromagnetic waves, acoustic waves provide less precise and noisier data. However, the density of water allows the sound to propagate well with fairly little absorption, whereas electromagnetic waves disperse very quickly. This makes sonar an interesting tool for underwater observation of the environment.

3.1.2 Oculus

As ENSTA Bretagne's robotics laboratory, Forssea Robotics chose to use an FLS Oculus M-Series[6] from Blueprint Subsea, cf FIGURE 3.3.

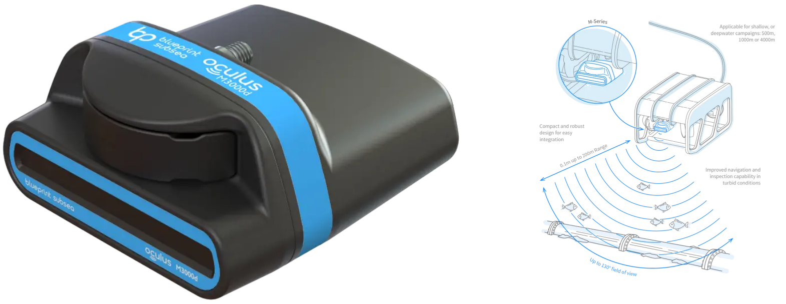

blueprint subsea
oculus
M1200d
M-Series
Applicable for shallow, or deepwater campaigns: 500m, 1000m or 4000m
Compact and robust design for easy integration
0.1m up to 200m Range
Improved navigation and inspection capability in turbid conditions
Up to 130° field of view

Figure 3.3: Blueprint Subsea Forward-Looking Sonar Oculus M-Series 1200d

3.2 ROS Driver

Bueprint Subsea only provides an Human Machine Interface (HMI) for using the sonar. As there is no API, the sonar has the major disadvantage of not being directly interfaceable with the Forssea Robotics software environment. Thus, the creation of a driver, allowing integration on the web HMI and intelligence codes, was necessary. To do this, we needed a C++ API to communicate with the sonar software, then a ROS package to interface the C++ API with the Forssea software environment.

3.2.1 Low Level Communication

The API is provided by the `oculus_driver` library developed at ENSTA Bretagne by Pierre NARVOR. This library, written in C++, offers a C++ and Python API for interacting with the sonar software via sockets. As is customary at Forssea, the ROS package is developed in C++, so the Python API is seldom used.

The `oculus_driver` library allows to enable and disable the connection with the sonar, to obtain and modify its configuration and to define callbacks. The first settable callback is called at every status message send by the sonar (in practice, every second), and the second settable callback is called at every ping message send by the sonar (in practice, as soon as the sonar software processes a ping echo).

The ping callback provides the data returned by the sonar software, as explained in FIGURE 3.4. The ping data consists of a header (`OculusSimplePingResult`) including metadata, echo beams bearings (`Bearings`) and echo beams sound intensity values (`PingData`). Metadata includes information such as timestamps and sonar configuration information during the ping. Bearings are sent with each ping, as they are not constant

[6] M750d for ENSTA Bretagne's robotics laboratory and M1200d for Forssea Robotics. Further information on the manufacturer's website: https://www.blueprintsubsea.com/oculus/oculus-m-series

over time. Indeed, due to the calculation method used by the sonar to process the echo data, the bearings of each direction heard by the sonar can change from one ping to the next and have no reason to be a linear scale, as explained by FIGURE 3.5. Consequently, angular resolution is neither linear nor fixed. More precisely, angular resolution is finer around the sonar center and coarser at the extremities. This resolution varies with distribution from ping to ping, while the number of bearings remains constant.

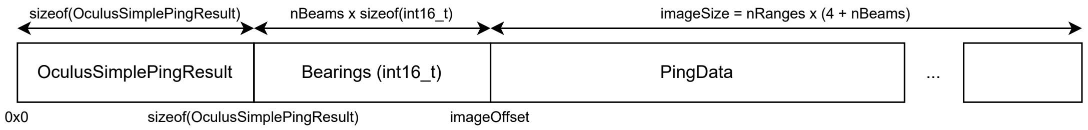

sizeof(OculusSimplePingResult)
nBeams x sizeof(int16_t)
imageSize = nRanges x (4 + nBeams)
OculusSimplePingResult
Bearings (int16_t)
PingData
...
0x0
sizeof(OculusSimplePingResult)
imageOffset

Figure 3.4: Oculus Sonar Ping Contents

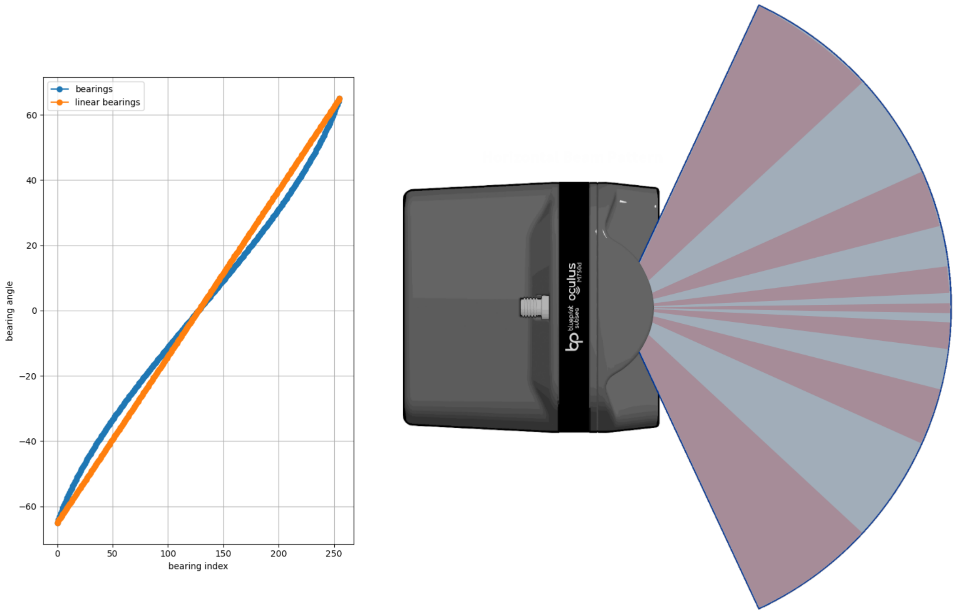
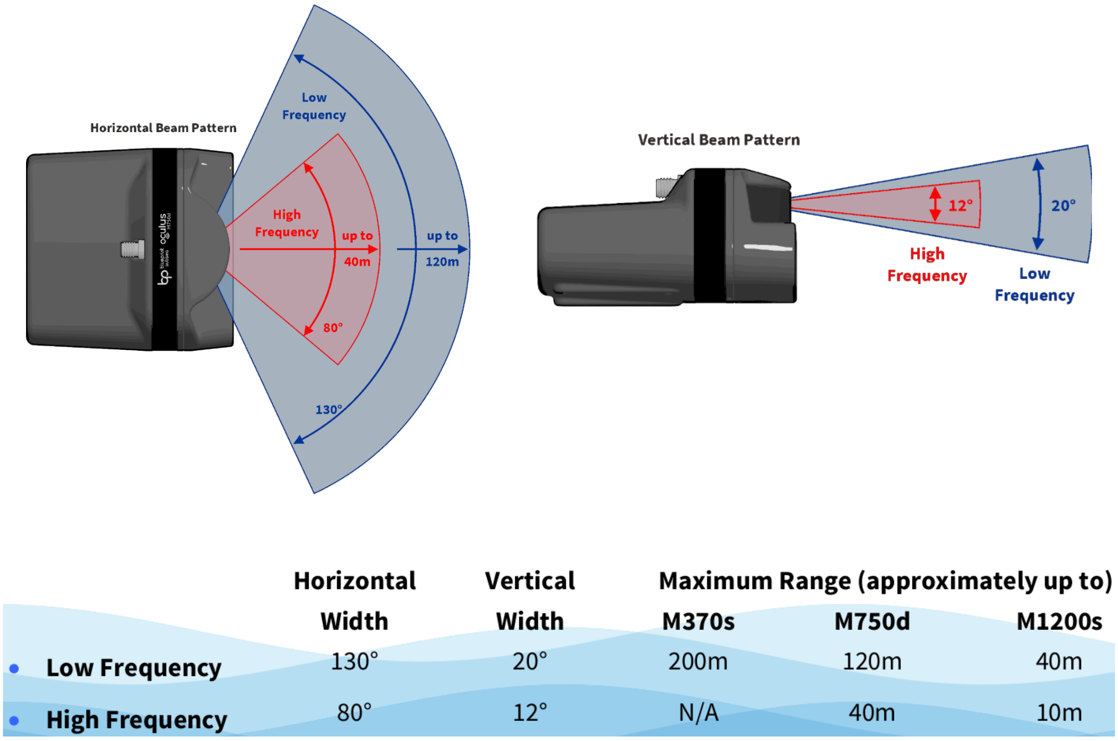


Figure 3.5: The Bearings of the Oculus Sonar: non-linear nor constant throught time.

However, frontal sonar has a vertical angular aperture of 12° or 20°, depending on the frequency used, see FIGURE 3.6. This vertical angular aperture is a source of error, as we cannot define exactly where the object is located vertically within this angle, see FIGURE 3.7. This is an important point to bear in mind before proceeding with any sonar data processing. Moreover, this vertical indeterminacy makes it difficult to use this type of sonar for 2D localization or mapping: depending on the distance from the object concerned, the sonar rendering can be very different, as you can see in FIGURE 3.8.

To reduce the weight of sonar data, the intensity of pings received by the sonar is weighted by gains. More precisely, each time a ping is sent by the sonar, it listens to the



	Horizontal Width	Vertical Width	Maximum Range (approximately up to)		
			M370s	M750d	M1200s
• Low Frequency	130°	20°	200m	120m	40m
• High Frequency	80°	12°	N/A	40m	10m

Figure 3.6: Aperture Angle Oculus M1200d Sonar from BluePrint Documentation.

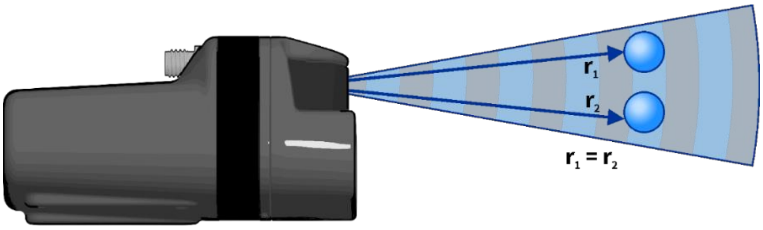


Figure 3.7: Aperture Angle Indetermination from BluePrint Documentation.

sound intensity on each of its bearings. At each instant, an intensity is obtained for each bearing. These intensities are normalized by a gain to obtain the best 1-byte resolution for all intensities at the same instant. The gain and data are then stored in memory in the form explained in Figure 3.9.

3.2.2 ROS Package

The package, named `oculus_sonar`[7], is a ROS 2 metapackage that contains the packages `oculus_interfaces` and `oculus_ros2`. The purpose of `oculus_interfaces` is only to provide the necessary ROS interfaces, i.e. to create customized ROS messages. The aim of `oculus_ros2` is to connect the ROS interfaces with the `oculus_driver`

[7] These packages are also available on GitHub from the robotics department at ENSTA Bretagne: `oculus_ros` and `oculus_ros2`.

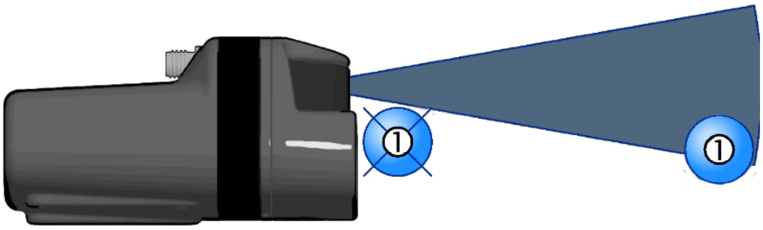


Figure 3.8: Importance of Vertical Aperture Angle.

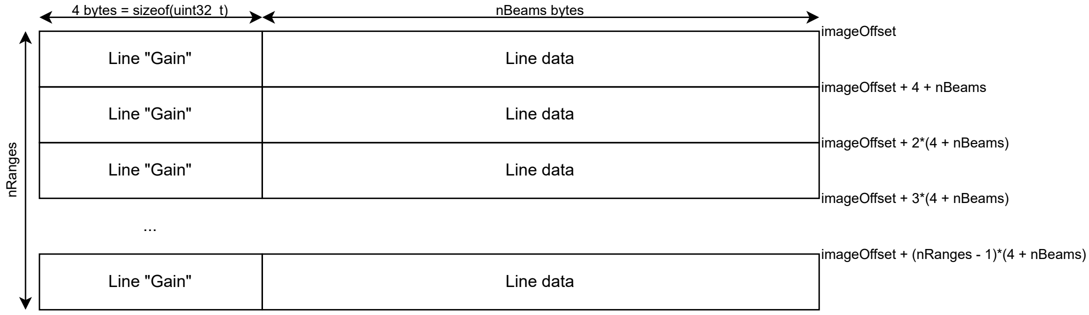


Figure 3.9: Ping Data Contents

API. This means requesting the sonar to ping, generating the raw image and/or the conical image, converting and publishing the status and ping data into ROS messages and creating ROS parameters and managing their modification from ROS but also from the sonar itself. To do this, the code is split into two C++ classes: `SonarViewer`, which handles all the transformation of a sonar ping into a raw image or a conical image; and `OculusSonarNode`, which uses the `oculus_driver` API, the `SonarViewer` class and the RCLCPP API to create the ROS node.

To produce an image, it is necessary to remove the gains (`Line "Gain"`) and correct the raw echo data (`Line data`), as shown in FIGURE 3.10. To do this, every pixel in the line must be corrected using the following formula:

$$\text{gain\_corrected\_pixel}_{i,j} = \frac{\text{raw\_pixel}_{i,j}}{\sqrt{\text{gain}_i}} \quad \forall i, j \in nRanges \times nBeams$$

The raw image is a cartesian image that makes the data visible as it is without constructing the conical image once the gains have been removed and the data corrected by the gains ; pixel (i, j) is the intensity of the echo received in the j-th angle at the i-th moment of beams reception. The conical image is constructed by taking into account the position (i.e. bearings and ranges) of each pixel to provide a scaled image that respects angles and distances. Eventually, filtering can be performed. The image is encoded in an 8-bit data channel. The intelligence code will use this image as is, so the HMI will need a YUV-I420 conversion to handle the image without any processing.

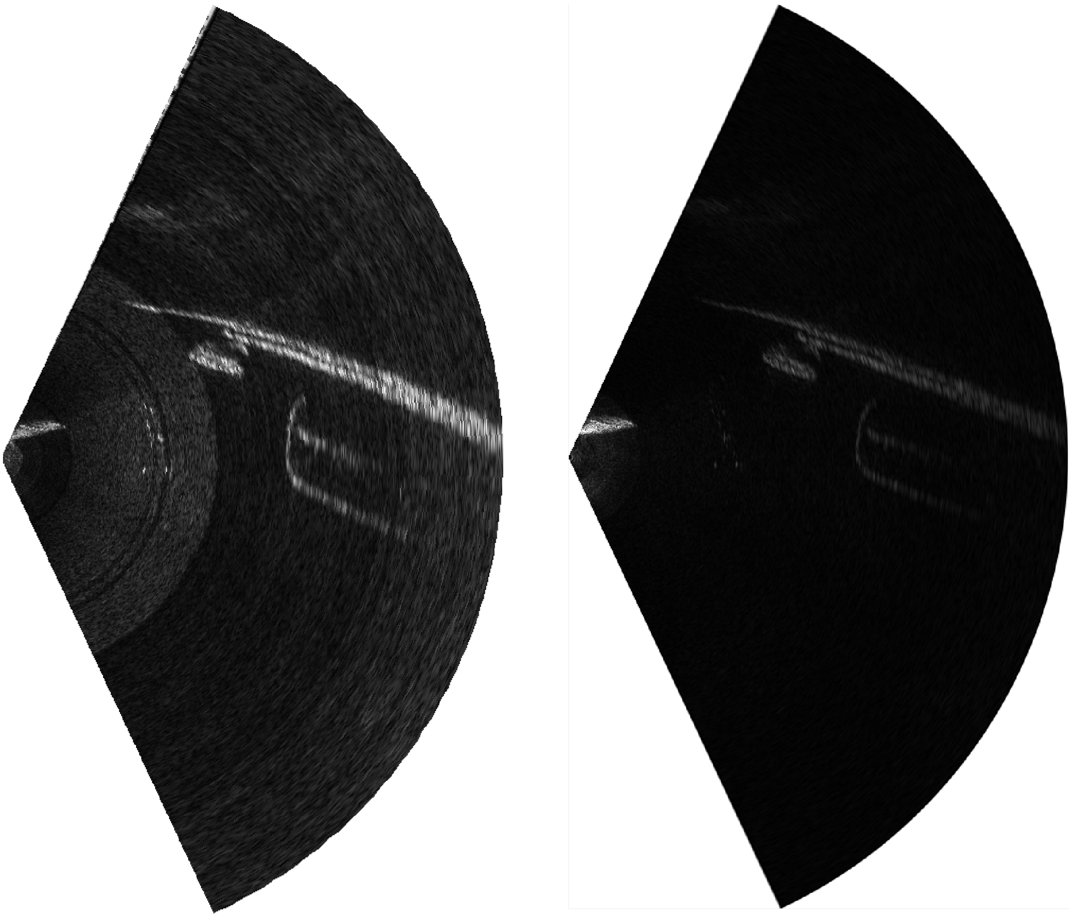


Figure 3.10: Range Gains Influence on Sonar Images.

In order to provide better protection to prevent the sonar from overheating and to enable ROS users to use the standby mode more intuitively, two FSM has been introduced.

3.2.3 Sequence Diagram

The sequence diagram in Figure 3.11 shows details of the driver's execution. We can see the sonar interacting with the low-level driver, which interacts with the ROS driver, which in turn interacts with ROS. We can count three types of interaction with the sonar: status data sent by the sonar, ping data sent by the sonar and configuration received and sent by the sonar. Status messages are parsed into ROS messages almost without change. In contrast, the Ping message needs the data to be transformed as described above. As the sonar may or may not accept a configuration change, each configuration request requires feedback.

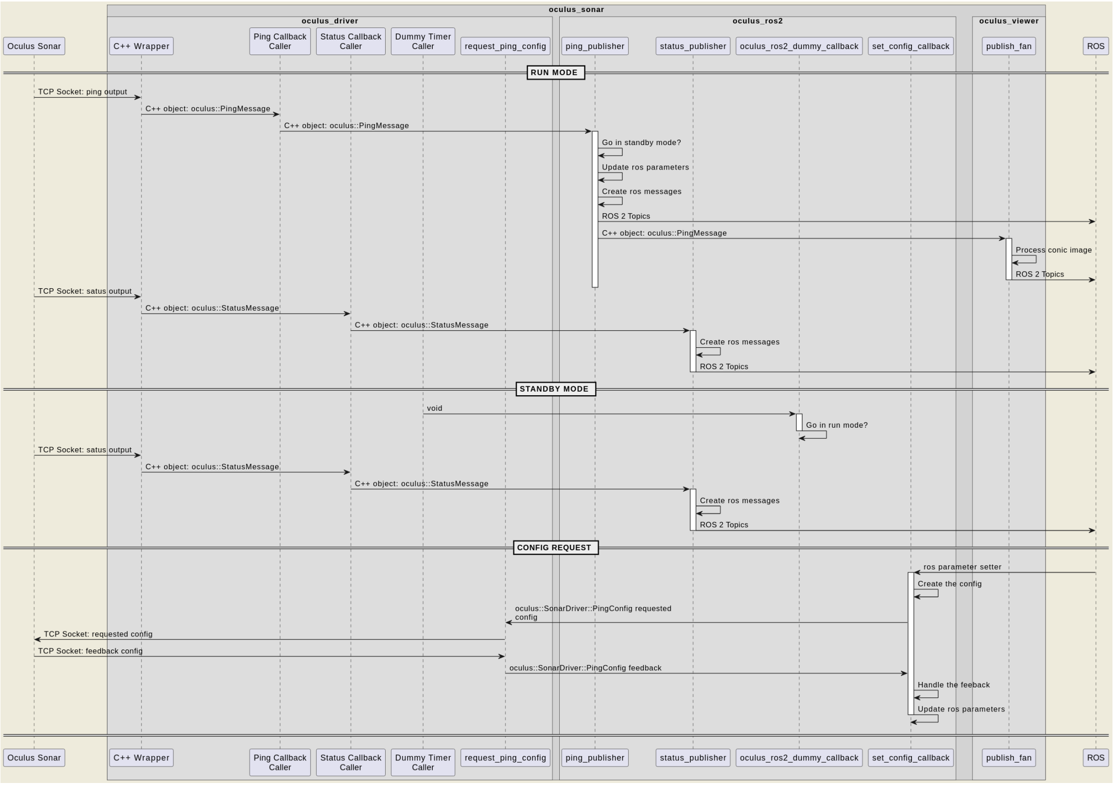


Figure 3.11: Sequence Diagram

3.3 Tests and Validations

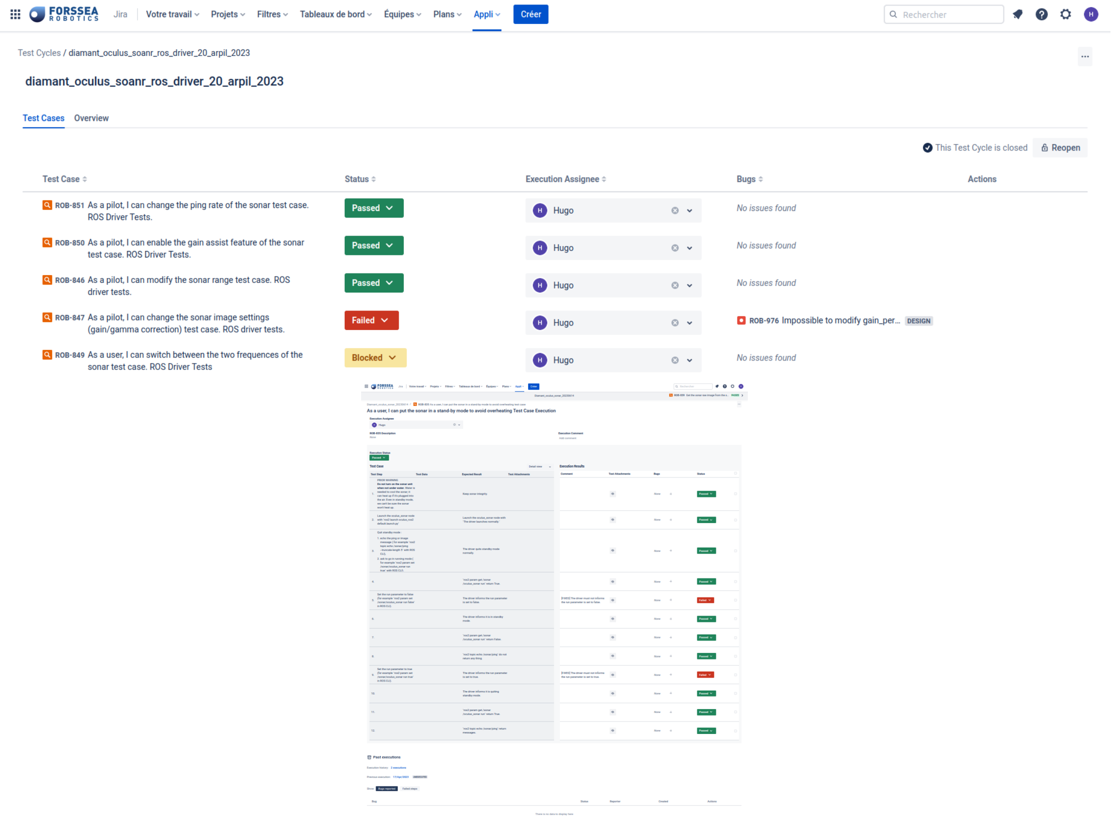
An important part of my work on the ROS package driver has been to write and execute test protocols to validate that the use of sonar conforms to the requested design. As Forssea aims to adopt more industrial processes, the idea is to generalize testing and validation of hardware and code independently and also in use cases. This enables bugs and limitations to be detected as soon as possible. Since writing and executing tests is time-consuming and tedious, this task naturally tends to be postponed. Performing a rigid test protocol also allows to change the development setup, in which I always use the same recorded data (in rosbag format) or with the same known environment to observe.

3.3.1 Low Level Integration Tests

Tests are usually divided into two categories: unit tests and integration tests. Unit tests are low level tests that validate the correct behavior of an elementary component. In software, unit tests are often a series of boolean checks to verify the correct implementation of a function or class. Integration tests can again be divided into two categories: low level integration tests and user integration tests. Whereas low level integration tests are there to validate the correct behavior of a part of an overall system, user integration tests aim to validate each user feature in the final use case. The tests I performed on sonar package are low level integration tests.

The aim of conducting these tests is to validate the quality of sonar outputs on ROS topics in relation to the chosen ROS configuration. In other words, these tests concern the package's interfaces with ROS, but also with the low level sonar driver.

To validate the ROS package I developed, I used the jira Quality Test tool, see FIGURE 3.12. This tool allows you to create a test for each specific feature and group them into a rigid test protocol. The test protocol enables an uninformed person to mechanically follow the protocol and have clear measurements to validate, or not, each test. Jira also allows the tester to create a bug ticket that will be assigned to a developer to fix the problem.



Test Cycles / diamant_oculus_soanr_ros_driver_20_arpil_2023

diamant_oculus_soanr_ros_driver_20_arpil_2023

Test Case	Status	Execution Assignee	Bugs	Actions
ROB-851 As a pilot, I can change the ping rate of the sonar test case. ROS Driver Tests.	Passed	Hugo	No issues found	
ROB-850 As a pilot, I can enable the gain assist feature of the sonar test case. ROS Driver Tests.	Passed	Hugo	No issues found	
ROB-846 As a pilot, I can modify the sonar range test case. ROS driver tests.	Passed	Hugo	No issues found	
ROB-847 As a pilot, I can change the sonar image settings (gain/gamma correction) test case. ROS driver tests.	Failed	Hugo	ROB-976 Impossible to modify gain_per... DESIGN	
ROB-849 As a user, I can switch between the two frequences of the sonar test case. ROS Driver Tests	Blocked	Hugo	No issues found	

As a user, I can put the sonar in a stand-by mode to avoid overheating Test Case Execution

Figure 3.12: Jira Quality Tests Tool

The test environments themselves can be classified into two types: Factory Acceptance Testing (FAT) and Sea Acceptance Testing (SAT). FAT is a common industrial protocol, while SAT is logically specific to certain subsea products. As the test involves software, a FAT does not take on the aspect of a product end-of-line test. By the acronym FAT, I mean all the tests that can be carried out in the workshop, i.e. water bucket tests and pool tests.

3.3.2 FAT

As the sonar needs to be in water to cool down prior to being safely powered on, the minimum setup for development or testing is to place the sonar in a bucket of water. In this case, the sonar images are totally unusable, but this setup allows to pass a number of configuration tests.

The workshop has a pool, which is the easiest way to obtain a proper setup allowing acceptable rendering of the sonar image. However, the pool still has its drawbacks, such as the lack of interesting features to observe (only the sides, corners and bottom of the pool) and the presence of echoes due to the confined environment.

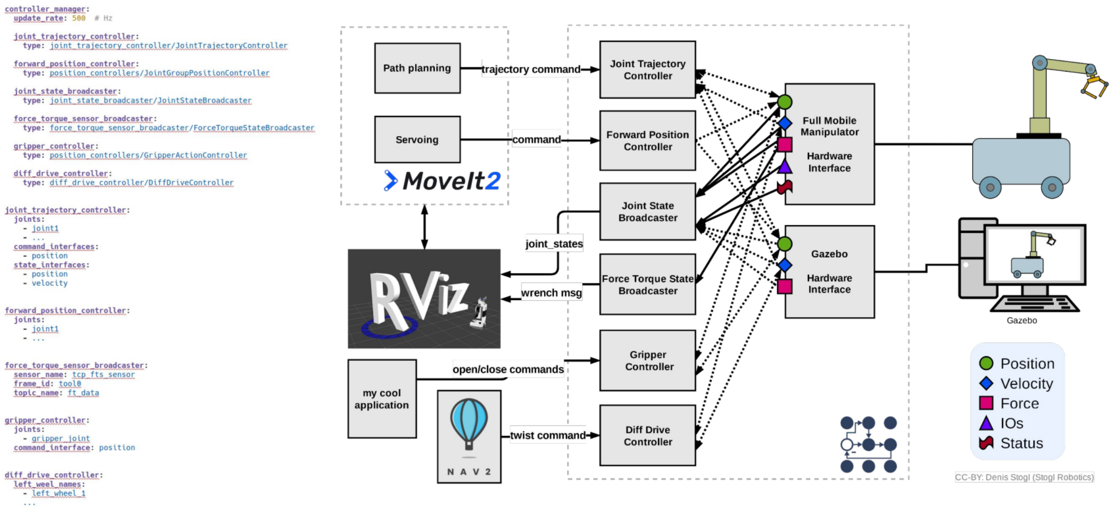
3.3.3 SAT

A SAT is the best way to pass all the tests. I took advantage of several Argos SATs on the Harmatan. The Harmatan is a boat moored on the Quai du Maros in Sète. This boat is an simple way for Forssea to carry out Argos development tests or SATs. As the boat rental and Argos deployment are costly and resource consuming, these tests take a full week. Most of the time, the Harmatan remains docked, which is sufficient for most Argos tests, allowing me to easily go on board to run the whole test protocol.

4 Controllers Chain

4.1 ROS 2 Control

ROS 2 Control is a ROS framework to work with controllers. The main idea of this framework is to provide a standard, efficient and reliable tool for implementing controllers and associated interfaces. Classic controllers are already implemented, and others are created by the ROS community. As the direct heir to ROS Control, which was the framework for ROS 1, ROS 2 Control, like ROS 2 itself, aims to be more efficient by rethinking the framework from top to bottom (clearer semantics and architecture, simpler and more customizable code, creation of chained controllers can be noted). A key feature of ROS 2 Control is to provide a level of abstraction by ensuring that all control blocks are called based on a configuration file. The aim is to be highly customizable and to cover a wider range of use cases with the same blocks.



```yaml
controller_manager:
  update_rate: 500  # Hz

  joint_trajectory_controller:
    type: joint_trajectory_controller/JointTrajectoryController

  forward_position_controller:
    type: position_controllers/JointGroupPositionController

  joint_state_broadcaster:
    type: joint_state_broadcaster/JointStateBroadcaster

  force_torque_sensor_broadcaster:
    type: force_torque_sensor_broadcaster/ForceTorqueStateBroadcaster

  gripper_controller:
    type: position_controllers/GripperActionController

  diff_drive_controller:
    type: diff_drive_controller/DiffDriveController


joint_trajectory_controller:
  joints:
    - joint1
    - ...
  command_interfaces:
    - position
  state_interfaces:
    - position
    - velocity


forward_position_controller:
  joints:
    - joint1
    - ...


force_torque_sensor_broadcaster:
  sensor_name: tcp_fts_sensor
  frame_id: tool0
  topic_name: ft_data


gripper_controller:
  joints:
    - gripper_joint
  command_interface: position


diff_drive_controller:
  left_weel_names:
    - left_wheel_1
    ...
```

Figure 4.1: An Example of ROS 2 Use in roscon22

To bypass ROS topics' communication style, which can be unnecessarily heavy depending on the DDS used, ROS 2 Control prefers to use a system based on shared memory named ROS interfaces. There are two different types of interfaces. Command interfaces, which are used to communicate commands from controllers (resp. previous controller) to resources (resp. next controller). And state interfaces, which are the interface to communicate feedback from resource (resp. next controller) to controller (resp. previous controller).

ROS 2 Control's architecture is based on a dual node management system. A Controller Manager, which manages the nodes related to the control proper ; and a Resource Manager, which manages the interface between the controllers and the resources, i.e. between the controllers and the actuators and sensor drivers. The Controller Manager is the central component of ROS 2 Contol. Its task is to load and update controllers in accordance with a configuration file as described in Figure 4.2.

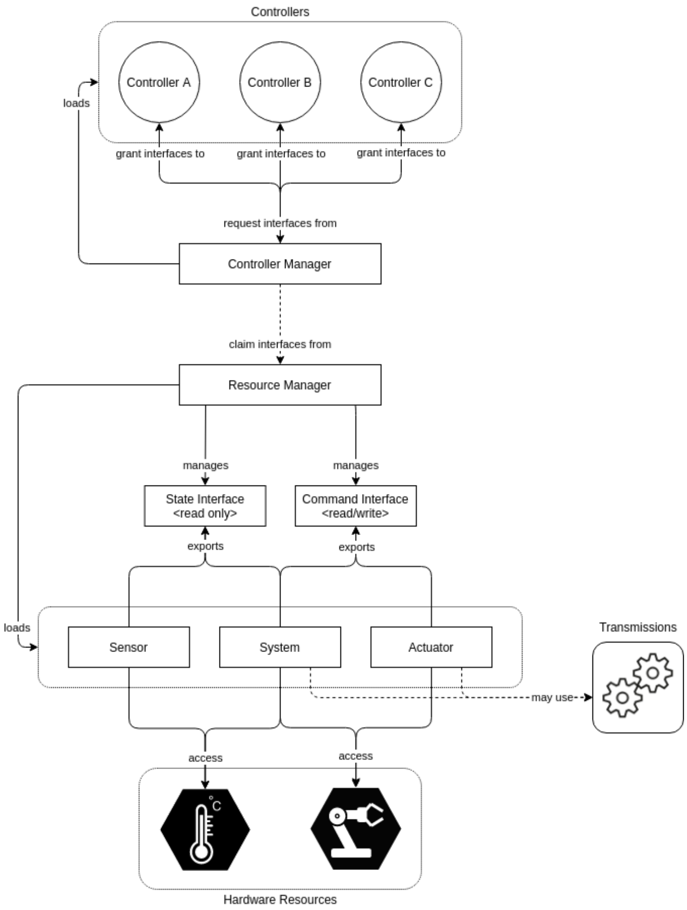
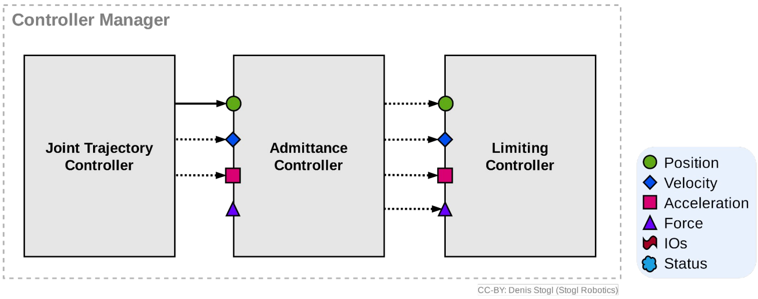


Figure 4.2: ROS 2 Control Architecture from ROS 2 Documentation

In control design, it is common to consider controllers as blocks that can be combined sequentially, as in FIGURE 4.3. The aim of ROS 2 Control's chainable controllers is to enable the framework user to employ this representation in the development and use of controllers. A chainable controller is a controller that can take another controller (or other controllers) as input. Each chainable controller can be used as a standalone controller in a way that is completely transparent to the user. In the FIGURE 4.4 example, the controllers `JointLimitsController`, `diff_drive_controller`, `pid left wheel` and `pid right wheel` are chainable controllers[8]. Besides `ControllerInterface`, the usual parent class for controllers, ROS 2 Control provides `ChainableControllerInterface` as a parent class for chainable controllers. Both parent classes follow essentially the same ideas, but `ChainableControllerInterface` includes the modifications needed to put the controller into chaining mode.

[8] Further details on this example can be found in the ROS 2 Control documentation [3] at https://control.ros.org/master/doc/ros2_control/controller_manager/doc/controller_chaining.html



CC-BY: Denis Stogl (Stogl Robotics)

Figure 4.3: An Example of ROS 2 Chainable Controllers in roscon22

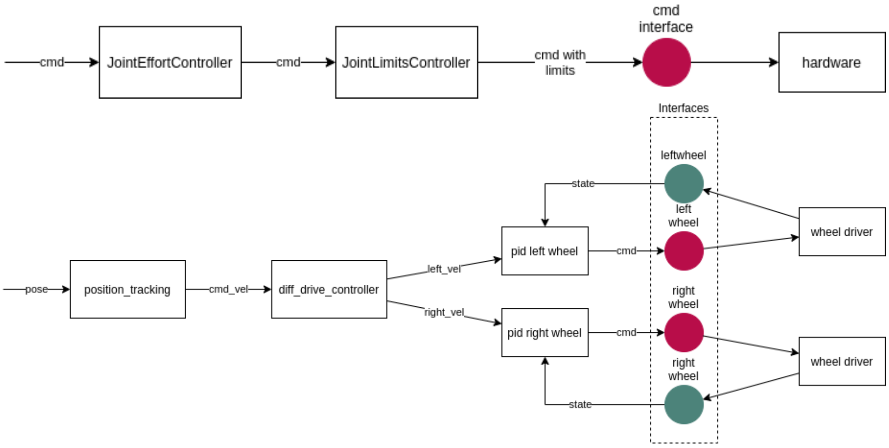


Figure 4.4: Examples of ROS 2 Chainable Controllers from ROS 2 Control Documentation (in development)

One major change to note between chainable and non-chainable mode is the modification of the `ControllerInterface::update` function. This function is called in the control loop by the Control Manager to update the controller's command values. In chainable controllers, the function is split between `ChainableControllerInterface::update_reference_from_subscribers` and `ChainableControllerInterface::update_and_write_commands`. The function `update_and_write_commands` is executed in the control loop and has the same task as `ControllerInterface::update`. `update_reference_from_subscribers` is executed before `update_and_write_commands` only if the controller is not in chained mode. In non-chained mode, the purpose of `update_reference_from_subscribers` is to update the controller's input without reading certain command interfaces provided by another controller. These inputs can be updated via topics, for example.

4.2 Argos Control

On Argos, there are three control modes, as illustrated in FIGURE 4.5. Thrust control is the basic control historically employed on the ROV. The pilot directly requests the desired wrench via the joysticks, and the ROV control computes and commands the thrust allocation of each thruster. Velocity control allows the pilot to be a notch higher in abstraction. This means the pilot is less perturbed by sea currents or umbilical cable disturbances; if the ROV doesn't take the desired direction, the control automatically corrects it. But the drift obtained during the control's correction will never be corrected. The final control mode is position control. The pilot only requests a position, in georeferenced coordinates or relative to the ROV, and the control brings the ROV to this point. Position control also allows the ROV to follow a trajectory represented as a series of positions. While thrust control requires no sensors, velocity control requires a DVL or INS, and position control requires an INS. At least for now, better home-made filtering of sensors like Fuse could enable the INS agnostic ROV to perform well in position control.

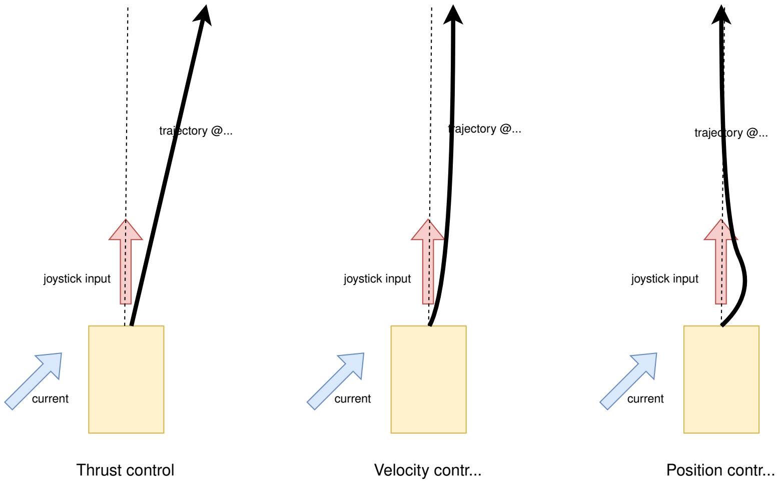


Figure 4.5: Behaviour of the ROV When Using the Different Joysticks Control Modes from Forssea Documentation.

As pilots have a strong experience of thrust control, and to allow the pilot to keep control of Argos in case of a sensor malfunction, the different operating modes do not replace each other, but must coexist. To manage the transition from one control mode to another, an FSM is implemented. This FSM is managed by the MainController component, which lives alongside the ROS 2 controller chain.

4.3 Control Framework Implementation

As we can see in FIGURE A.2 in annexe, the control framework is divided into different controllers, interfaces and work libraries. We present below an overview of the control framework more focused on our concerns, referring to FIGURE 4.6. The framework can be divided into two sequential blocks, the controller `rov_controller` and the controller `thrust_allocation_controller`. The `rov_controller` handles user inputs and calculates the desired wrench. The `thrust_allocation_controller` then receives the wrench, calculates the thrust allocation for each ROV thruster and sends it to the hardware components.

rov_controller needs various components to compute the desired wrench command. The database must store the control framework's inputs and the FSM's internal state (from MainControllerInterface, which handles all inputs, and from StateMachine via ControlReference), and make them available. StateMachine handles the FSM already described above. The BaseController provides all the useful control equations to calculate the wrench, depending on the state, different functions of the BaseController are called by MainController. And MainController is the class that returns the wrench when requested by RovController.

RovController is a ROS 2 controller, it inherits from ControllerInterface and is managed by the Control Manager (Control manager in FIGURE 4.6). RovController uses MainController to get the wrench command and send it to ThrustAllocationController. Then, it uses the work library Allocator to transform the wrench into a thrust allocation.

4.4 Chainable Controllers Implementation

As the chainable controllers of ROS 2 Control were not yet available when the Forssea control framework was created, the RovController class was inheriting from ThrustAllocationController which was inheriting from ControllerInterface. So, before my work, there were two ROS 2 Controllers but they couldn't be used by the Control Manager as controllers at the same time. The solution was to have the Control Manager use the RovController and have the RovController::update function call the ThrustAllocationController::update function. My work on the control framework was to ensure that RovController inherits directly from ControllerInterface and ThrustAllocationController is a ROS 2 Control controller, used by the Control Manager. ThrustAllocationController is a chainable controller, which means that it inherits from ChainableControllerInterface, it can be used as an output of the RovController or listen to a wrench ROS topic.

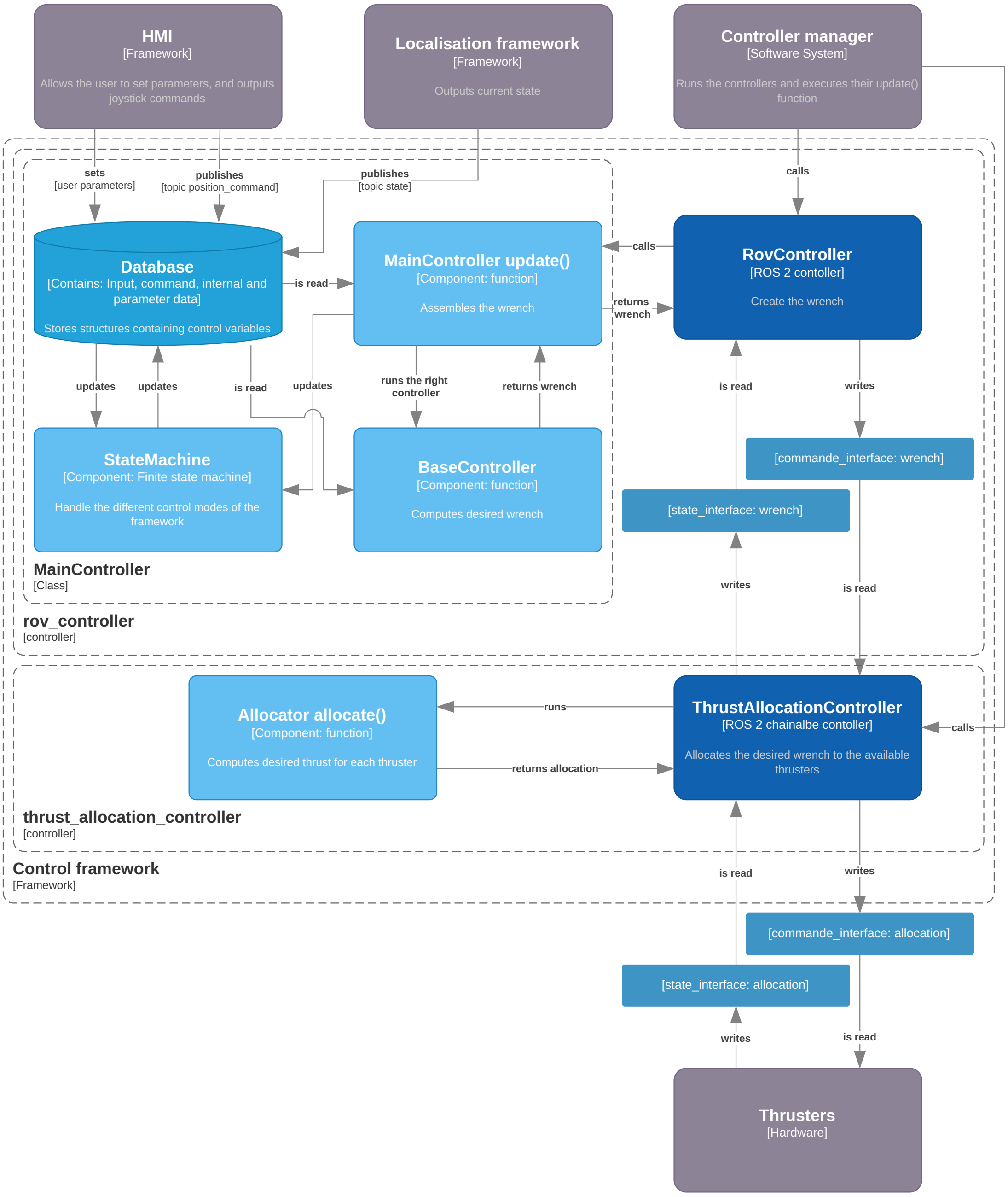


Figure 4.6: New Control Framework Components Simplify Overview focused on Chainable Controllers.

5 Conclusion

Working at an industrial level to make an underwater robot more autonomous was a perfect practical application of my ENSTA Bretagne training. First of all, the general concepts of robotics (ROS, EKF, controllers) but also the specificities of the underwater world (inertial navigation, DVL, FOG INS, use of acoustic waves) had been discovered at the school and were used and consolidated at Forssea. It enabled me to make the link between the theoretical and practical skills acquired during my training at ENSTA Bretagne and industrial practices (fusion algorithm, Lie theory). In order to make Argos autonomous, we had to rely on different areas of research in robotics, as well as mathematics or geodesy. This research was followed by a no less significant work of implementation. My work has sometimes been more research-based and sometimes more implementation-based, but most of the time the skills I've brought to bear have been right on the frontier of both.

The variety of tasks, from (from implementing C++ code, to creating graph constraints and writing and validating tests), I was given enabled me to become aware of the different facets of the project to improve the autonomy of a 200kg ROV. This gave me an overview of the efforts required for successful implementation and a better understanding of the challenges involved. This experience helped me to understand the subtleties of missions that might have seemed simple at first sight (rework code containing bugs, manage different versions of code under development). This experience also enabled me to acquire skills, both in the applied technical dimension (advenced C++ and ROS) and in the ability to step back from each task and consider it in the context of the project as a whole (git management, github pull request, better general understanding).

As my work is part of a more global project, the prospects for development are very interesting. Whether it's the use of sonar in an acoustic SLAM or the finalisation of the implementation of Fuse, the continuation of my work will enable me to finalise the building blocks needed to implement an autonomous Argos.

I believe that this experience will be a real added value for Forssea and also for me in the rest of my career. It complements my previous training in preparatory classes and engineering school. In particular, for the doctoral thesis that I'm starting after my experience at Forssea.

A Annexes

List of Figures

Glossary

API An Application Programming Interface (API) is a particular set of rules and specifications that one software program can follow to access and use the services and resources provided by another particular software program that implements that API. IV, 14, 16, 19

AUV An Autonomous Underwater Vehicle (AUV) is a self-propelled, untethered underwater robot designed to operate autonomously without human intervention. AUVs are equipped with various sensors, navigation systems, and onboard computers that allow them to navigate, collect data, and perform tasks underwater. Unlike ROVs, AUVs operate independently, following pre-programmed paths or using real-time decision-making algorithms to explore and survey underwater environments for scientific research, underwater mapping, oceanography, and other applications. III

DVL A Doppler Velocity Log (DVL) is an underwater sensor used to measure the velocity of a moving vehicle, typically a watercraft or underwater robot, relative to the surrounding water. It operates based on the Doppler effect, which involves measuring the frequency shift of sound waves reflected off particles in the water. DVLs provide crucial information for underwater navigation, enabling precise speed and direction estimation without relying on external signals like GNSS. They are commonly used in various marine applications, including Autonomous Underwater Vehicle (AUV)s, ROVs, and surface vessels, to enhance underwater navigation and mapping accuracy. 2, 3, 11–13, 27

EKF The Extended Kalman Filter (EKF) is one of the most widely used sensor filters in robotics and underwater localization. The EKF can use non-linear evolution and prediction functions and linearise the current estimate for covariances and Kalman gain calculation. V, 4

FAT Factory Acceptance Testing (FAT) is a process where equipment or systems, often industrial or technological in nature, are tested and verified to ensure they meet the agreed-upon specifications, standards, and requirements before they are delivered to the customer. This type of testing occurs at the factory or manufacturing site to ensure that the product is ready for installation and use. IV, 22

FLS Forward Looking Sonar (FLS) is a sonar system designed and implemented to provide real-time, direct-view underwater imagery for navigation, obstacle avoidance and seabed mapping. 15, 16

FOG Fiber Optic Gyroscopes (FOG) INS refers to a type of INS that uses FOG as its primary sensors. FOG are based on the principle of the Sagnac effect, where light propagating in opposite directions along a coiled optical fiber experiences a phase shift due to rotation, which can be used to measure angular velocity accurately. 2, 4

GNSS Global Navigation Satellite System (GNSS) is a system of satellites and ground stations that enables precise positioning, navigation, and timing information worldwide.

The most well-known GNSS is the Global Positioning System (GPS), operated by the United States. Other systems like GLONASS (Russia), Galileo (European Union), BeiDou (China), and NavIC (India) also contribute to the GNSS network. These systems work by using signals from multiple satellites to triangulate a device's location, providing accurate positioning data for various applications, including navigation, mapping, and geolocation-based services. III, IV, 2

HMI A human-machine interface (HMI) is the set of interfaces used by the user to interact with the machine, in our case the robot. The Argos HMI is mainly based on joysticks and a WEB graphic interface. 16, 19

IMU Inertial Measurement Unit (IMU), is a sensor device commonly used in robotics for inertial navigation that combines various sensors such as accelerometers, gyroscopes, and sometimes magnetometers to measure linear acceleration and angular velocity, without the need for external references. IV

INS Inertial Naviagation System (INS) is a software layer to filter data from a Inertial Measurement Unit (IMU) and often other various sensors to obtain linear and angular position and velocity. INS are used when GNSS positionning is not sufficient, beceaus of its inaccuracy are because the environment do not allow satelite communication (as underwater environment). As all dead reckoning methode, INS need to inegrate two time linear acceleartion to get position, this mean their major weakness is its dirft. III, 2, 4, 13, 27

PFE Projet de Fin d'Étude (PFE) is the last semester graduation project. 1

R&D Research and Development (R&D). 3

RCLCPP RCLCPP is the name of ROS 2 API for C++. 19

ROS Robot Operating System (ROS) is an open-source robotics middleware that provides a large community and tends to become a standard in robotics research laboratories. The main interest of ROS is to manage inter-process communication, standardise data communication and coding style, and facilitate the integration of the number of open-source packages available. IV, 2

ROV A Remotely Operated underwarter Vehicle (ROV) is an underwater robotic device controlled from the surface through a umbilical calbe connection. Equipped with cameras, sensors, and often mechanical arms, ROVs are deployed in underwater environments to perform tasks that are hazardous, inaccessible, or logistically challenging for humans. They find applications in scientific research, offshore industries, marine exploration, and underwater infrastructure maintenance, providing a means to visually inspect, collect data, and manipulate objects beneath the water's surface with precision and control. Argos is the name of Forssea's ROV model. II, III, 1–4, 11, 13–15, 26, 27

SAT Sea Acceptance Testing (SAT) is a phase in maritime equipment and vessel testing, occurring after FAT. During SAT, the equipment or vessel is tested at sea to validate its performance, functionality, and adherence to specifications in real-world marine conditions. This testing is essential to ensure the equipment operates safely and effectively before deployment, contributing to its reliability and suitability for operational use in maritime environments. 22, 23

SLAM Simultaneous Localization and Mapping (SLAM) is a computational technique used in robotics to create maps of an unknown environment while also determining the robot's position within that environment. SLAM is particularly useful when a robot needs to navigate in an environment without prior knowledge of its surroundings. The process involves integrating data from various sensors, such as cameras, lidars, and odometry, to build a map of the environment and simultaneously estimate the robot's location within that map. SLAM algorithms are essential for enabling robots to autonomously explore and navigate in dynamic or unfamiliar environments, such as in autonomous vehicles, drones, and mobile robots. 1, 2, 13, 15

UKF The Unscented Kalman Filter (UKF) is a major alternative to EKF in Kalman-based filters. In the UKF, the Kalman covariances and gain are not calculated by linearising the evolution and prediction functions, but by using an uncentred transform to calculate them. The uncentred transform evaluates the functions at different points to estimate the covariances and then calculate the Kalman gain. 4

UUID Universally Unique Identifier (UUID), is a 128-bit identifier that is used to uniquely identify information in computer systems. It's designed to be globally unique across time and space, meaning that the probability of two UUIDs being the same is extremely low. UUIDs are commonly represented as a sequence of hexadecimal digits separated by hyphens. They have various applications, such as identifying resources in distributed systems, generating unique filenames, and ensuring data integrity in databases. See 2.3.2 for Fuse context. 5, 8

References

[1] Wgs 84 implementation manual. *Eurocontrol & IfEN*, Version 2.4, 1998.

[2] Fuse github. *Locus Robotics*, 2023.

[3] Ros 2 control documentation. ROS Control, 2023.

[4] Ros 2 humble documentation. ROS, 2023.

[5] The Ceres Solver Team A. Sameer, M. Keir. Ceres Solver. *https://github.com/ceres-solver/ceres-solver*, 2.1, 2022.

[6] The Ceres Solver Team A. Sameer, M. Keir. Ceres Solver Modeling Non-linear Least Squares. *http://ceres-solver.org/nnls_modeling.html*, 2022.

[7] The Ceres Solver Team A. Sameer, M. Keir. Ceres Solver Solving Non-linear Least Squares. *http://ceres-solver.org/nnls_solving.html*, 2022.

[8] National Geospatial Intelligence Agency. World geodetic system 1984 datasheet. *United Nations Office for Outer Space Affairs, United Nations*, 2021.

[9] J.L. Blanco. A tutorial on $SE(3)$ transformation parameterizations and on-manifold optimization. *University of Malaga*, (Technical report 012010), 2022.

[10] J.L. Blanco-Claraco. A tutorial on SE(3) transformation parameterizations and on-manifold optimization. *CoRR*, abs/2103.15980, 2021.

[11] A. El Jawad A. Bougois. Localization and frame convention in couronne. *Forssea Robotics*, (v1.0), 2023.

[12] Tully Foote. tf: The transform library. *Open Source Robotics Foundation*, Mountain View, CA 94043.

[13] T. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley Sons, Ltd, 2011.

[14] J. Hajjami. Sonar: Sound navigation and ranging. *Forssea Robotics*, 2023.

[15] J. Deray J. Solà and D. Atchuthan. A micro lie theory for state estimation in robotics. *Institut de Robòtica i Informàtica Industrial, CSIC-UPC*, (Technical Report IRI-TR-18-01), 2018.

[16] M. Legris. *Navigation sous-marine*. ENSTA Bretagne, 2020.

[17] T. Laidlow M. Burri G. Nuetzi P. Fankhauser D. Bellicoso C. Gehring S. Leutenegger M. Hutter M. Bloesch, H. Sommer and R. Siegwart. A primer on the differential calculus of 3d orientations. *ETH Zurich*, (Report 10.3929/ethz-a-010666114), 2016.

[18] B. Magyar. A practitioner's guide to ros2_control. In *ROSCon*, 2022.

[19] W. Meeussen. Coordinate frames for mobile platforms. ROS REP 105, 2010.

[20] W. Meeussen V. Pradeep A. R. Tsouroukdissian J. Bohren D. Coleman B. Magyar G. Raiola M. Lüdtke S. Chitta, E. Marder-Eppstein and E. Fernandez Perdomo. ros_control: A generic and simple control framework for ros. *Journal of Open Source Software*.

[21] B. Gerkey C. Lalancette W. Woodall S. Macenski, T. Foote. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*.

[22] M. Purvis T. Foote. Standard units of measure and coordinate conventions. ROS REP 103, 2014.

A.1 Sonar Run Mode FSM

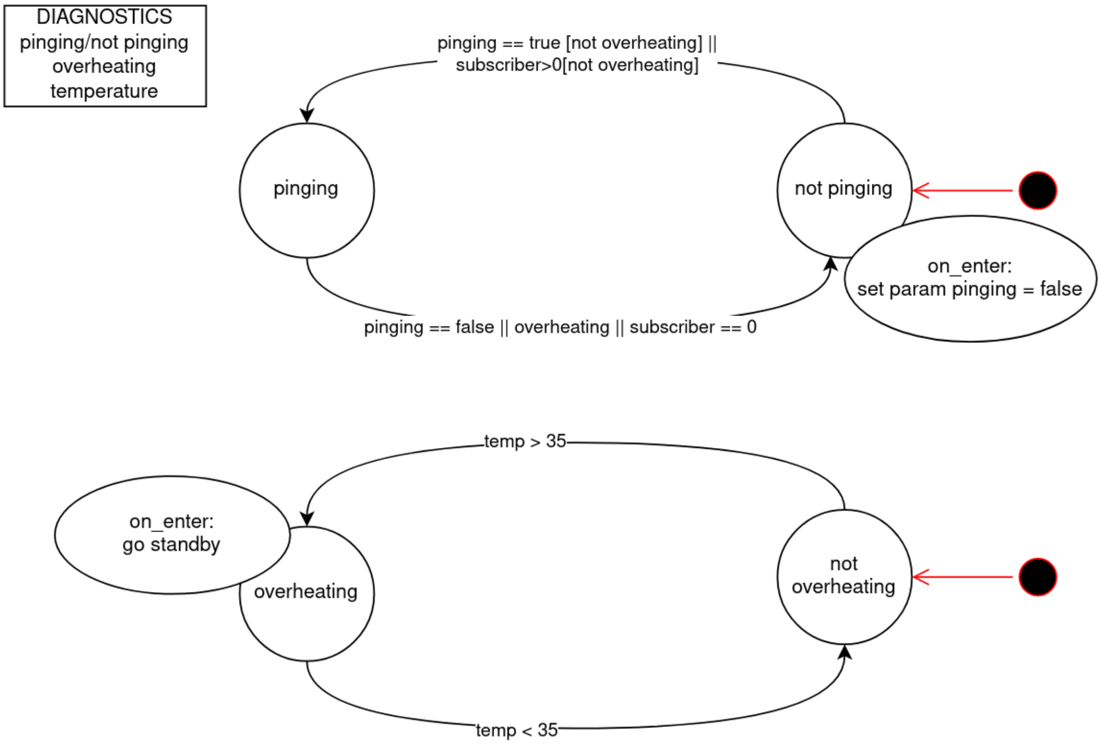


Figure A.1: Finite State Machine for the Running Mode of `oculus_sonar`

A.2 Control Framework Overview

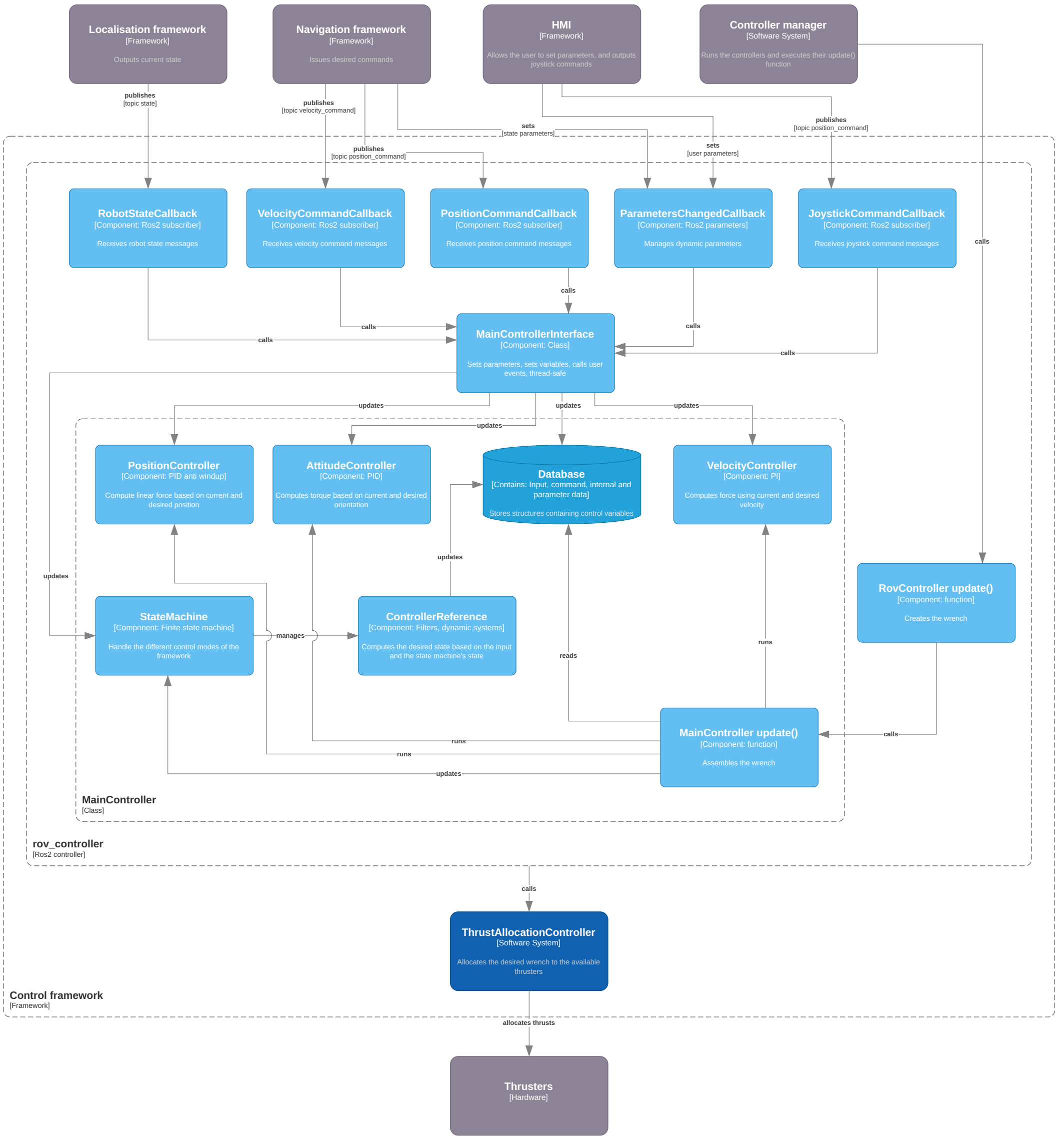

Localisation framework
[Framework]
Outputs current state
Navigation framework
[Framework]
Issues desired commands
HMI
[Framework]
Allows the user to set parameters, and outputs joystick commands
Controller manager
[Software System]
Runs the controllers and executes their update() function
publishes
[topic state]
publishes
[topic velocity_command]
publishes
[topic position_command]
sets
[state parameters]
sets
[user parameters]
publishes
[topic position_command]
RobotStateCallback
[Component: Ros2 subscriber]
Receives robot state messages
VelocityCommandCallback
[Component: Ros2 subscriber]
Receives velocity command messages
PositionCommandCallback
[Component: Ros2 subscriber]
Receives position command messages
ParametersChangedCallback
[Component: Ros2 parameters]
Manages dynamic parameters
JoystickCommandCallback
[Component: Ros2 subscriber]
Receives joystick command messages
calls
calls
calls
calls
calls
calls
MainControllerInterface
[Component: Class]
Sets parameters, sets variables, calls user events, thread-safe
updates
updates
updates
updates
updates
PositionController
[Component: PID anti windup]
Compute linear force based on current and desired position
AttitudeController
[Component: PID]
Computes torque based on current and desired orientation
Database
[Contains: Input, command, internal and parameter data]
Stores structures containing control variables
VelocityController
[Component: PI]
Computes force using current and desired velocity
updates
StateMachine
[Component: Finite state machine]
Handle the different control modes of the framework
manages
ControllerReference
[Component: Filters, dynamic systems]
Computes the desired state based on the input and the state machine's state
reads
runs
RovController update()
[Component: function]
Creates the wrench
MainController update()
[Component: function]
Assembles the wrench
calls
runs
runs
updates
MainController
[Class]
rov_controller
[Ros2 controller]
calls
ThrustAllocationController
[Software System]
Allocates the desired wrench to the available thrusters
Control framework
[Framework]
allocates thrusts
Thrusters
[Hardware]

Figure A.2: Control Framework Components Overview from Forssea Documentation.

A.3 Images of Forssea Materials

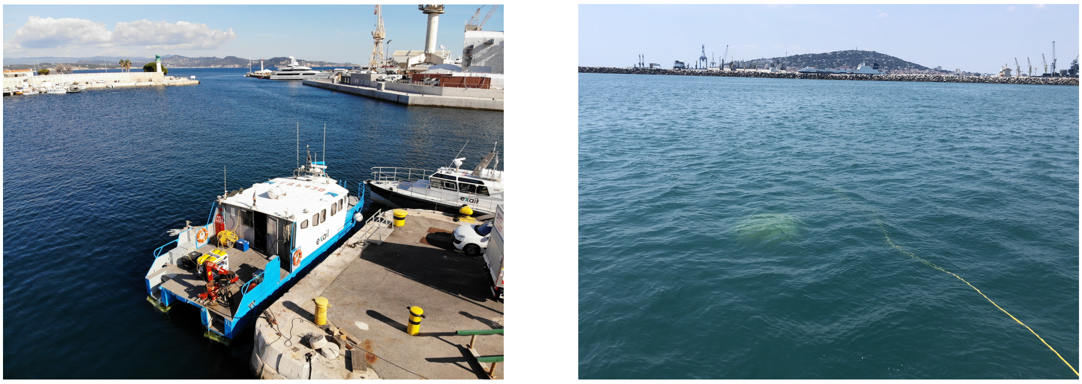


Figure A.3: Argos in operation

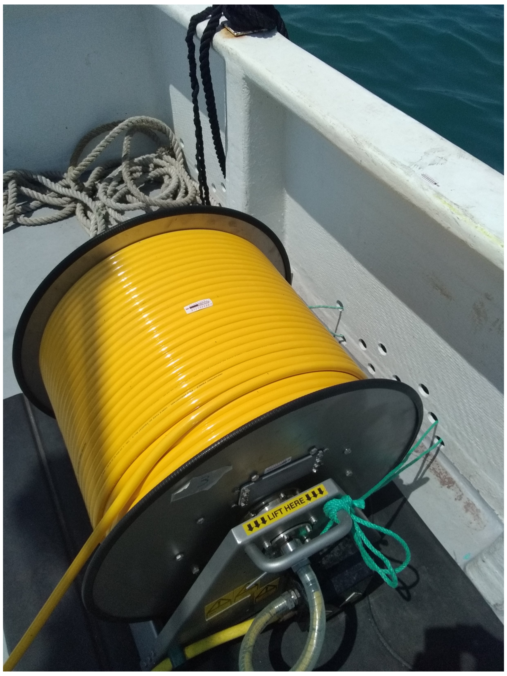


Figure A.4: Argos's Winch

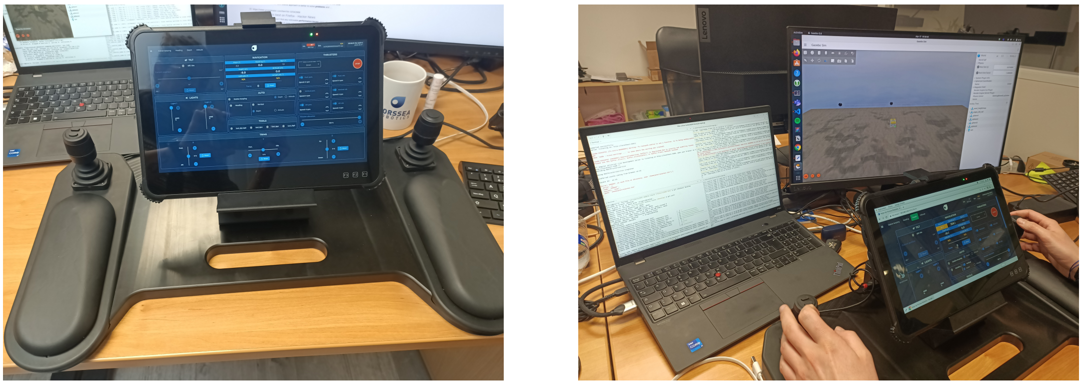


Figure A.5: Hand Controller

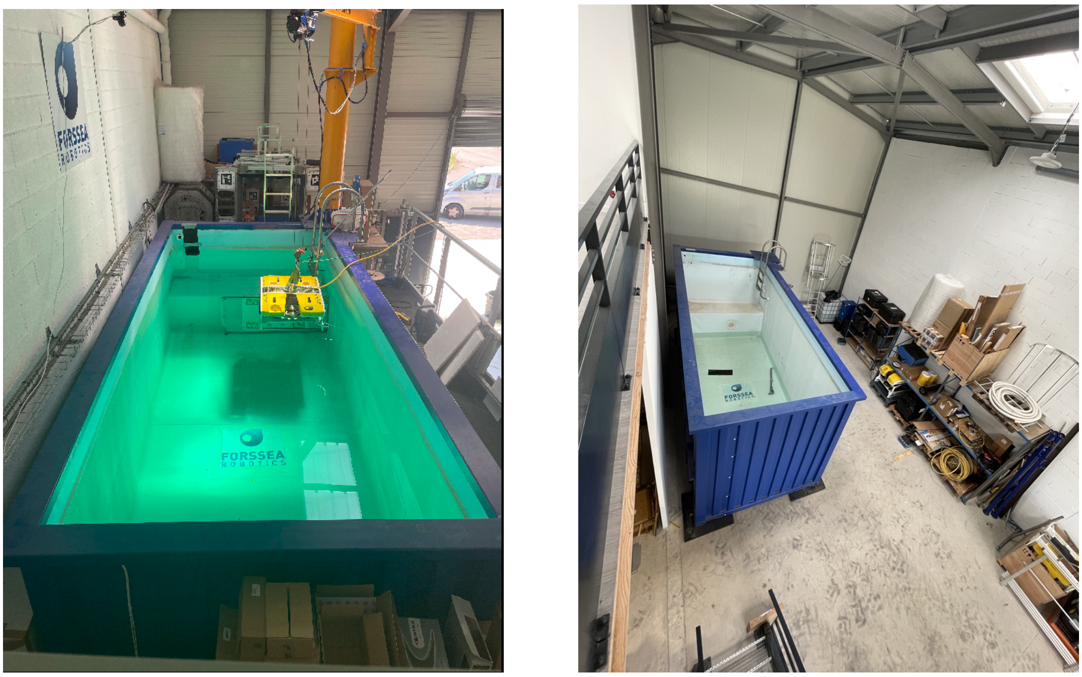
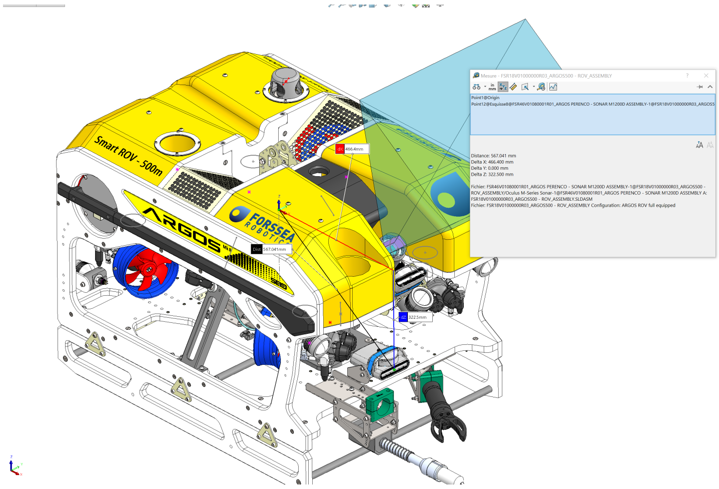


Figure A.6: Test pools (in the old workshop on the left and in the new workshop on the right)



Mesure - FSR18V01000000R03_ARGOS500 - ROV_ASSEMBLY

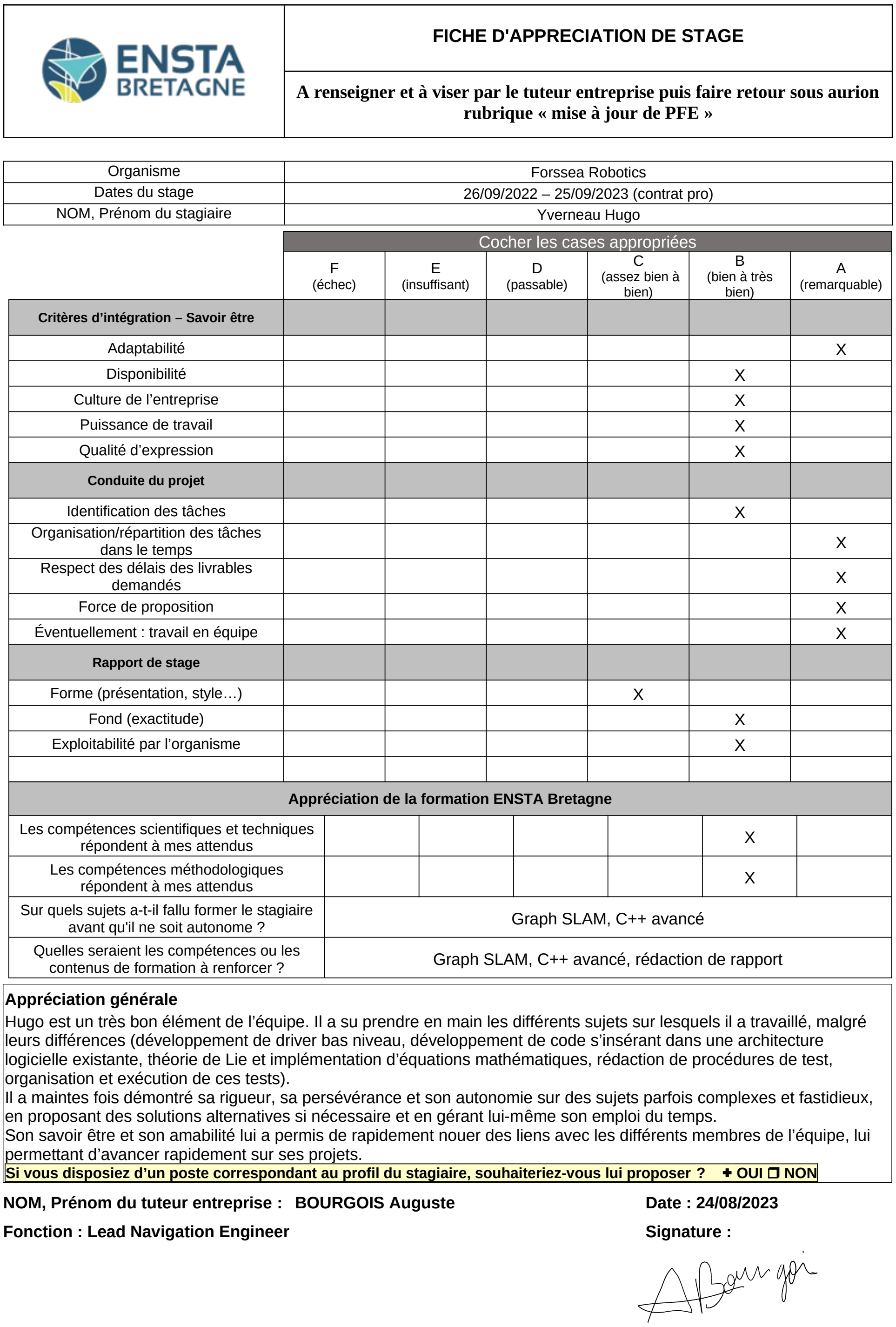
Point1@Origin
Point12@Esquisse8@FSR46V01080001R01_ARGOS PERENCO - SONAR M1200D ASSEMBLY-1@FSR18V01000000R03_ARGOS5

Distance: 567.041 mm
Delta X: 466.400 mm
Delta Y: 0.000 mm
Delta Z: 322.500 mm

Fichier: FSR46V01080001R01_ARGOS PERENCO - SONAR M1200D ASSEMBLY-1@FSR18V01000000R03_ARGOS500 - ROV_ASSEMBLY/Oculus M-Series Sonar-1@FSR46V01080001R01_ARGOS PERENCO - SONAR M1200D ASSEMBLY A: FSR18V01000000R03_ARGOS500 - ROV_ASSEMBLY.SLDASM
Fichier: FSR18V01000000R03_ARGOS500 - ROV_ASSEMBLY Configuration: ARGOS ROV full equipped

Figure A.7: Measurement of Sonar Lever Arm on Argos SolidWorks CAD Model

A.4 Evaluation



ENSTA BRETAGNE	**FICHE D'APPRECIATION DE STAGE**
	A renseigner et à viser par le tuteur entreprise puis faire retour sous aurion rubrique « mise à jour de PFE »

Organisme	Forssea Robotics
Dates du stage	26/09/2022 – 25/09/2023 (contrat pro)
NOM, Prénom du stagiaire	Yverneau Hugo

	Cocher les cases appropriées					
	F (échec)	E (insuffisant)	D (passable)	C (assez bien à bien)	B (bien à très bien)	A (remarquable)
Critères d'intégration – Savoir être						
Adaptabilité						X
Disponibilité					X	
Culture de l'entreprise					X	
Puissance de travail					X	
Qualité d'expression					X	
Conduite du projet						
Identification des tâches					X	
Organisation/répartition des tâches dans le temps						X
Respect des délais des livrables demandés						X
Force de proposition						X
Éventuellement : travail en équipe						X
Rapport de stage						
Forme (présentation, style…)				X		
Fond (exactitude)					X	
Exploitabilité par l'organisme					X	
Appréciation de la formation ENSTA Bretagne						
Les compétences scientifiques et techniques répondent à mes attendus					X	
Les compétences méthodologiques répondent à mes attendus					X	
Sur quels sujets a-t-il fallu former le stagiaire avant qu'il ne soit autonome ?	Graph SLAM, C++ avancé					
Quelles seraient les compétences ou les contenus de formation à renforcer ?	Graph SLAM, C++ avancé, rédaction de rapport					

Appréciation générale

Hugo est un très bon élément de l'équipe. Il a su prendre en main les différents sujets sur lesquels il a travaillé, malgré leurs différences (développement de driver bas niveau, développement de code s'insérant dans une architecture logicielle existante, théorie de Lie et implémentation d'équations mathématiques, rédaction de procédures de test, organisation et exécution de ces tests).
Il a maintes fois démontré sa rigueur, sa persévérance et son autonomie sur des sujets parfois complexes et fastidieux, en proposant des solutions alternatives si nécessaire et en gérant lui-même son emploi du temps.
Son savoir être et son amabilité lui a permis de rapidement nouer des liens avec les différents membres de l'équipe, lui permettant d'avancer rapidement sur ses projets.

Si vous disposiez d'un poste correspondant au profil du stagiaire, souhaiteriez-vous lui proposer ? ✚ OUI ❐ NON

NOM, Prénom du tuteur entreprise : BOURGOIS Auguste **Date : 24/08/2023**

Fonction : Lead Navigation Engineer **Signature :**

A Bourgoi

Figure A.8: Forssea Evaluation of the PFE