CONSTRUCTOR UNIVERSITY

Comparative Analysis of Simulation Environments for Autonomous Underwater Vehicles: Sensor Integration, Physics-Based Model and Scenario Implementation

by

Jose Carlos Tejeda Guzman

Bachelor Thesis in Robotics and Intelligent Systems

Submission: May 16, 2024 Supervisor: Prof. A. Birk

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Tejeda Guzman, Jose Carlos
Matriculation number	30005645
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

.....

Date, Signature

Abstract

This paper presents a comparative analysis between two main simulation environments: HoloOcean and Stonefish. It considers the physics-based model, implementation of sensors for underwater activities, and complexity to modify the environment to demonstrate the strengths and weaknesses of each simulator. The goal is to provide insight into research projects in which these simulators might yield better results. The evaluation involves creating different environments where Autonomous Underwater Vehicles (AUVs) are equipped with various sensors to perform underwater tasks. By collecting information through the sensors used in each simulation and observing the overall behavior of the AUVs, a comparison between these two simulators is conducted. Given that acoustic images are widely used to develop algorithms for improving vehicle autonomy, sonar systems are evaluated more deeply than any other sensor.

Both simulation environments showcase excellent rendering capabilities and are highly advanced in creating underwater scenarios for autonomous vehicles. However, the intrinsic nature of each simulator yields remarkable results: HoloOcean, leveraging an external graphics engine, produces sensor images of superior quality and resolution, while Stonefish excels in simulating water physics and provides more advanced underwater sensors.

Contents

1 Introduction

The development of Autonomous Underwater Vehicles (AUVs) is a complex field where every decision could lead to success or failure, potentially consuming significant time and resources. Therefore, it is crucial to minimize the risk of errors by creating a simulation environment that mirrors real-world scenarios. This allows testing the robot's software without the risk of losing or damaging the hardware. Additionally, investing time in finding the most optimal simulation environment (according to the needs and tasks that the AUV is planned for) is essential. This ensures addressing behavioral mistakes or the absence of sensor implementations that may not have been considered during the initial design phase but could emerge during testing.

Nowadays, it may appear relatively simple to find a simulation environment that allows the user to conceive the desired scenarios, adapt every component that the vehicle might use to perform optimally and access necessary documentation for practical underwater simulation. However, selecting the correct platform to conduct the necessary tests is a rigorous task that requires a lot of time in research and investigation, as there are numerous options boasting specific characteristics. Knowledge about the simulation environment and comprehending every operation executed by the simulator is crucial since this understanding significantly impacts testing results and subsequently, the final implementation of the robot. Therefore, having extensive documentation that explains in detail the operations performed for the sensor to display the corresponding data and the general structure of every simulator is important.

Given that the primary objective of every underwater simulation environment is to recreate as realistically as possible the scenario where the AUV is planned to be introduced, accurately representing the physics and general properties of this scenario is essential. Consequently, it is vital to analyze the realism of the physics-based model that every simulation environment offers and observe how the world may influence the behavior of the vehicle during the simulation.

In addition to what has been said, the recent exponential increase in the use of Machine Learning and Computer Vision as solutions to achieve the complete autonomy of Unmanned Vehicles emphasizes the necessity of data collection. Gathering data is crucial to initiate the development of algorithms essential for the success of the Autonomous Underwater Vehicle's decision-making processes. Therefore, the complexity of vision sensors such as cameras, forward-looking, mechanical and side-scan sonar, the accessibility to these while creating a vehicle and the quality of the output displayed at the moment of running the simulation are crucial factors to take into consideration when deciding on the simulation environment that best fits the expected outcome of the project.

Despite the fact that vision sensors are extremely useful when evaluating the actual state of the vehicle and providing a better understanding of the volume, brightness and physical characteristics of all the components within the environment, they are just a small part of a larger group of devices needed to achieve a very detailed simulation. Some sensors necessary for measuring basic properties such as position, linear velocity, angular velocity, navigation ground truth and water velocity are equally important. Therefore, it is imperative to closely assess the performance of sensors (also known as Scalar or Measurement Sensors due to their capacity to return a scalar value once the computational process is complete) like the Inertial Measurement Unit (IMU), Doppler Velocity Log (DVL), Compass, and Pose Sensor

To conduct a proper comparative analysis between simulation environments, collecting data generated during the extensive implementation of underwater scenarios is crucial. Consequently, the AUV must be tested in various worlds with variations in their sensors and parameters to obtain the necessary information. This approach ensures that the data collected throughout the various tests is sufficient to derive concrete solutions.

In order to conduct a deeper analysis on sensors implementation, complexity, physics model, image quality, accessibility, underwater agents and their interaction inside the world, as well as path-following capabilities, a comparison of all these aspects will be carried out. This comparison will involve two main simulation environments:

1. *"HoloOcean* [1] *: An open-source Underwater Robotics Simulator"* [1]
2. *"Stonefish* [2] *: An advanced simulation tool developed for marine robotics"* [2].

These two robotics simulators were selected for further evaluation due to the similarities they present in terms of the number of sensors that can be implemented on the robot, sonar computation, physics implementation beyond visual appreciation, rendering quality and the output displayed by the sonar and camera systems.

To fully explore and maximize the benefits of the Stonefish simulator, the already implemented Autonomous Underwater Vehicle "GIRONA 500" [3] will be equipped with measurement and vision sensors. The "GIRONA 500" is a lightweight reconfigurable vehicle with a versatile propulsion system, capable of transitioning from a 3-thruster configuration and three degrees of freedom to an 8-thruster configuration and being fully actuated. To avoid computational issues arising from insufficient hardware power, the sensors will not be used all at once but gradually, as different scenarios are created. Every time a new scenario is generated, the AUV will follow a specific path, either by implementing an algorithm such as carrot-chasing or a simple straight line. Throughout the entire simulation, the sensors in use will continuously collect data for further evaluation.

On the other hand, HoloOcean includes a torpedo, HoveringAUV, and a surface vehicle within its open-source library. To analyze the capabilities of the simulator on underwater activities, the HoveringAUV will be used across different testing scenarios. Since HoloOcean utilizes an external graphics engine to compute sonar images and camera output, some scenarios are specifically created to test sonar performance and avoid potential crashing problems caused by the multiple implementation of sensors. Following the same structure used to analyze Stonefish, the sensors used throughout the simulations will collect data for further evaluation.

[1] *E. Potokar and S. Ashford and M. Kaess and J. Mangelson,"HoloOcean: An Underwater Robotics Simulator, Proc. IEEE Intl. Conf. on Robotics and Automation, ICRA, Philadelphia, USA, May 2022*

[2] *Patryk Cieślak, "Stonefish: An Advanced Open-Source Simulation Tool Designed for Marine Robotics, With a ROS Interface", In Proceedings of MTS/IEEE OCEANS 2019, June 2019, Marseille, France*

2 Statement and Motivation of Research

The primary objective of this study is to compare two main simulation environments highly supported for marine robotics activities, particularly those related to maximizing the autonomy of underwater vehicles. Through the assessment of sensors, adaptability and operational behavior of the Autonomous Underwater Vehicle implemented and scenario visualization, the weaknesses, strengths and efficacy of both simulators will be examined. This will provide insights into situations and projects where either HoloOcean or Stonefish might yield better results. With this information, future research projects in marine robotics will benefit from a comprehensive comparison of these two high quality underwater simulators, optimizing the search for a functional and stable simulation environment. In cases where both environments exhibit similar weaknesses, potential optimizations of the process will be sought to make a relevant contribution to the current state-of-the-art in the simulation of marine robotics.

The maritime environment is known for its high unpredictability and constant changes, making underwater tasks particularly challenging and often resulting in lower success rates than anticipated. Therefore, a simulation environment that provides the opportunity to incorporate ocean properties such as currents offers a more accurate representation of realistic underwater scenarios. Consequently, understanding the physics affecting the vehicle and accurately simulating various underwater phenomena within the simulator provide users with the capability to implement control system algorithms, thereby enabling a higher degree of vehicle autonomy.

A complete implementation of sensors is extremely important for the creation of fully autonomous underwater vehicles. Therefore, accurately simulating sensors that assist marine robots in obtaining a clear underwater perception is crucial for algorithm implementation. Over the past years, the development of realistic sonar imagery has proven highly significant results for mapping [41], localization [42] and object recognition [12]. The integration of sonar sensors into autonomous activities arose from the need for sensors that offer broader visibility over long distances, where optical cameras struggle due to high turbidity or low lighting. Consequently, there is a growing need for realistic simulation of acoustic images produced by various sonar types such as Side Scan Sonar, Forward-Looking Sonar, Mechanical Sonar and others. A deep analysis of sonar rendering techniques and the level of realism provided by simulation environments like HoloOcean and Stonefish aims to provide insights into the effectiveness of these simulators for creating acoustic images.

2.1 Stonefish

The foundation of Stonefish was conceived with the aim of building a simulation tool capable of rendering high-quality scenarios. This initiative arose from the surprising number of simulators that lack the incorporation of crucial properties essential for the development of marine vehicles, including those related to water physics such as buoyancy and geometry-based hydrodynamics. The motivation to develop a simulator that fulfilled these high computational power requirements without sacrificing good visualization led to the creation of Stonefish, a C++ Library. This framework utilizes the core functionality of the "*Bullet Physics Library*" [3] as a basis for its physics engine due to its capability to detect and calculate collisions, soft body, rigid body and vehicle dynamics.

Now, with the correct physics-based model, the necessity to generate the right visualization of the world emerged. Due to the heavy calculations that the graphics processing units (GPUs) might require to accelerate the rendering process of sensors, vehicle characteristics, and lighting, it is fundamental to find a graphics tool that does not demand excessive hardware resources. Therefore, to avoid expensive hardware components, bulkiness, and processing delays, Stonefish does not utilize any 3D graphics engine. Consequently, the absence of external graphics engine during rendering encourages the search for new techniques to create a realistic representation of the expected 3D marine world.

[3] *Bullet Physics [4] is primarily known as a C++ library, but in recent years, the development of a Python module has expanded its utility. This Python module enables the use of Bullet Physics for implementation in machine learning, robotics simulation, games, and more within the mentioned programming language.*

The rendering pipeline of this simulator utilizes the OpenGL[5] application programming interface (API), which works simultaneously with GLM, a simple but convenient mathematics library for software rendering, especially raytracing [6], and SDL, a cross-platform development library often used for its low-level access to keyboard, joystick, and graphics hardware. OpenGL successfully executes the rendering process due to its use of shader-based techniques. These shaders [4] are compiled and linked together, allowing for the creation of endless specific stages in the rendering pipeline. This approach, coupled with the closely integrated nature of the Bullet physics engine, enables a lightweight and highly efficient solution.

In addition to the previous statements, Stonefish stands out among the numerous simulation environments available for underwater vehicles due to the consistent support and updates provided by its developers and the marine robotics community. This aspect is crucial to consider since continuity allows researchers and users to develop and utilize an optimal and up-to-date simulation environment. While other simulation environments such as UWSim[7] may appear attractive initially due to their popularity and documentation, the lack of ongoing improvement and support inevitably results in an outdated simulation. Furthermore, the initial documentation combined with the research projects [8] in which Stonefish has been involved provides a deep insight into the correct implementation of the library and the potential limitations that the simulator might have.

Visibility and comprehensive guides about simulation environments add value when deciding on the best environment. Even though some alternatives like MARUS [9] (Marine Robotics Unity Simulator) or UNav-Sim [10] may appear more attractive due to their rendering quality and ROS support, inefficient documentation and limited project involvement can create difficulties during testing.

2.1.1 Sonar and Camera systems

Recreating acoustic images is always challenging due to the complex nature of the tasks involved. In the case of sonar (Sound Navigation And Ranging), the modeling of acoustics within a maritime scenario, combined with the rendering of data, often results in distorted images of poor quality. The library offered by Stonefish allows the implementation and rendering of the Forward Looking Sonar (FLS), Side Scan Sonar (SSS), and scanning imaging sonar (MSIS). OpenGL is capable of computing [5] these visual systems using two specific classes available in Stonefish: OpenGLSonar and OpenGLView. These classes can be customized by the user, providing the opportunity to modify parameters such as the position of the sonar eye in world space *[m]*, resolution of the sonar display *[px]* the horizontal and vertical field of view of the sonar *[deg]*, range and direction.

The technique used to deliver high-quality resolution images without the support of any graphical engine is known in the rendering community as ”Hybrid Rasterization and Ray Tracing” [6]. In this technique, the first intersections of the rays are computed by rasterization, which is commonly used to display objects established in a three-dimensional world on a two-dimensional screen. Rasterization achieves this by using triangles and other polygons to describe the 3D objects that are part of the scene, with every corner of the triangle used to keep information about the objects to later display this information as pixels in the 2D world.

During the second part of the technique, the reflective areas of the simulation are computed by ray tracing [11]. This rendering technique, quite the opposite of rasterization due to its intrinsic computational process, generates better lighting effects, such as reflection. The methodology behind ray tracing can be explained in a few sentences as the computation pixel-by-pixel of the primitives around the 3D world that are hit by the first intersection of rays, creating a computational cost considerably higher than rasterization. The combination of these techniques grants a high-quality simulation where only the objects reflected by the first rays are computed by ray tracing, while the rest of the scenario is calculated with rasterization.

[4] *”Independent compilation units written in this language are called shaders” John Kessenich, Dave Baldwin, Randi Rost, ”The OpenGL® Shading Language”, Document Revision 7, Language Version: 4.50, 09 May 2017, Editor: John Kessenich, Google, Version 1.1, p. 6.*

[5] *Once the computational process concludes, the vision sensors display a matrix of data as output.*

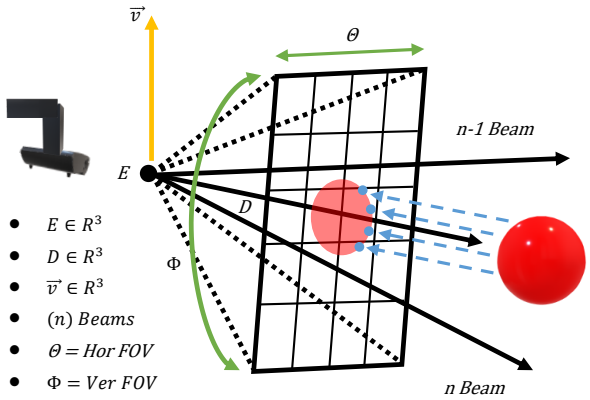


Figure 1: Example of the Sonar Image Rendering using Raytracing techniques

The recreation of images taken by the camera works with a similar principle to the one used for rendering the data collected by the sonars. The main difference can be found in the matrix data displayed by the sensor, where the range of RGB is wider.

2.1.2 Sensors Attached to Links

These sensors are typically attached to the body of the Autonomous Underwater Vehicle to measure its motion and the environmental characteristics, providing valuable data for determining the current state of the robot within the scenario. They constitute the largest group of sensors available in the Stonefish Library, sharing common characteristics such as sensor update frequency *[Hz]* and history length (unlimited, specific length, or no history at all). While all scalar sensors return a single scalar value, some may return a vector depending on the number of samples the sensors are capable of calculating. Each sensor, including vision and joint sensors, features a function where specific noise characteristics can be set if standard deviation needs to be taken into account for any reason.

Accelerometer: Measures the linear acceleration along the three perpendicular axes (X, Y, Z) of the body to which it is attached. Users can set a range for the maximum and minimum acceleration. The resulting scalar vector consists of three channels, one for each axis. To calculate the acceleration class, the cross product between the origin vector translation and the pose of the body in the world frame is utilized.

Gyroscope: As a complement to the accelerometer, this sensor is responsible for calculating the angular velocity of the link around the perpendicular axis. It also allows setting the measurement range, standard deviation and measurement bias. The method to obtain the three-dimensional scalar output vector involves calculating the inverse 3x3 rotation matrix of the body's pose in the world frame. This matrix is then used to compute the angular velocity of the body to which the sensor is attached. If the bias measurement is non-zero, the output vector is affected by the bias at the end of the process.

Inertial Measurement Unit (IMU): The inertial sensor responsible for measuring the angular velocity, orientation, and linear acceleration of the link follows the same principle as the previous sensors, allowing the modification of the ranges and standard deviation. Additionally, the yaw angle drift rate can be specified to achieve the desired result. The angular velocity is computed similarly to the gyroscope, while the linear acceleration follows the same procedure as the accelerometer. On the other hand, the orientation is computed by calculating the rotation matrix [14] based on the pose of the body in the world frame. Subsequently, the Euler angles are derived from this matrix [15], providing scalar values for yaw, pitch, and roll. The final sample dimension of the Inertial Measurement Unit is composed of 9 channels, displaying the current angular velocity, linear acceleration, and orientation of the body.

Odometry: Following the same concept as any other odometry system [16], the principal task of this virtual sensor (virtual since it is not a sensor itself but uses data collected by other means) is to display the localization of the body to which it is attached. In Stonefish, the odometry sensor is primarily used to obtain the navigation ground truth while additionally offering the capability to simulate errors in the navigation systems during simulation.

The Odometry sensor displays the longest vector output from the scalar sensors, where a 12-dimensional array stores the values for the position, velocity, angular velocity, and orientation along the three axes. To obtain the ground truth of the position, it returns the origin vector translation at that point. For velocity, it calculates the linear velocity of the body using the origin vector translation. Regarding orientation, the sensor computes a quaternion with the angles [0, 2π] of rotation of the body in the world frame and the noise added by the user.

Compass (FOG): The technique used to simulate the compass in Stonefish is quite similar to other simulation environments [20], where the sensor measures the direction in which it is moving along the x-axis. Opposite to odometry, this sensor only outputs a single channel, where the yaw is determined through the computation of the matrix represented as Euler angles around Y, X, Z of the sensor frame.

Global Positioning System (GPS):The global positioning system for underwater robotics presents complexity due to the inherent characteristics of water and the properties of the signals emitted by positioning systems. In marine robotics, various methods have been devised to ensure the accurate implementation of GPS [21], such as underwater acoustic positioning systems through communication devices like Ultra Short BaseLine (USBL). USBL includes a ping function used to transmit information to surface buoys, which then compute this data into Earth-related coordinates. However, since Stonefish does not provide any buoys to implement this communication system, the use of this sensor can only be implemented when the link is above the water level.

Doppler Velocity Unit (DVL): The effectiveness of implementing the Doppler Velocity Unit has a direct impact on the quality of the Internal Navigation System. Therefore, the fidelity of the information obtained by the DVL must be as accurate as possible to simulate a realistic scenario. When analyzing the performance of this sensor, it's crucial to consider not only the computational technique or methodology behind this marine device, but also factors such as the placement of this object [22] and the incorporation of new technology [23], as they play significant roles.

Given the complexity involved in emulating the process often used by a real DVL sensor, Stonefish does not measure the velocity of the link using the Doppler effect, where the velocity and the distance relative to the sea bottom are computed based on the acoustic waves sent by angled transducers and then the difference of frequency from the received echo is used to calculate these parameters. Instead, Stonefish uses the motion of the simulation to determine the water and AUV velocity along the three axes. Some of the parameters that the user can define include the angle between the beams and the vertical axis *[deg]*, a boolean value specifying if the beams are pointing in the same direction as the z-axis, frequency, and history length.

Pressure: This sensor is utilized to measure underwater pressure. It computes the pressure using the gravity vector and its value over the z-axis, along with the depth where the vehicle is located and the liquid density. Maximum pressure can be defined, as well as the standard deviation.

Profiler: The profiler is a marine device used to measure the distance between the AUV and potential nearby obstacles. It accomplishes this task using a rotating beam profiler, where parameters such as direction, minimum range, and maximum range vector can be specified, along with the standard deviation, field of view *[deg]*, number of rotation steps, frequency, and history length.

2.1.3 Sensors Attached to Joints

Opposite to the link sensors that are attached to the body of Autonomous Underwater Vehicles, these sensors are fixed to the joints of the robot. They all share some basic properties, including a unique name, sensor update frequency *[Hz]*, history length, and the name of the joint to which they are attached.

Rotatory Encoder: While the calculation of the rotation angle of the specified joint may appear straightforward, the algorithm implemented by the Stonefish Library aims to be as realistic as possible. Therefore, the resulting rotation angle depends on various factors, including whether the specified joint is fixed, spring-based, revolute, spherical, prismatic, or cylindrical, and whether it is part of the motor or the thruster. Once the joint type is determined by the class, the rotatory encoder displays a normalized value in the range [-π, π], where the hinge joint angle [24] is obtained from the hinge constraint between two rigid bodies, each with a pivot point providing information about the axis location in local space.

Torque (1-axis): To compute the torque exerted on a specific joint, the joint must be fixed to the motor so the class can calculate the difference in force. The measurement range can be optionally modified, as well as the standard deviation of the measured torque.

Force-torque (6-axis): This sensor is attached to the joint but measures the forces acting on its child link. Its implementation slightly differs from other joint sensors because it requires defining a pointer to the solid entity that will serve as a reference and specifying the origin of the sensor in the body frame. This library class computes the torque and force in all three directions, taking a Cartesian frame as reference. The output of this sensor consists of 6 channels.

2.1.4 Physics-Based Model

Given that the Bullet Physics Library is designed to simulate body collisions, static dynamics, and kinematic bodies, intrinsic properties of the ocean are not simulated using functions from the library but rather by classes created by the author. Hydrodynamic forces acting on the robot are calculated using the geometry of the bodies to create a more realistic effect.

Buoyancy: To introduce the concept of buoyancy, it is essential to begin with the Archimedes principle. This principle states that any object lacking compressive properties will displace a volume of liquid equal to its own volume when is partially or fully submerged in a fluid. According to the Archimedes principle, buoyancy can be defined as the upward force exerted by a fluid, directed opposite to the force of gravity. This buoyant force (Eq. 5) can be calculated with the gravity force g, fluid density ρ and the volume V of the object.

$$F_B = \rho V g \tag{1}$$

When creating a solid entity in Stonefish, it is crucial to specify the shape along with the chosen physics setting mode. Stonefish offers five options for physics settings, but for the purpose of comparing maritime vehicles, only three are relevant:

1. *DISABLED: This setting involves no computation of physics, resulting in zero mass and inertia.*
2. *FLOATING: An advanced simulation tool developed for marine robotics"*
3. *SUBMERGED: This mode incorporates hydrodynamics with buoyancy and added mass.*

Added Mass: The added mass is a crucial factor when studying and evaluating the motion of bodies in an underwater environment. Its implementation is critical in recreating maritime scenarios accurately. This motion effect can be understood as the additional torque generated by the fluid in response to the vehicle's acceleration changes. In simpler terms, as the AUV accelerates, it effectively becomes heavier due to the need to displace the surrounding liquid along with the robot.

It is typically represented in a 6x6 matrix (Eq. 2 and Eq. 3), but due to some limitations in the properties available in the Physics Bullet Library, the added mass matrix is displayed using mean values for all axes and a vector of three inertia moments.

$$F_{AM} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} & m_{15} & m_{16} \\ m_{21} & m_{22} & m_{23} & m_{24} & m_{25} & m_{26} \\ m_{31} & m_{32} & m_{33} & m_{34} & m_{35} & m_{36} \\ m_{41} & m_{42} & m_{43} & m_{44} & m_{45} & m_{46} \\ m_{51} & m_{52} & m_{53} & m_{54} & m_{55} & m_{56} \\ m_{61} & m_{62} & m_{63} & m_{64} & m_{65} & m_{66} \end{bmatrix} \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \\ \dot{u}_4 \\ \dot{u}_5 \\ \dot{u}_6 \end{bmatrix} \tag{2}$$

$$\begin{aligned} &F_{AM} = F_i, \quad where \\ &i = 1,2,3 \quad Linear\ force \\ &i = 4,5,6 \quad Inertia\ Moments \end{aligned} \tag{3}$$

Drag: This phenomena is not fully implemented in Stonefish because simulating these forces, which usually act on each face of the body surface and oppose the vehicle's motion, is difficult for real-time 3D rendering. Therefore, a rough approximation is used, considering the fluid velocity as if there were no body. However, this approach might not provide fully accurate results for a realistic implementation of an underwater simulator.

2.2 HoloOcean

With a similar motivation than Stonefish, the development of HoloOcean [8] aims to address the challenges often encountered in creating realistic simulation environments. HoloOcean is an underwater simulator built on Unreal Engine 4 (UD4) [33], a powerful 3D computer graphics game engine widely known for its high rendering capabilities. Leveraging an external graphics engine such as UE4 allows the integration of community-made environments, offering numerous advantages for testing underwater vehicles.

HoloOcean is an open-source simulator based on HoloDeck [34]. Its primary characteristic, which makes it very attractive for the development of algorithms for autonomous marine vehicles, is the implementation of an interface designed to be familiar to OpenAI Gym [35]. This interface utilizes the "agent-environment loop" (Fig. 2) concept, where classes such as state, reward, terminal and info are called. Additionally, HoloDeck supports multi-agent environments, where the implementation and access of reward, terminal and location classes is also possible.

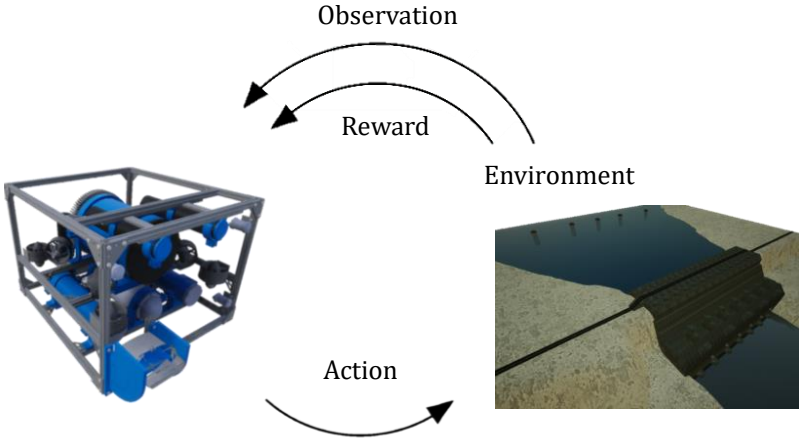


Figure 2: Example of the "Agent-environment loop", where a reward is given if the action performed by the robot in the environment is considered successful.

[8] *This high-fidelity simulator is supported and continuously maintained by The Field Robotic Systems Lab (FRoSt Lab) at Brigham Young University.*

The physics-based model implemented by HoloOcean relies on the PhysX Nvidia Physics engine [36]. This engine offers a comprehensive variety of physics calculations, ranging from fluid and soft body dynamics simulations to collision detection, momentum conservation and the computation of forces applied to rigid bodies. The realism provided by PhysX is at the forefront of rendering techniques, offering excellent visualization of vision sensors and an almost exact representation of real underwater scenarios.

HoloOcean offers three open water complex testing scenarios with static objects and wide dimensions for comprehensive simulations. Scenario files are written in .json format and can be found in packages under the "world" directory[9]. Following the same principle, sensor objects are also written in a .json file, where the structure of the file depends on the characteristics of the sensor and the underwater vehicle used by the user.

HoloOcean was selected alongside Stonefish for a comprehensive comparative analysis due to its sonar image rendering using octrees and its high-quality visualization. Opposite to Stonefish, the use of an external graphics engine such as Unreal Engine provides a different perspective, where computational power is not a primary concern and the focus is on creating the most realistic scenario possible. The rendering technique, combined with the Python interface, the use of different physics-based models and the overall methodology, allows a deeper analysis since both simulation environments may appear quite contrasting in general terms.

2.2.1 Sonar and Camera Systems

The method implemented by HoloOcean to simulate sonar [40] and camera systems is innovative compared to other simulation environments, making this simulator stand out. The sonar imagery approach is based on representing the environment using octrees [38], [39]. To understand the technique used to display sonar images, it's necessary to introduce the concept of octrees and how these data structures help represent environmental information.

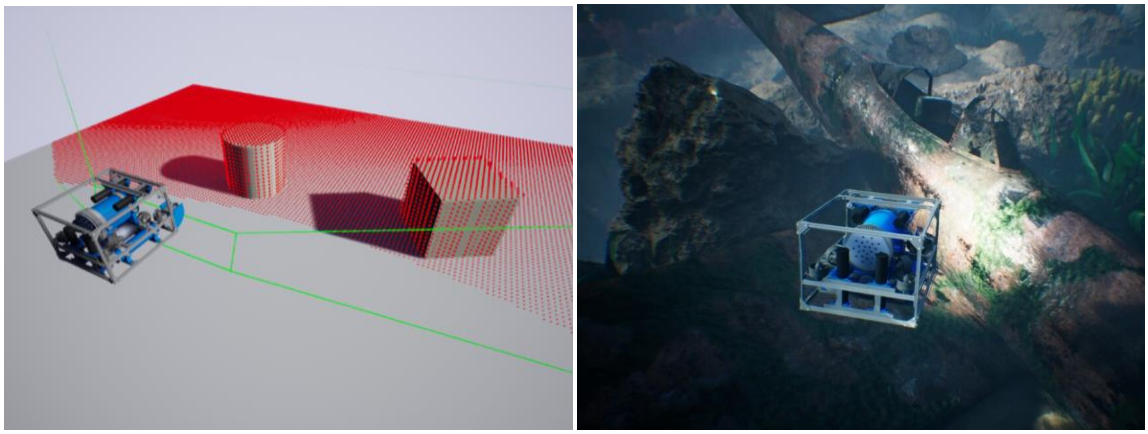
Octrees are tree data structures containing information about the three-dimensional domain. Each node is split into 8 equal-sized squares, known as children or leaf nodes. These squares can be recursively split into 8 more squares to provide more detailed information about the environment. HoloOcean utilizes octrees to access the location and surface normal vectors of objects placed in the scenario. During the creation of these data structures, only squares that are fully or partially occupied are added to the rendering image.

For a complete implementation of this technique, agents that might appear in the sonar's field of view have a small octree that is constantly updated based on the time step. Once the leaves containing essential information for the sonar rendering image are identified, the surface unit normal and the impact unit normal are used to calculate the dot product. Leaves with a dot product equal to or less than zero are not stored or considered, as the angle between leaves is greater than $\frac{\pi}{2}$. Therefore, one leaf is located on the backside of the object and is unnecessary for the 2D image.

Now that the octrees are defined, the beams created by the acoustic waves of a multi-beam sonar sensor, and their reflections caused by hitting an object, help correctly place the octrees in the 2D image. This is possible because each beam has a horizontal angle and a vertical width corresponding to a specific column of the display image.

Once that the leaves containing information about the front side of the static objects or dynamic agents are identified, those lying in shadows should be removed to avoid inefficiency in computational power and memory usage. This process can be accomplished by iterating through the already sorted leaves and computing the difference between neighboring elements. If the result of this calculation is greater than a predefined value, the leaves and the information they contain must be removed. The implementation of the Octrees in the simulation environment to create a realistic sonar image can be visualized in Fig. 3.

[9] *The following format is used to call the scenario: WorldName-ScenarioName.json*



(a) (b)

Figure 3: a) Green lines visible in the image depict the field of view of the agent, while the octree leaves are represented in red. This process is utilized for sonar rendering images. b) The RGB image resolution is shown, demonstrating the high-quality visualization techniques implemented by Unreal Engine. *"HoloOcean: An Underwater Robotics Simulator" [1]*

Now that the sonar sensor model used by HoloOcean is defined, knowledge about the configuration of the Octree during the initialization of the simulated world is equally important to achieve high-quality results. The parameters necessary to start implementing the sonar algorithm are ***'env.min'***, ***'env.max'***, ***'octree.min'***, and ***'octree.max'***. The first two are used by the program to determine the upper and lower bounds of the environment. These properties should be set in the package structure, which is a .json file where the world name is also specified.

'octree.min' and ***'octree.max'*** define the minimum and mid-level size of the octrees. In other words, these parameters specify how low an octree can go during the recursive split process and the maximum size an octree can reach.

Considering that Unreal Engine is primarily used to provide realistic image visualization by implementing high rendering techniques, HoloOcean might possibly have the best camera systems (Fig. 3) compared to other simulation environments. The resolution of the image displayed by the sensor depends on four channels. Three of these store the RGB information with a range that goes from 0 to 256, with an extra channel for Alpha that controls the opacity of the color generated by the RGB combination. The resolution of the camera can be changed to provide an even better visualization of the scenario, but it might lead to deficient performance of other sensors.

To initialize the camera, the "RGBCamera" sensor object class must be called, setting the basic parameters shared with any other sensors. These parameters include the socket name where the sensor will be placed, its location and rotation, either attached to the agent or at a specific place in the scenario and the sample rate. Additional information to configure includes setting the width and height of the captured image.

2.2.2 Sensors attached to the agent's body

This group of sensors consists of those that can be physically attached to the agent body, such as the DVL, IMU, Camera, Sidescan Sonar, Forward-looking sonar, Multi-beam sonar, Depth Sensor, GPS, and pose sensor. These sensors share specific characteristics like sensor name, location, publish message (where the type of publish has to be specified), sample rate, rotation in degrees, and the socket that determines the sensor's placement within the simulator.

Doppler Velocity Log (DVL): Besides the common characteristics shared by every sensor in this group, the configuration block of the DVL allows the modification of various properties. These modifications include the angle of each acoustic beam sent by the sensor, considering the z-axis

that points downward; the default value for this property is 22.5 degrees. Another configurable property is the maximum range that can be returned by the beams. One of the modifications that may have the most significant impact on the sensor output is the boolean value assigned to "ReturnRange", as this determines whether the range of beams should also be returned. This can reduce the output array from 7 scalar values (***velocity.x, velocity.y, velocity.z, range.x.forw, range.y.forw, range.x.back, range.y.back***) to only 3, meaning only the velocity is calculated.

The method implemented by HoloOcean to calculate the sensor velocity according to the sensor frame is based on the Doppler effect. The mathematical notation used to explain the calculations performed by the sensor [1] is shown in Eq.4 and Eq.5, where α denotes the predefined acoustic beam angle, v_b denotes the velocity for each beam as a four-dimensional vector, G^n takes the value of the Gaussian noise, R_s^b is the transformation from the beam frame to sensor frame and u denotes the output displayed by the sensor.

$$u^v = R_b^s(v_b + G^v), G^v \sim (0, \sum^v) \tag{4}$$

$$R_b^s = \begin{bmatrix} \frac{1}{2sin(\alpha)} & 0 & \frac{-1}{2sin(\alpha)} & 0 \\ 0 & \frac{1}{2sin(\alpha)} & 0 & \frac{-1}{2sin(\alpha)} \\ \frac{1}{4cos(\alpha)} & \frac{1}{4cos(\alpha)} & \frac{1}{4cos(\alpha)} & \frac{1}{4cos(\alpha)} \end{bmatrix} \tag{5}$$

Inertial Measurement Unit (IMU):The configuration box of this sensor does not allow any modification of the sensor itself, but it permits the user to set specific covariances for the acceleration and the angular velocity. It is also possible to define a covariance for the acceleration bias component and the angular velocity bias component. Additionally, a boolean expression specifies whether the sensor should return the bias of the angular velocity and acceleration. The output displayed by the sensor is a 2D array. Depending on the boolean expression set for the return bias option, it will create either a 3 x 2 matrix or a 3 x 4 matrix.

Depth Sensor: It might be the simplest sensor in HoloOcean since it returns a single scalar value after the computing process ends, which is directly proportional to the depth. Given the nature of the sensor, it provides the z-axis position. Only the covariance can be modified in the configuration block.

Global Positioning System (GPS): The GPS can be attached to the agent body, but it might not be as attractive for the simulation of underwater robots since it is only useful when the AUV is close enough to the surface to receive localization measurements. Therefore, it is not necessary to delve deeply into the implementation of this sensor.

Pose Sensor: The pose sensor is often used in HoloOcean to access the navigation ground truth. Its output is a 2-dimensional array where the rotation matrix of the sensor frame and the robot's world location are calculated. The configuration box does not allow any modification since it is implemented to provide the ground truth. The mathematical model to express this result can be observed in Eq. 6 and Eq. 7, where $R_s^r t_s^r$ detonates the transformation between the sensor frame and the robot frame and $R_r^w R_s^r$ represents the transformation to the world frame.

$$z^p = \begin{bmatrix} R_s^w & t_s^w \\ 0_{1x3} & 1 \end{bmatrix} \tag{6}$$

$$\begin{aligned} R_s^w &= R_r^w R_s^r \\ t_s^w &= R_r^w t_s^r + t_r^w \end{aligned} \tag{7}$$

2.2.3 Sensors not attached to the agent's body

The following group of sensors are not physically attached to the agent's body, but they can be initialized at the moment of creating the scenario to retrieve information about the current state of the Autonomous Underwater Vehicle or the simulation itself. Since these sensors do not try to mimic real components, there is no specific computational technique required to obtain the output. Therefore, the configuration block is very simple for all of them and the opportunity to modify the final result is almost nonexistent.

The principal sensors that form this group are: *LocationSensor, MagnetometerSensor, RotationSensor, VelocitySensor, WorldNumSensor*, and *DynamicsSensor*.

2.2.4 Physics-Based Model

Considering that HoloOcean incorporates the Nvidia PhysX engine, which is supported and constantly improved by Unreal Engine 4, the simulation of water physics using an external graphics engine might seem as one of the strengths of this simulation environment. Some of the properties emulated when creating an underwater scenario include buoyancy, which is calculated based on the 3D model that is in contact with the water and the percentage in terms of volume that is submerged into the liquid. HoloOcean mimics this property with the implementation of a function called ***'ApplyBuoyantForce()'***, which can be found in the file ***'HolodeckBuoyantAgent.cpp'***. Additionally, properties such as gravity, collision of rigid bodies and their response based on other object types, as well as other hydrodynamics forces, are also implemented.

Viscous Drag: Given that viscosity is a natural property of liquids, every object fully or partially submerged into the liquid will experience viscosity drag. This phenomenon can be explained as the resistance of the fluid to the motion of the vehicle and is calculated with the following equation, where ρ = Density of the fluid, v = Velocity of the object, C_D = Drag coefficient and A = cross sectional Area

$$F^D = \frac{1}{2}\rho v^2 C_D A \tag{8}$$

Since HoloOcean utilizes the predefined formulas and functions provided by Unreal Engine, viscous drag is computed using linear damping, which is an essential property of the Physics Bodies and Physics Constraints. Linear damping within the engine is employed to resist the translation of the object.

Fluid Friction: Closely related to the previously explained viscous drag, fluid friction may seem similar since both forces oppose the motion of a vehicle submerged in fluid. However, in HoloOcean, fluid friction is calculated using angular damping instead of linear damping. In other words, friction in this context refers to the resistance encountered when an underwater vehicle rotates along any of its axes.

Inertia Tensor: It is generated automatically from the 3D model. It is calculated based on the moment of inertia along the three axes of the vehicle and the angular acceleration needed to rotate the vehicle inside the simulation. Together with fluid friction and viscous drag, it is a force that needs to be considered to generate smoother movement of the vehicle within the scenario.

$$\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix} \tag{9}$$

3 Description of the Investigation

Given that the primary objective of this research is to compare two main simulation environments, focusing on autonomous underwater vehicles, the key properties of each simulator are the data collected by the sensors and the realism of the physics during the tests. As a result, the approach to discern differences between simulation environments relies on a thorough analysis of the sensors output and the methods employed to present this data. Furthermore, rendering quality will also be assessed to facilitate a comprehensive comparison.

Evaluate all the sensors and observe the behavior of the Autonomous Underwater Vehicle in a single simulation could potentially lead to hardware issues, which may cause bias in the sensors measurements, affecting the analysis of the simulation environment. Therefore, the sensors will be attached according to the scenarios where they can perform optimally.

Datasets will be generated after each simulation to facilitate a more detailed analysis. The sensors contributing data to these datasets will be evaluated based on the quality and potential utility of the collected data. This assessment will also consider whether the data is suitable for implementing algorithms or for accurately localizing the robot.

3.1 Resources and Tools

This project is implemented on Ubuntu 22.04 [26] as the main operating system, with ROS2 [6] Iron [27] being continuously utilized to enhance communication between nodes. The programming languages and their versions vary: Stonefish predominantly utilizes C++ 11.4.0, whereas HoloOcean employs ROS Python 3.10.12. An 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz × 8 processor [28] is utilized alongside a Mesa Intel® Xe Graphics (TGL GT2) [29] and NVIDIA Data Center GPU Driver version 535.104.05 [30] throughout the simulation process.

3.2 Stonefish

The development of scenarios in Stonefish was based on the same C++ architecture, maintaining consistent rendering settings such as window size, shadow quality, ocean and atmosphere, all set to the highest quality supported by the library. The Graphical User Interface was initialized using OpenGLPrinter. The Autonomous Underwater Vehicle used in the simulations is the GIRONA500, Fig. 4, equipped with five thrusters and their corresponding servomotors, allowing the vehicle 5 degrees of freedom (DOF).

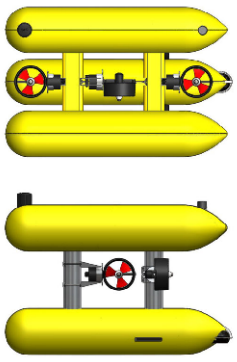


Figure 4: AUV GIRONA 500 with 5 Thrusters, 5 DOF, *"The Girona 500, a multipurpose autonomous underwater vehicle" [31]*

[6] *It is possible to develop simulators in Stonefish using ROS packages, which are compatible with both ROS and ROS2, not limited exclusively to ROS2. Additionally, the XML parser function is available for use.*

3.2.1 Testing 1

During the first test scenario, the environmental characteristics were configured to observe and evaluate the realism of water physics, including hydrodynamics, underwater currents and the interaction of the AUV GIRONA 500 with the ocean surface. Two out of the three water currents were implemented: Jet and Uniform. Uniform creates a velocity field across the entire ocean, while Jet attempts to simulate a rapid stream of water coming from a circular underwater outlet. Four static figures (two spherical shapes and two boxes) were created and placed in the world to test the sonar sensors. Additionally, the AUV was equipped with the forward-looking sonar, Side Scan sonar and Mechanical scanning imaging sonar. It was programmed to move in a straight line and counteract the force applied by the current on the robot. Physical properties such as buoyancy were adjusted. The ThrusterHeaveStern and ThrusterHeaveBow, both located at the top of the robot, were set to 0.550, while the ThrusterSurgePort, located on the left side of the robot, was set to 0.15.

During the testing, acoustic sonar images were displayed on the screen at all times (Fig. 5), providing information about the environment.

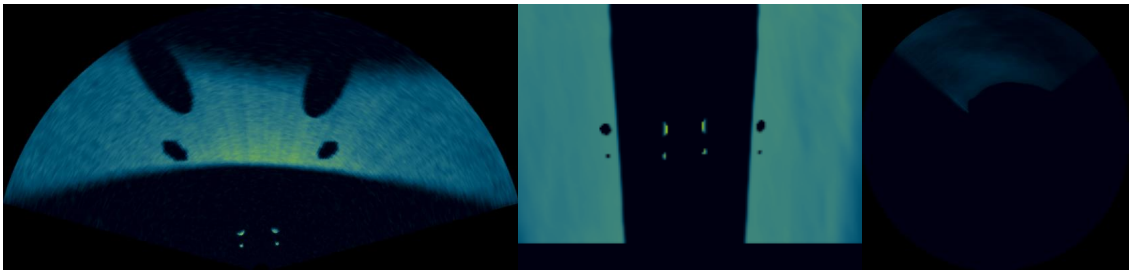


Figure 5: a) Image provided by the Forward-Looking Sonar (FLS), the maximum range was set at 30 meters. The boxes within the image were situated closest to the bottom. b) The output from the Side-Scan Sonar (SSS) shows the boxes inside the image as small dots. c) The Mechanical Scan Sonar did not detect anything as the boxes and spheres were outside its range.

3.2.2 Testing 2

To begin with the data collection process, the next scenario was created with a focus on the potential future implementation of algorithms using the acoustic images displayed by the Forward-Looking Sonar. Given that the sonar updates the displayed image every 0.005 seconds, the complete simulation will generate excessive data and the memory required to store it all is virtually nonexistent. Therefore, the data collection process commences after the first second and concludes after four collections.

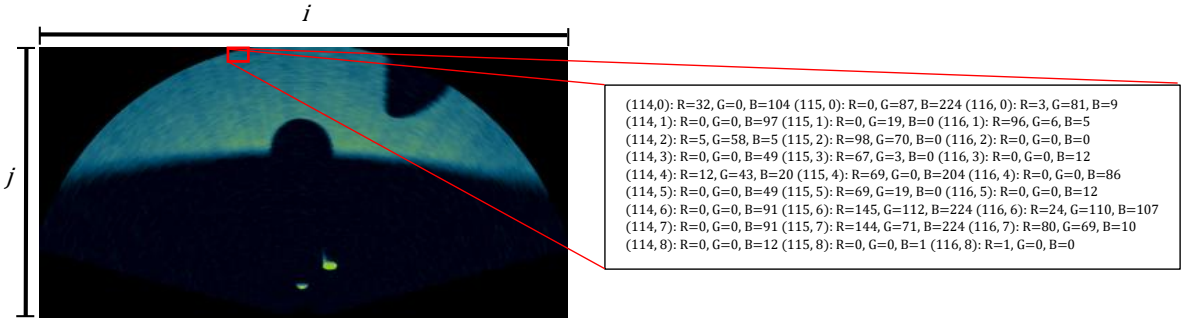


Figure 6: Image taken during the collection. Example of the RGB representation pixel-by-pixel

By employing C++ pointers to access the RGB combination pixel by pixel of the output image (Fig. 6), the gathered information was sent to a .txt file. This file was subsequently converted into a .csv format for easier handling.

3.2.3 Testing 3

The third test focuses on evaluating the scalar sensors rather than the visual sensors. Consequently, sensors crucial for the internal navigation system, such as the Doppler Velocity Log and IMU, will be under testing. The output generated will be displayed on the terminal and saved in a .txt file for a deeper analysis of the sensors' behavior.

Measurements from DVL
Number of channels: *8*
Number of samples: *2283*
Frequency: *200.000 Hz*
Unit system: *SI*

Time	Velocity X	Velocity Y	Velocity Z	Altitude	Water velocity X	Water velocity Y	Water velocity Z
0.00000	-0.00228,	-0.01302,	-0.02106,	9.06376,	0.03801,	0.03250	-0.01051
0.00000	-0.00228	-0.01302	-0.02106	9.06376	0.03801	0.03250	-0.01051
0.00500	-0.03196	-0.03270	-0.00906	9.06500	0.00630	0.01058	0.01578
0.01000	-0.01541	-0.04851	-0.01049	9.06352	0.00153	0.01031	-0.00907
0.01500	0.00858	-0.02366	-0.00162	8.99665	-0.01013	0.01203	0.00921
0.02000	-0.00982	0.01001	0.01451	9.05153	0.05000	-0.00624	0.00658
0.02500	0.02576	-0.00461	-0.00770	9.10499	0.01117	-0.01003	0.01302
0.03000	0.02219	-0.03346	0.01986	9.03233	0.03544	0.01747	-0.02510
0.03500	0.00558	-0.03402	0.00694	9.15195	0.00525	-0.01408	0.02606
0.04000	-0.02762	-0.04262	0.01378	9.13763	-0.00686	0.00010	-0.00359
0.04500	0.01521	-0.04521	0.01874	9.11605	0.02600	-0.00003	0.00256
0.05000	0.04225	-0.02390	-0.00517	9.01242	-0.01435	-0.00035	-0.03212
0.05500	0.04510	0.00161	-0.01921	9.11701	-0.00200	-0.00620	-0.00918
0.06000	0.05000	0.01009	-0.04055	9.07990	0.02250	0.04307	0.01231
0.06500	0.02073	-0.00866	0.00316	9.07021	-0.00942	-0.01227	0.01591
0.07000	0.02242	-0.04046	-0.00336	9.06809	-0.00677	0.02593	0.00803
0.07500	0.03680	-0.00346	0.00981	9.01328	-0.00053	-0.01845	-0.01892
0.08000	0.01429	0.02220	0.01596	9.10013	0.03895	0.02021	-0.00853
0.08500	0.00709	0.00571	0.04804	9.05078	-0.01239	-0.00138	-0.00026
0.09000	0.00268	-0.04449	0.00051	9.05954	0.00049	0.00487	-0.01803

Table 1: Example of the data collected from the DVL: The total number of samples collected during the test is 2283, but the table displays only 22 measurements for simple appreciation.

3.2.4 Testing 4

This is the final and most complex scenario implemented to analyze the Stonefish Library. During this simulation, the AUV GIRONA 500 will be equipped with a color camera, DVL, IMU, Compass, and Odometry. In this last test, the vehicle will autonomously execute various activities, relying on the data computed by the sensors to drive its decision-making processes. Four spheres will be placed in the world, forming a square with a distance of 10 meters between each other. The mission for the GIRONA 500 is to reach all four spheres using a modified version of the carrot-chasing algorithm.

The algorithm employed by the AUV to accomplish the task begins by utilizing the Odometry sensor, which measures and calculates the vehicle's location within the environment. Subsequently, the positions of the four boxes are determined using classes provided by the Stonefish Library.

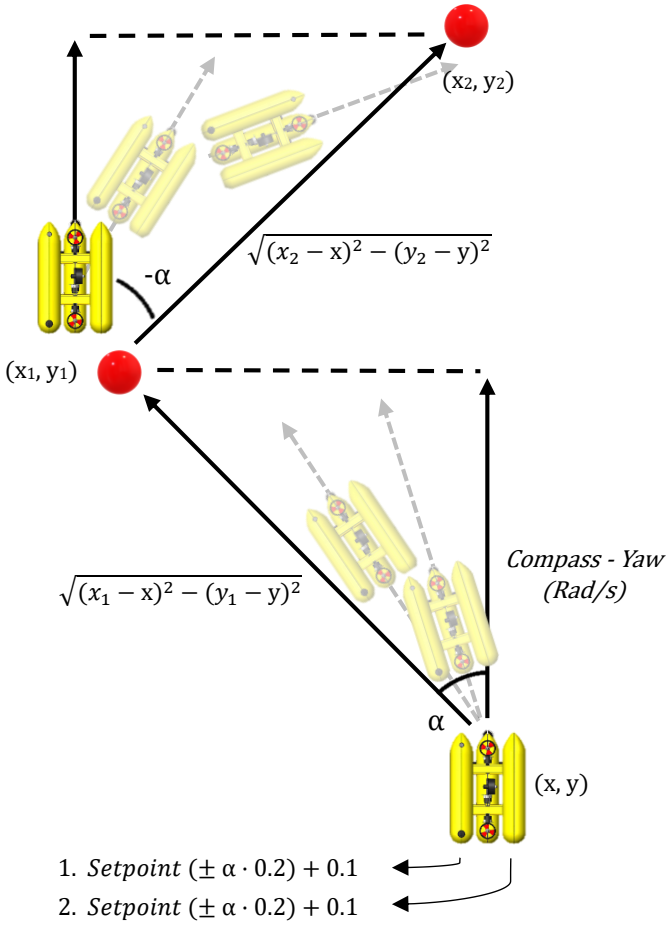


Figure 7: Example of the modified version of the carrot-chasing algorithm

The algorithm then computes the Euclidean distance between the GIRONA 500 and the static objects. Once the distances are determined, ”If statements” are utilized to determine which box is closest to the AUV. The output from the Compass sensor, in combination with the position of the nearest box and the current AUV position, is used to calculate the arctangent between these two points, providing information about the angle. The sign of the calculated value determines the direction in which the thrusters will move. The torque generated by each thruster is determined based on the steepness of the angle. If the angle between the box and the AUV is less than 5 degrees, both thrusters will be set to the same power.

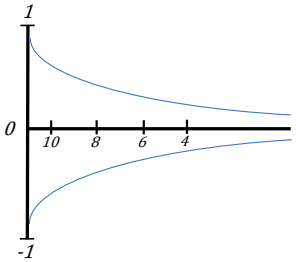


Figure 8: The thrust power decreases as the Euclidean distance gets smaller. The x-axis represents the distance in meters.

Since the GIRONA 500 is moving towards the box previously marked as the closest, the Euclidean distance decreases with each iteration of the algorithm. Consequently, as the distance between

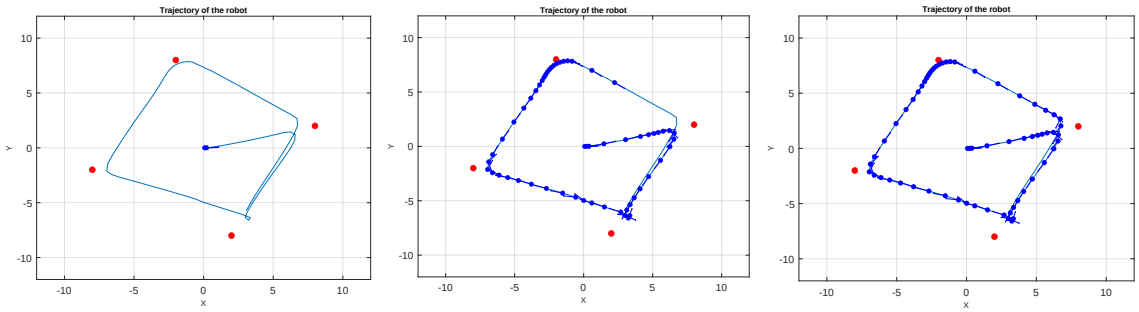
the AUV and the box decreases, the velocity and force generated by the thrusters[7] should also decrease to avoid errors that could lead to the disorientation of the robot. By implementing control theory (Fig. 8), the range of power that the thrusters have is between 1 and -1. Once the AUV reduces the Euclidean distance to the value defined by the user as optimal, it would move to the subsequent box based on the specified order.

Algorithm 1 Path-Following Modified Carrot Chasing

$Odometry \rightarrow AUV.Pos$
$Origin \rightarrow Box^{(1)}.Pos$
...
$Origin \rightarrow Box^{(n)}.Pos$
$X^{(1)} \leftarrow EuclideanDistance(Box^{(1)}.Pos, AUV.Pos)$
...
$X^{(n)} \leftarrow EuclideanDistance(Box^{(n)}.Pos, AUV.Pos)$
if $(X^{(1)} > i)$ **then**
    $\alpha \leftarrow arctan(Box^{(1)}.Pos, AUV.Pos)$
    if $(\alpha > 0)$ **then**
        if $(0.05 < (\alpha - Compass))$ **then**
            $ThrusterS \leftarrow SetPoint(0.1 + (\alpha * 0.3))$
            $ThrusterP \leftarrow SetPoint(0.1 + (-\alpha * 0.3))$
        else if $(-0.05 > (\alpha - Compass))$ **then**
            $ThrusterS \leftarrow SetPoint(0.1 + (-\alpha * 0.3))$
            $ThrusterP \leftarrow SetPoint(0.1 + (\alpha * 0.3))$
        else
            $ThrusterS \leftarrow SetPoint(e^{(X^{(1)} * \lambda)} * (X^{(1)} * \lambda))$ [7]
            $ThrusterP \leftarrow SetPoint(e^{(X^{(1)} * \lambda)} * (X^{(1)} * \lambda))$
        end if
    else if $(\alpha < 0)$ **then**
        if $(-0.05 > (\alpha - Compass))$ **then**
            $ThrusterS \leftarrow SetPoint(0.1 + (\alpha * 0.3))$
            $ThrusterP \leftarrow SetPoint(0.1 + (-\alpha * 0.3))$
        else if $(0.05 < (\alpha - Compass))$ **then**
            $ThrusterS \leftarrow SetPoint(0.1 + (-\alpha * 0.3))$
            $ThrusterP \leftarrow SetPoint(0.1 + (\alpha * 0.3))$
        else
            $ThrusterS \leftarrow SetPoint(e^{(X^{(1)} * \lambda)} * (X^{(1)} * \lambda))$
            $ThrusterP \leftarrow SetPoint(e^{(X^{(1)} * \lambda)} * (X^{(1)} * \lambda))$
        end if
    end if
else if $(X^{(n)}) > i$ **and** $(X^{(1)}) \rightarrow reached$ **then**
    ...
end if

To visualize the entire path followed by the AUV throughout the simulation, its position data was recorded in a .txt file. Using Matlab 2023b [32], a graph (Fig. 9) was generated with this information. The spheres are represented by red dots on the graph. The GIRONA 500 position is plotted every 300 measurements taken by the Odometry to avoid excessive data clutter and observe changes in the AUV's state. Its direction is represented by a straight line indicating its heading at that moment.

[7] *The variable λ is defined based on user preference. Its definition directly affects the power of the thrusters during the AUV movement around the world. The larger λ is, the more power the thrusters will have. This is crucial, as precise adjustment of the thruster force is necessary to prevent any malfunction of the algorithm.*



(a) (b) (c)

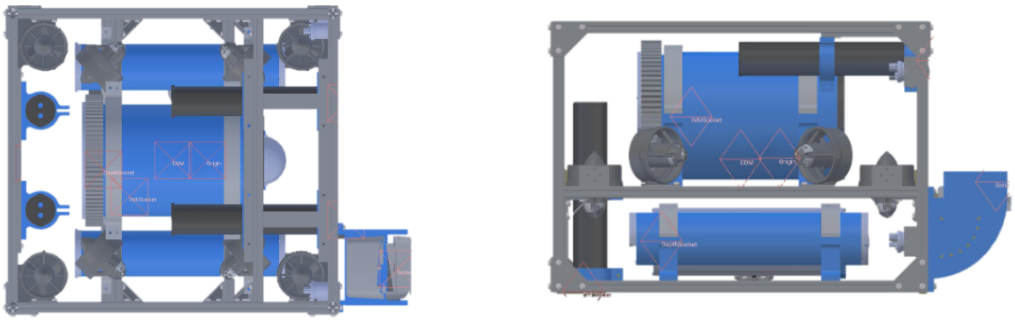
Figure 9: a) The lighter-toned line represents the path created by the AUV using the modified version of the carrot chasing algorithm. b) The dots in a stronger blue tone represent the current AUV position, with the line indicating the direction. c) Final image representation.

3.3 HoloOcean

During the first part of the development of scenarios in HoloOcean, the packages available for download that do not require any additional source or specific modification of the files were mainly used to test the different configurations offered by the open-source library. Therefore, the general simulation structure stayed within the possibilities that the ocean and agent packages provide to the user.

The second part in the development of underwater scenarios involved creating a Unreal Engine project based on HoloDeck. After building and compiling the world, the next step was to attach the Python file process with the HoloOcean Library and HoveringAUV sensor specifications. This process provides the freedom to create underwater worlds where water physics properties and the entire environment can be modified according to the project's objectives. Additionally, leveraging pre-existing community-made underwater worlds is also possible. The Unreal Engine version used to avoid package mismatch and configuration problems was UE4 4.27.

The autonomous Underwater Vehicle employed for the simulations is the HoveringAUV (Fig. 10), designed specifically to perform activities inside HoloOcean [1]. The structure of this vehicle is based on the BlueROV2 Heavy, created by the company Blue Robotics Inc [43][44]. This AUV is equipped with 8 thrusters and their corresponding motors, allowing the vehicle 6 degrees of freedom.

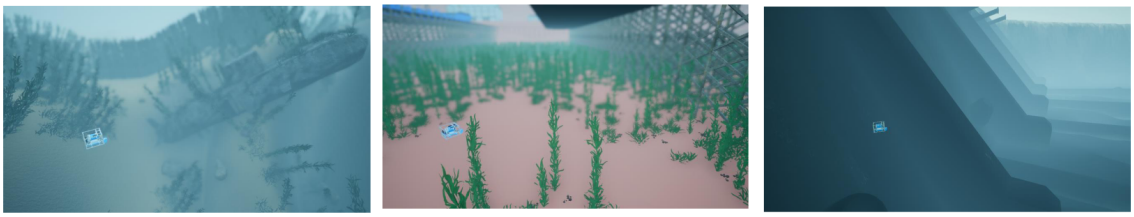


(a) (b)

Figure 10: a) Visualization of the HoveringAUV from the top. b) Visualization of the HoveringAUV from the side *HoloOcean: An Underwater Robotics Simulator [1]*

HoloOcean incorporates five worlds within its open-source library, each created with very specific characteristics to cover every possible scenario that might arise during the implementation of

autonomous underwater vehicles. Furthermore, extensive sensor testing across these scenarios demands considerable time, limiting the capacity for in-depth analysis of collected data. Therefore, only ***'Dam'***, ***'Open Water'***, and ***'Pier Harbor'*** were considered for conducting the corresponding HoveringAUV tests.



(a) (b) (c)

Figure 11: a) Visualization of the "Open Water" world. b) "Pier Harbor" world. c) "Dam" world by *"HoloOcean: An Underwater Robotics Simulator" [1]*

3.3.1 Testing 1

The first testing scenario involves ***'Open Water'*** as the main world, where the HoveringAUV was introduced to observe and evaluate the realism of the water physics. Since the available packages for download contain assets that are property of Unreal Engine and are subject to author rights, changing the environment without access to these assets is not possible. Therefore, customizing and modifying the scenario according to the needs of the project becomes a non-viable task.

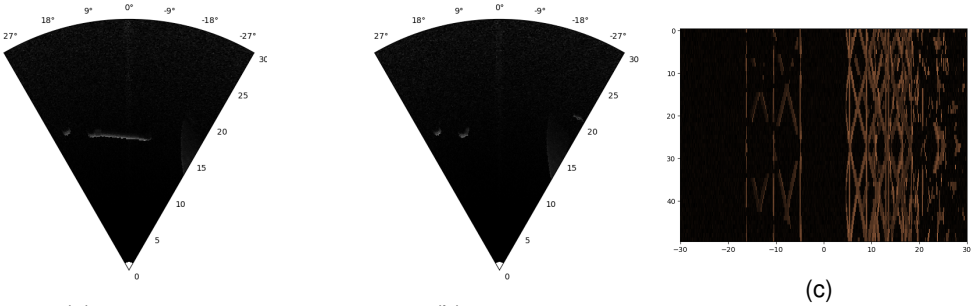
Additionally, a forward-looking sonar, together with a sidescan sonar [8], was attached to the HoveringAUV to recreate acoustic images. Similar to the first testing scenario for Stonefish, four static shapes were created and placed in the world to observe the quality of the image simulated by the sonar sensors. Introducing external shapes into the underwater environment inevitably alters the scenario to some extent, thus the file containing information about the octrees generated during the initialization of this world should be updated to reflect the new shapes. Consequently, the cache file, automatically created by the editor to store scenario information, must be deleted. This ensures that the octrees are regenerated, taking into account the presence of the external shapes. Once this procedure is complete, the shapes placed in the scenario should have been visible in the acoustic images simulated by the forward-looking sonar. However, this was not the case, suggesting that the implementation of external objects in HoloOcean is purely for collision interaction or object detection using other sensors, e.g., RGB camera.

Given that the intention of this testing scenario includes the evaluation of sonar sensors, the Hovering AUV had to be placed in a different location where the sonar could detect objects originally included within the world. Therefore, the new spawn location of the AUV is close to a wall with additional figures.

The employment of the ***'Open Water'*** world to evaluate the image quality of the Side Scan Sonar is insufficient for a comprehensive evaluation. Since the shapes and figures found in the original world are very simple and do not create a challenging environment for this sensor, the ***'Pier Harboring'*** environment was employed to collect additional images due to its complex structure with multiple columns and rows.

Implementing the sonar requires significant computational time for the computer to create the octrees. To expedite this process, the sonar properties were adjusted to a lower value than the one predefined by the documentation, which is recommended for optimal visualization of the acoustic image.

[8] *The integration of the Side Scan Sonar demands substantial computational power, as the resulting image typically consists of 2000 range bins by default (compared to 512 for the forward-looking sonar). Additionally, the Azimuth and Elevation range of the Side Scan Sonar are often larger than the range of other sonar sensors, further increasing the computational power required.*



(a) (b) (c)

Figure 12: a) Acoustic image recreated by the Forward-looking sonar in the 'Open Water' world. b) Image rendering by the Forward-looking sonar in a different location inside the world. c) Side Scan Sonar image where the structure of the Harbor is captured.

To ensure a better implementation of the Side Scan Sonar within the simulation, it is essential to use Unreal Engine 4 to edit the environment properties and subsequently attach the simulation process. Adjusting both scenario characteristics and assets is necessary to mitigate high computational processes and achieve a smooth recreation of the image. Otherwise, relying on the predefined packages available for download often leads to rendering issues, resulting in editor crashes or prolonged waiting times for octree generation.

Given that this procedure is part of the second phase of scenario development in HoloOcean, evaluating the implementation of the Side Scan Sonar solely using the HoloOcean packages and tools available in the open-source library is conducted during this initial testing scenario.

The images collected through the testing were saved into a .txt file using the numpy.ravel function, which captures the pixel-by-pixel RGB combination in a 1-D array. Here, 1.0 represents full color (white), while 0.0 signifies non-color (black). An example of the format used to store the image numerically can be seen in Table 2.

Measurements from Imaging Sonar Number of samples: *17* Frequency: *1.000 Hz*							
N. Forward Looking Sonar Image	[R,	G,	B]	...	[R,	G,	B]
Imaging Sonar [1]:	0.0000,	0.0000,	0.0000,	...	0.08214,	0.05716	0.02972
Imaging Sonar [2]:	0.0000	0.0000	0.0000	...	0.05491	0.05732	0.14233
Imaging Sonar [3]:	0.0000	0.0000	0.0000	...	0.10646	0.05853	0.03801
Imaging Sonar [4]:	0.0000	0.0000	0.0000	...	0.03331	0.05833	0.07279
Imaging Sonar [5]:	0.0000	0.0000	0.0000	...	0.02263	0.06000	0.08101
Imaging Sonar [6]:	0.0000	0.0000	0.0000	...	0.09486	0.05414	0.03829
Imaging Sonar [7]:	0.0000	0.0000	0.0000	...	0.11497	0.05044	0.05041
Imaging Sonar [8]:	0.0000	0.0000	0.0000	...	0.08128	0.05426	0.08606
Imaging Sonar [9]:	0.0000	0.0000	0.0000	...	0.06892	0.03017	0.01883
Imaging Sonar [10]:	0.0000	0.0000	0.0000	...	0.07537	0.13007	0.02713
Imaging Sonar [11]:	0.0000	0.0000	0.0000	...	0.06520	0.10508	0.03801
Imaging Sonar [...]:	...	...	...	...	...	...	...
Imaging Sonar [17]:	0.0000	0.0000	0.0000	...	0.11008	0.10782	0.01546

Table 2: Example of the numerical representation of the acoustic image created by the forward-looking sonar. Since it's a 1-D array, the first three digits correspond to the first pixel and this pattern continues throughout the array.

3.3.2 Testing 2

In order to start data collection, the second testing scenario was created, considering sensors that can be attached to the agent's body, such as DVL, IMU, and Pose sensors. The world used to observe the performance and quality of the output data was the ***'Dam'***. Measurements from the sensors were displayed in the terminal, as well as saved in .txt files for deeper analysis.

Measurements from DVL
Number of channels: *7*
Number of samples: *3716*
Frequency: *200.000 Hz*
Unit system: *SI*

Time	Velocity X	Velocity Y	Velocity Z	Ran Beams X FW	Ran. B. Y FW	Ran. B. X BW	R. B. Y BW
0.00000	3.077e-02,	-1.861e-02 ,	1.240e+00,	5.012e+01,	5.004e+01,	5.000e+01	4.991e+01
0.00500	9.699e-03	2.068e-02	1.501e+00	5.009e+01	4.995e+01	5.004e+01	5.013e+01
0.01000	-9.700e-02	7.825e-03	1.782e+00	5.001e+01	5.013e+01	4.985e+01	4.994e+01
0.01500	-4.409e-02	-9.235e-03	2.034e+00	4.982e+01	5.020e+01	4.997e+01	4.984e+01
0.02000	2.942e-02	4.891e-02	2.263e+00	5.005e+01	4.987e+01	5.019e+01	4.989e+01
0.02500	7.889e-02	1.198e-01	2.543e+00	4.988e+01	4.996e+01	5.006e+01	5.004e+01
0.03000	5.091e-02	9.613e-02	2.743e+00	5.000e+01	5.013e+01	4.977e+01	5.013e+01
0.03500	-6.472e-03	-1.117e-02	2.975e+00	4.998e+01	4.998e+01	5.014e+01	5.006e+01
0.04000	-6.322e-02	4.894e-02	3.207e+00	4.987e+01	5.022e+01	5.001e+01	4.993e+01
0.04500	1.587e-02	-4.163e-02	3.432e+00	5.010e+01	4.997e+01	4.991e+01	4.998e+01
0.05000	-2.290e-03	-1.406e-02	3.624e+00	4.998e+01	5.010e+01	5.015e+01	4.999e+01
0.05500	8.656e-04	8.109e-02	3.823e+00	4.993e+01	5.000e+01	5.005e+01	4.998e+01
0.06000	9.330e-02	-6.020e-02	4.034e+00	5.020e+01	5.006e+01	4.957e+01	5.010e+01
0.06500	-1.662e-02	-3.665e-02	4.211e+00	5.000e+01	5.005e+01	4.998e+01	5.003e+01
0.07000	-1.624e-03	2.152e-02	4.394e+00	5.009e+01	5.009e+01	5.002e+01	5.001e+01

Table 3: Example of the data collected from the DVL: The total number of samples collected during the test is 3716, but the table displays only 15 measurements for simple appreciation. The output displayed by the sensor is expressed as a 1-D numpy array.

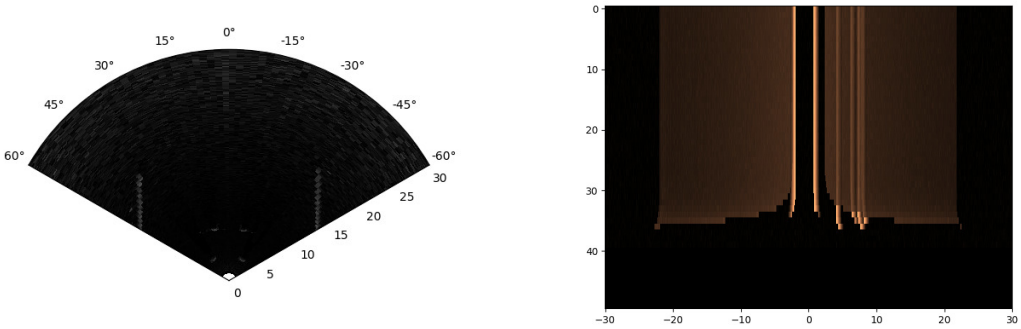
3.3.3 Testing 3

The second phase of the development of simulation environments in HoloOcean begins with the creation of this scenario. To establish an environment where the HoveringAUV can perform the desired activities, it was necessary to implement an Unreal Engine project with characteristics that enable the generation of octrees in conjunction with Holodeck. The setup of the environment consists of two static spheres, two static boxes and three walls, simulating the sea bottom and possible obstacles that can be found within it.

The HoveringAUV will be equipped with forward-looking sonar and side-scan sonar so it can obtain information about the environment. As mentioned before during **'Testing 1'**, every time the world is modified to some extent, the Octree file of the engine must be deleted so the file can be created again, considering all the modifications made to the simulation environment.

By using the Unreal Engine 4 editor directly, the water physics and environment characteristics in general can be modified as they are more convenient for the project's development. Even though this is possible, the environment created for this testing scenario does not have any underwater characteristics, since this would take time and is not directly related to HoloOcean anymore, but to the external graphics engine and the knowledge about it. In other words, the analysis of this world is more focused on HoloDeck, which is the Python library used by HoloOcean to recreate the sensors and vehicle characteristics.

The parameters set for the simulation of the forward-looking and side-scan sonar images are considerably higher than the values used during **'Testing 1'**, given that the world is smaller and does not contain additional assets. The acoustic images collected during the testing can be seen in Fig. 13.

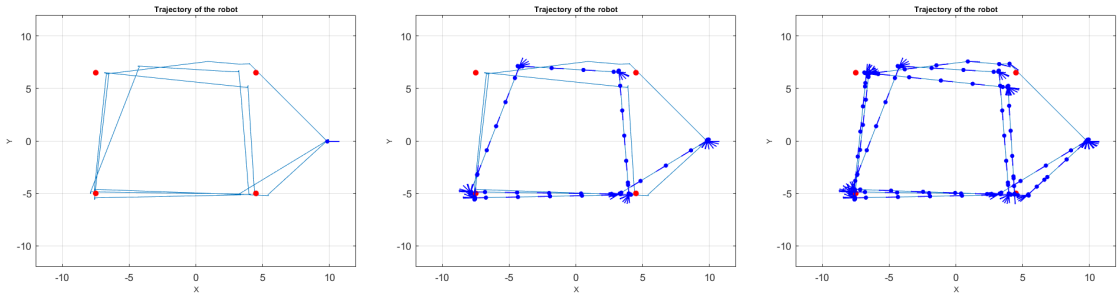


(a) (b)

Figure 13: a) Acoustic image created by the FLS, the maximum range was set to 30 meters. The characteristics of the objects in terms of volume were adjusted to replicate the scenario implemented in Stonefish. b) Image displayed by the Side Scan Sonar. It can be seen that the spheres are not well represented, suggesting that the sonar update is not functioning correctly.

3.3.4 Testing 4

This world is created to evaluate the behavior of the robot in semi-autonomous activities. In this scenario, the Hovering AUV will execute commands provided by the user via keyboard input, along with following predefined paths. Similar to the testing scenario designed for evaluating the GIRONA 500, the environment in Unreal Engine consists of four spheres arranged in a square of approximately 10 meters of length for each side. The objective for the Hovering AUV is to reach all four spheres while recording its position and other sensor measurements into a .txt file. This data will later be utilized to plot the robot's position and the path followed using Matlab 2023b (Fig. 14).



(a) (b) (c)

Figure 14: a) Using the same structure and properties as the graphs presented for Stonefish, the line represents the path created by the AUV, while the red dots represent the spheres. b) Dots in a stronger blue indicate the current position of the vehicle. c) Image illustrating the full path.

4 Evaluation of the Investigation

In order to correctly evaluate the data collected throughout the various underwater scenarios previously created in both simulation environments, it is crucial to introduce and discuss the criteria used to evaluate the performance of the autonomous underwater vehicles. Although the implementation of sensors in Stonefish and HoloOcean may vary due to the intrinsic nature of each simulator, every simulation environment share properties when developing autonomous underwater robots and their corresponding sensors. Some common properties include simulation time, sensor update frequency and unit system (SI). Vision sensors such as sonars and cameras also share other specifications like resolution, minimum and maximum range. Therefore, sensors with similar properties will be compared deeply in terms of quality and frequency of the computed data. In cases where the sensor offers additional output or wider configuration settings, this will be mentioned, and advantages or disadvantages of having this extra configuration will be provided.

Stonefish, as a comprehensive simulation environment for marine vehicles, will be evaluated in terms of sensor effectiveness, ease of implementation and quality output. Sensors implemented exclusively within its open-source library will be considered, emphasizing their importance and impact in algorithm development and potential contributions to the state-of-the-art of underwater technology. Similarly, HoloOcean and its features, which could offer solutions to present simulation challenges, will also be discussed.

4.1 Comparative Analysis of Sonar Sensors

Addressing one of the primary challenges in underwater tasks, the issue of limited vision for autonomous underwater vehicles appears to be solved through the integration of high-quality sonar sensors and accurate decision-making algorithms. Even though the current rendering techniques used to generate acoustic images are highly sophisticated, the creation of datasets that assist in algorithm development remains limited.

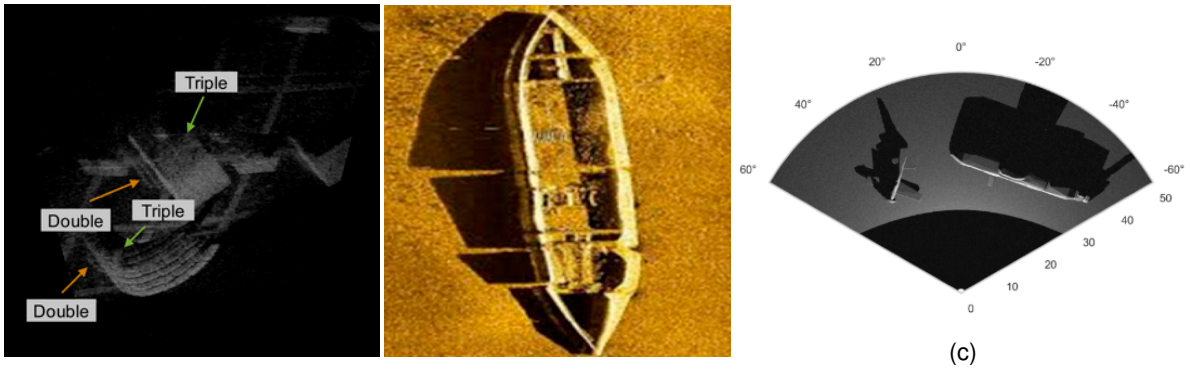
The images collected using the forward-looking sonar available in the Stonefish library are ideal for initiating a basic object recognition algorithm, given their clarity and high quality. However, the development of higher-level object recognition algorithms capable of handling noisy and distorted images typically captured by real sonars may not be feasible, as common problems often encountered by real sonar sensors are not fully replicated by this simulator.

Some undesired situations commonly encountered when using sonar sensors in field practices, such as echo, particularly ground echo (Fig. 13, *(a)*) and noisy images, are not fully simulated by the Stonefish library. Consequently, training object detection algorithms with the dataset collected by the forward-looking sonar might not yield the expected results when implemented in a real environment. This discrepancy could potentially lead to behavioral mistakes when autonomous underwater vehicles are close to the sea bottom or any other surface that may create a multiple bounce effect of the sonar beams.

HoloOcean encounters similar challenges in simulating ground echo effects but may offer a more accurate representation of noisy images, thereby providing a more realistic depiction of acoustic images compared to Stonefish. Additionally, the sonar rendering technique based on octrees provides a shadowing effect (Fig. 13, *(b)*), which is incredibly useful for better interpretation as it offers extra information about the shape and three-dimensional properties of the detected object.

Despite the higher computational power required by HoloOcean to generate sonar output due to its high-level rendering models, it ultimately produces a more realistic image overall.

In terms of implementation complexity, frequency update, and visualization settings, both simulation environments are quite different from each other. Stonefish simplifies the creation of acoustic images due to its less complex rendering algorithm and the absence of an external graphics engine, thereby making the initial configuration easier. Only a few classes need to be called into the main program to attach the sensor to the AUV and start generating sonar images.



(a) (b) (c)

Figure 15: a) Real sonar image where double and triple bounce beam effect between the object and the ground produce a visible sound echo *[46]*. b) Real sonar image with shadow *[45]*. c) Acoustic sonar image created by the forward-looking sonar in HoloOcean *[1]*.

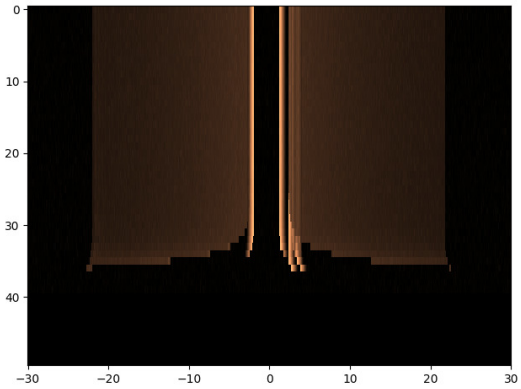
On the other hand, HoloOcean requires an external graphics engine to create octrees and simulate acoustic images. This leads to a longer procedure to configure the settings correctly, which can encounter issues due to constant package updates, Unreal Engine version mismatches and the use of the wrong external tools.

The frequency update of the Forward Looking Sonar sensor in HoloOcean might take several seconds since the generation of the image depends on the position of the AUV, along with the recalculation of the acoustic sonar image based on the already created octrees. This process requires a significant amount of time due to the heavy calculations performed by Unreal Engine, which can be accelerated with higher computational power. However, even with the use of the most modern hardware devices, the simulated sonar speeds can only be increased up to 2 times real-time. In Stonefish, the frequency update of the sensor aligns with the simulation speed, which updates every 0.005 seconds, resulting in a maximum frequency update of 200 Hz.

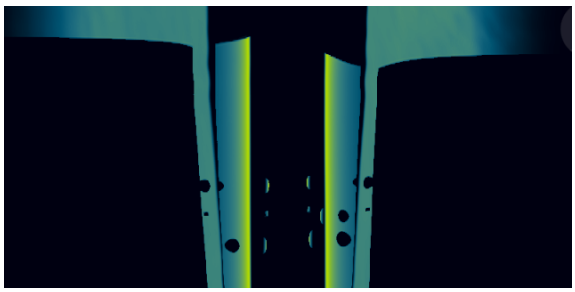
The configuration settings of the forward-looking sonar in both simulation environments are quite similar. The most significant difference lies in the initialization of parameters for Octree generation. Since HoloOcean utilizes Octrees to compute the sonar image, it is possible to modify the minimum and maximum Octree generation distances, directly impacting the image generated by the sensor. Additionally, visualization of the Octree calculation is also possible in HoloOcean.

In terms of the Side Scan Sonar, both simulation environments create realistic images, but problems in the final image as well as during the implementation of the sensor appeared in both HoloOcean and Stonefish. For HoloOcean, the biggest issue is that the Side Scan Sonar does not update the image, resulting in the final representation of the scenario being just the initial measurement repeated over and over (Fig. 16 *(a)*), leading to a wrong interpretation of the environment. This issue can be mitigated using the predefined worlds included within the HoloOcean packages, but the implementation of external shapes might not be considered for the editor at the moment of computing the octrees. Additionally, the waiting time for the generation of octrees is quite long and might encounter issues such as crashing or program malfunctioning. Despite the time-consuming procedure and potential problems, the few images successfully simulated by the sensor are commendable in terms of resolution and realism.

The implementation of the Side Scan Sonar in Stonefish is straightforward and provides excellent image visualization (Fig. 16 *(b)*), encompassing basic effects commonly observed in the representation of sonar images, such as shadows generated by the angle at which the beams hit basic shapes. However, despite the sonar's capability to replicate certain effects, the main concern arises from the lack of realistic shadowing effects overall. As a result, algorithms relying on shadowing to identify volume properties may not benefit from the dataset generated using Stonefish.



(a)



(b)

Figure 16: For both scenarios, several balls were placed between the AUV and two parallel walls. The maximum detection ranges in HoloOcean and Stonefish were set to 30 meters. a) Acoustic images created by the Side Scan sonar in HoloOcean b) Images simulated by Stonefish.

4.2 Comparative Analysis of Navigation Sensors

To commence a comprehensive evaluation of the navigation sensors implemented by both simulation environments, it is crucial to begin with sensors entirely related to underwater activities simulated by HoloOcean and Stonefish. The sensors comprising this subset are the Doppler Velocity Log, Inertial Measurement Unit, Depth sensor (The simulation of the depth sensor can be found in both simulation environments under different names and with slight differences between each other, but it is considered to be the same sensor since both use the pressure of the water over the robot as a parameter) and Odometry[9].

Doppler Velocity Log: This sensor is commonly used on underwater vehicles to estimate the velocity of the robot relative to the sea bottom. Consequently, the basic output of this sensor shows the velocity along the three axes. In both simulation environments, the velocity of the AUV is calculated efficiently with a maximum frequency update of 200 Hz. The difference between the simulation of the DVL is found in the additional information provided together with the speed of the vehicle. In HoloOcean, additional information about the range of beams in the x-axis pointing forward and backward is returned, as well as the range of beams in the y-axis. On the other hand, Stonefish provides more information about the environment in which the robot is found since water velocity along the three axes and the current altitude of the robot is also computed by the DVL.

Inertial Measurement Unit: The basic implementation of the Inertial Measurement Unit is also simulated by Stonefish and HoloOcean with a frequency update of 200 Hz, calculating angular velocities and acceleration along the three axes. The difference between both sensors is found in the additional output provided by Stonefish since values for Yaw, Roll, and Pitch are also given by the IMU to have a better interpretation of the environment. Even though HoloOcean can be modified to create a 4x3 output matrix with three more values than the nine channels provided by Stonefish, these values are merely the bias for each parameter previously mentioned. Therefore, the IMU implemented by Stonefish might be slightly more complex.

Depth Sensor: The implementation of this sensor in both simulators is quite different and it is only considered for a comparative analysis, given that the sensor in both Stonefish and HoloOcean is used to provide an approximation of the pressure measurement. In HoloOcean, this sensor returns a single scalar value with the current position of the AUV in the z-axis, while Stonefish displays an actual value of the pressure based on the characteristics of the water.

[9] *Odometry is a sensor implemented in Stonefish that combines the Location, Pose and Orientation sensors created by HoloOcean. Performing identical calculations as the three sensors simulated by HoloOcean, which is why it is considered for a deeper comparative analysis.*

Odometry: Serves as a ground truth sensor in Stonefish, providing the vehicle's position, angular velocity, orientation and linear velocity. In HoloOcean, an equivalent to this sensor is achieved through the implementation of three different sensors: the Location sensor, which provides the robot's position along the three axes; the Orientation sensor, which returns the forward, right, and up vectors relative to the agent; and the Pose sensor, which utilizes the orientation output to recalculate the robot's location.

Similar to the previous sensors, it operates at a frequency update of 200 Hz in both simulation environments, ensuring high-quality results. The principal difference is that Stonefish employs this sensor as the ground truth for the simulation.

The overall simulation of underwater sensors in Stonefish and HoloOcean is very similar in terms of frequency update and quality. However, Stonefish offers a broader output, providing additional information often essential for fully comprehending the interaction of the Autonomous Underwater Vehicle with the environment. In addition to the extra information provided by sensors that can be attached to the agent's body, the implementation of joint sensors contributes to better vehicle control in underwater activities.

HoloOcean may have fewer sensors available for simulating water vehicles, but since the use of an external graphics engine allows users to create larger environments, developing sensors in HoloOcean is also feasible.

Documentation detailing the correct procedure is available in the user manual [1]. While creating new sensors might require some time and could lead to complications during scenario compilation, the possibility of designing new classes might be easier given the guidance provided by the developers of HoloOcean.

In general, the sensors implemented by Stonefish are more complex and offer the research project a better overview of the environment. However, even though HoloOcean provides a developing option supported by the user manual, the creation of sensors in Stonefish is also feasible. This is because the open-source library is written in C++ and its structure allows for better interpretation by users with a solid programming background in this language.

4.3 Comparative Analysis of Physics-based Models

Accomplishing a detailed analysis of this section requires a significant amount of time and specific scenarios along with extensive knowledge in this area. Evaluating HoloOcean can be challenging due to its reliance on predefined worlds or the external graphics engine, which can include additional physics through community-made environments. Assessing the physics-based model is particularly challenging due to these factors.

To maintain simplicity and consistency and to form a strong opinion, HoloOcean will be evaluated based on the predefined worlds available for download.

Stonefish offers a comprehensive implementation of physics, including features such as current and water velocity along the three axes, added mass, drag, waves, buoyancy, modifications of ocean optics and collision detection. Additionally, Stonefish allows for the adjustment of these physics parameters within the world settings. Users have the flexibility to omit specific physics entirely or implement them strongly to create a highly challenging environment.

In contrast, HoloOcean does not allow for the detailed modification of physics to the same extent as Stonefish. While some physics are implemented, the ability to alter water properties directly is not possible within HoloOcean without utilizing the external graphics engine.

4.4 Analysis of Stonefish

Stonefish, as a simulation environment, offers numerous predefined tools that are highly beneficial for the development of autonomous activities. Due to the absence of an external graphics engine, implementing the simulator poses no significant difficulty, enabling users to invest their time in

algorithm development instead. The available documentation and the projects in which Stonefish has been involved provide valuable insights into the simulator's limitations and the range of scenarios that can be created.

Even though the Autonomous Underwater Vehicle GIRONA 500 is widely used within the marine robotics community for underwater activities, it is possible to develop new robots in Stonefish. Furthermore, modifying the physics of each agent part independently is relatively straightforward due to the availability of multiple classes in the open-source library.

Characteristics such as brightness, azimuth of the sun, water turbidity, and general properties that directly affect visual sensors (e.g., color cameras) can also be adapted to provide different testing environments in Stonefish. The modification of these values can be done either in the main CPP file or during the simulation through the graphical user interface available in Stonefish.

The graphical user interface (GUI) developed in Stonefish significantly expands the possibilities of making changes to the scenario while the simulation is running, without the need to directly modify the code. Additionally, adjustments to the actuators and sensors are also possible through this interface. This feature enables the development of semi-autonomous activities and allows for the display of additional information about the parameters and properties in the scenario. The implementation of this feature enhances the simulation value in Stonefish, particularly as HoloOcean lacks a GUI with similar characteristics.

The creation of C++ classes for actuators, linked sensors, joint sensors, graphics, entities, and other essential properties necessary for creating realistic simulations within the same directory simplifies the understanding of the simulator and how it works. This arrangement enables users to easily modify existing files or create new classes, allowing the customization of the scenario according to the aim of the research project.

The Stonefish open-source C++ library is continuously improving and regularly updated by its principal authors, ensuring that the simulation environment remains at the forefront of advancements in the marine robotics community.

4.5 Analysis of HoloOcean

The implementation of Unreal Engine alongside the classes created in Holodeck results in a highly realistic simulation environment. However, achieving the correct setting and selecting the appropriate version of the external graphics engine can present a significant challenge, as a good understanding of game engines is necessary for ensuring the functionality of HoloOcean.

The packages available for download and the predefined worlds created to test HoloOcean have several limitations. The incorporation of assets does not allow the modification of the scenario, thus reducing the possibilities for the development of algorithms and underwater activities in general. However, despite these limitations, the implementation of the side-scan sonar using the predefined worlds yielded better results than those obtained in the implementation of the HoveringAUV directly in the Unreal Engine editor.

The classes created for the sensors in HoloOcean are indeed quite limited, and the displayed output is significantly reduced, making it challenging to fully understand the vehicle's behavior in the current environment. Despite the small group of sensors simulated for underwater activities compared to Stonefish, they are easy to include in the simulation. Additionally, saving the values calculated by these sensors into a file is not difficult.

The implementation of Unreal Engine expands the possibility to create high-quality sensors and highly realistic underwater environments for testing autonomous vehicles. The continuously increasing interest in marine activities leads to the creation of new community-made worlds in this game engine, considerably improving HoloOcean. This is especially important given that the open-source library is barely updated and seems to have a mismatch between the version in which HoloOcean was created and the current editor version.

5 Conclusions

The characteristics of both simulators allow the creation of very different simulation environments. While Stonefish simulates water physics more efficiently than the worlds created by HoloOcean, it presents weaknesses in terms of the possible effects seen on acoustic images. Deciding between HoloOcean and Stonefish entirely depends on the aim of the research project since they are completely opposite in nature.

If the objective of the project is based on object detection or visual algorithms, the best option between these two simulators is HoloOcean. This is due to the implementation of an external graphics engine and the freedom to create new worlds using the editor of this game engine. However, this process takes considerable time. Therefore, any research project considering HoloOcean as the main simulator must be planned for medium to long duration to ensure the correct implementation of the simulation environment.

Stonefish might be better suited for navigational tasks and underwater activities, as the creation of classes allows the user to implement grabbers or similar robotic arms. The simulation of currents and other water properties ensures that the AUV is capable of performing activities in challenging conditions.

HoloOcean is currently a very comprehensive simulator for autonomous underwater vehicles. However, considering that the open-source library is not constantly improved and the Unreal Engine version used to create the worlds and scenarios is 4.27, the absence of new sensors and the poor update rate might lead to several version mismatches, resulting in a non-functional and outdated simulator. On the other hand, Stonefish is constantly improved by its authors and appears to be stable on many operating systems and computers.

In general, both HoloOcean and Stonefish are marine robotics simulators with high capabilities, demonstrating very good performance across different testing scenarios implemented throughout this research. While the strength of HoloOcean lies in the quality of the displayed acoustic images and the possibility of creating detailed underwater scenarios through Unreal Engine, this does not diminish Stonefish. Stonefish also allows for the creation of sonar images through different sensors and the modification of the scenario.

Similarly, Stonefish excels in simulating water properties and has a relatively simple simulation structure, but this does not mean that HoloOcean cannot simulate water physics. In other words, while one simulator may perform better in certain areas, it does not negate the potential of the other to be used in the same areas. Both simulators are highly capable and demonstrate excellent performance.

Some weaknesses present in both simulators include the noticeable absence of documentation regarding the techniques and methodology used by their sensors to provide the corresponding output. This information was often deduced by delving deeply into the C++ classes, OpenGL, Bullet Physics Library, and PhysX engine. The absence of information about the sensors may lead to a misunderstanding of their functionality. Considering that every single aspect must be considered when choosing the correct simulator, the lack of information in this aspect is something that can be improved in both HoloOcean and Stonefish.

References

[1] *E. Potokar and S. Ashford and M. Kaess and J. Mangelson,"HoloOcean: An Underwater Robotics Simulator", Proc. IEEE Intl. Conf. on Robotics and Automation, ICRA, Philadelphia, USA, May 2022*

[2] *Patryk Cieślak, "Stonefish: An Advanced Open-Source Simulation Tool Designed for Marine Robotics, With a ROS Interface", In Proceedings of MTS/IEEE OCEANS 2019, June 2019, Marseille, France*

[3] *D. Ribas, N. Palomeras, P. Ridao, M. Carreras and A. Mallios, "Girona 500 AUV: From Survey to Intervention," in IEEE/ASME Transactions on Mechatronics, vol. 17, no. 1, pp. 46-53, Feb. 2012*

[4] *Erwin Coumans, Bullet 2.83 Physics SDK Manual, 2015. [Online]. Available: http://bulletphysics.org*

[5] *John Kessenich, Dave Baldwin, Randi Rost, "The OpenGL® Shading Language", Document Revision 7, Language Version: 4.50, 09 May 2017, Editor: John Kessenich, Google, Version 1.1.*

[6] *Rômulo Cerqueira, Tiago Trocoli, Jan Albiez, Luciano Oliveira, "A rasterized ray-tracer pipeline for real-time, multi-device sonar simulation", Graphical Models, Volume 111, 2020, 101086, ISSN 1524-0703, https://doi.org/10.1016/j.gmod.2020.101086.*

[7] *Prats, M.; Perez, J.; Fernandez, J.J.; Sanz, P.J., "An open source tool for simulation and supervision of underwater intervention missions", 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2577-2582, 7-12 Oct. 2012*

[8] *Pi Roger, Cieslak Patryk, Ridao Pere, Sanz Pedro. (2021). TWINBOT: Autonomous underwater cooperative transportation. IEEE Access. PP. 1-1. 10.1109/ACCESS.2021.3063669.*

[9] *I. Lončar et al., "MARUS - A Marine Robotics Simulator," OCEANS 2022, Hampton Roads, Hampton Roads, VA, USA, 2022, pp. 1-7, doi: 10.1109/OCEANS47191.2022.9976969*

[10] *Amer Abdelhakim, Alvarez-Tunon Olaya, Uğurlu Halil, Sejersen Jonas, Brodskiy Yury, Kayacan Erdal. (2023). "UNav-Sim: A Visually Realistic Underwater Robotics Simulator and Synthetic Data-Generation Framework." 570-576. 10.1109/ICAR58858.2023.10406819.*

[11] *Stålberg J., Ulmstedt M. (2019). "GPU Accelerated Ray-tracing for Simulating Sound Propagation in Water". Master of Science Thesis in Electrical Engineering, Department of Electrical Engineering, Linköping University.*

[12] *Gaspar A.R., Matos A. "Feature-based Place Recognition Using Forward Looking Sonar", 27 October 2023, doi: 10.20944/preprints202310.1759.v1.*

[13] *Choi W.-S., Olson, D. R. Davis D., Zhang M., Racson A., Bingham B., McCarrin M., Vogt C., Herman J. (2021). "Physics-Based Modelling and Simulation of Multibeam Echosounder Perception for Autonomous Underwater Manipulation." Frontiers in Robotics and AI, 8, 706646. doi:10.3389/frobt.2021.706646*

[14] *Eberly, D. (1999). "Euler Angle Formulas." Geometric Tools, Redmond WA 98052. Available at: https://www.geometrictools.com/. Last modified: April 28, 2020.*

[15] *Slabaugh, G.G. (1999). "Computing Euler angles from a rotation matrix."*

[16] *Bellavia F., Fanfani M., Colombo C. (2017). "Selective visual odometry for accurate AUV localization." Autonomous Robots, 41, 133–143. doi:10.1007/s10514-015-9541-1*

[17] *Saksvik, I.B., Alcocer, A., Hassani, V. (2021). "A Deep Learning Approach To Dead-Reckoning Navigation For Autonomous Underwater Vehicles With Limited Sensor Payloads." Department of Mechanical, Electronic and Chemical Engineering, Oslo Metropolitan University, Oslo, Norway.arXiv:2110.00661v1 [cs.RO]. 1 Oct 2021.*

[18] *Lindve, K. (2021). "Tracking of Sinking Underwater Node Using Inertial Navigation." Mälardalen University, School of Innovation Design and Engineering, Västerås, Sweden. Date: 16/06/2021.*

[19] *Prof. Marco G. Beghi, "Ray Trace Modeling of Underwater Sound Propagation", August 2013, DOI: 10.5772/55935, In: Modeling and Measurement Methods for Acoustic Waves and for Acoustic Microdevice, Publisher: InTech - Open Access Publisher, Rijeka, Croatia, 2013.*

[20] *Potokar, Easton Robert, "Simulation and Localization of Autonomous Underwater Vehicles Leveraging Lie Group Structure" (2022). Theses and Dissertations. 10012. Available at: https://scholarsarchive.byu.edu/etd/10012*

[21] *Vadim Kramar, Aleksey Kabanov, and Kirill Dementiev, "Autonomous Underwater Vehicle Navigation via Sensors Maximum-Ratio Combining in Absence of Bearing Angle Data,". Department of Informatics and Control in Technical Systems, Sevastopol State University, 299053 Sevastopol, Russia. https://doi.org/10.3390/jmse11101847*

[22] *C. D. Monaco, S. N. Brennan and K. A. Hacker, "Doppler Velocity Log Placement Effects on Autonomous Underwater Vehicle Navigation Accuracy", 2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV), Porto, Portugal, 2018, pp. 1-6, doi: 10.1109/AUV.2018.8729737.*

[23] *Øyvind Hegrenæs, Audun Ramstad, Torstein Pedersen, David W. Velasco, "Validation of a new generation DVL for underwater vehicle navigation", November 2016, DOI: 10.1109/AUV.2016.7778694, Conference: 2016 IEEE/OES Autonomous Underwater Vehicles (AUV)* Here's the formatted reference based on the provided information:

[24] *P. Cheng, B. Oelmann, "Joint-Angle Measurement Using Accelerometers and Gyroscopes — A Survey," Instrumentation and Measurement, IEEE Transactions on, Volume 59, Number 2, Pages 404-414, February 2010.*

[25] *Thomas Seel, Thomas Schauer. "IMU-based Joint Angle Measurement Made Practical", January 2013, Conference: Proc. of the 4th European Conference on Technically Assisted Rehabilitation - TAR 2013. Available from: https://www.researchgate.net/publication/264230720_IMU-based_Joint_Angle_Measurement_Made_Practical*

[26] *Ubuntu. (2024). Ubuntu Server Guide. https://assets.ubuntu.com/v1/544d9904-ubuntu-server-guide-2024-01-22.pdf*

[27] *S. Macenski, A. Soragna, M. Carroll, Z. Ge, "Impact of ROS 2 Node Composition in Robotic Systems", IEEE Robotics and Autonomous Letters (RA-L), 2023.*

[28] *"Intel Corporation. (2011). Intel® Core™ i5-600, i3-500 Desktop Processor Series and Intel® Pentium® Desktop Processor 6000 Series", Datasheet – Volume 2. Document Number: 322910-003.*

[29] *Intel Corporation. (n.d.). Intel Iris Xe Dedicated Graphics Card - 80 EU: Specifications. https://www.intel.com/content/www/us/en/products/sku/211014/intel-iris-xe-dedicated-graphics-card-80-eu/specifications.html*

[30] *NVIDIA Corporation. (2023). "NVIDIA Data Center GPU Driver version 535.104.05 (Linux)/537.13 (Windows)". RN-08625-535 v3.0. https://www.nvidia.com/en-us/data-center/gpu-driver-support/*

[31] *Ribas D., Ridao P., Magí L., Palomeras N., Carreras M. (2011). "The Girona 500, a multipurpose autonomous underwater vehicle" OCEANS 2011 IEEE - Spain, 1-5.*

[32] *The MathWorks Inc. (2022). MATLAB version: 9.13.0 (R2022b), Natick, Massachusetts: The MathWorks Inc. https://www.mathworks.com*

[33] *Epic Games. (2019). "Unreal Engine." Retrieved from https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/Linux/BeginnerLinuxDeveloper/SettingUpAnUnrealWorkflow/*

[34] *J. Greaves, M. Robinson, N. Walton, M. Mortensen, R. Pottorff, C. Christopherson, D. Hancock, J. Milne, and D. Wingate, "Holodeck: A high fidelity simulator,"* https://github.com/BYU-PCCL/holodeck,2018.

[35] *Brockman G., Cheung V., Pettersson L., Schneider J., Schulman J., Tang J.,Zaremba W. (2016). "OpenAI Gym". Retrieved from* https://github.com/openai/gym

[36] *VIDIA, "Physx",* https://github.com/NVIDIAGameWorks/PhysX

[37] *Yang R., Hou M., Wang J., Zhang, F. (2023). "OceanChat: Piloting Autonomous Underwater Vehicles in Natural Language". Retrieved from arXiv preprint server (Primary Class: cs.RO). (eprint: 2309.16052)*

[38] *Revelles J., Ureña C., Lastra M. (May 2000). "An Efficient Parametric Algorithm for Octree Traversal". Retrieved from SourceCiteSeer, E. T. S. Ingeniera Informatica.*

[39] *Laurmaa V., Picasso M., Steiner G. (March 2016). "An octree-based adaptive semi-Lagrangian VOF approach for simulating the displacement of free surfaces". Computers and Fluids, 131. DOI: 10.1016/j.compfluid.2016.03.005. License: CC BY 4.0.*

[40] *Potokar, E. Lay, K. Norman, K. Benham, D. Neilsen, T. B. Kaess, M. Mangelson J. G. "HoloOcean: Realistic Sonar Simulation."*

[41] *Westman E. (2019). "Underwater Localization and Mapping with Imaging Sonar" (CMU-RI-TR-19-77). The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213.*

[42] *J. Park and J. Kim, "Robust Underwater Localization Using Acoustic Image Alignment for Autonomous Intervention Systems," in IEEE Access, vol. 10, pp. 58447-58457, 2022, doi: 10.1109/ACCESS.2022.3179430.*

[43] *Robotics, B. BlueROV2: The World's Most Affordable High-Performance ROV. "In BlueROV2 Datasheet". Blue Robotics: Torrance,CA, USA, June 2016.*

[44] *Von Benzon, Malte Sørensen, Fredrik Uth, Esben Jouffroy, Jerome Liniger, Jesper Pedersen, Simon. (2022). An Open-Source Benchmark Simulator: Control of a BlueROV2 Underwater Robot. Journal of Marine Science and Engineering. 10. 1898. 10.3390/jmse10121898.*

[45] *Xi Jier and Xiufen Ye. (2024). "Sonar Image Target Detection Based on Simulated Stain-like Noise and Shadow Enhancement in Optical Images under Zero-Shot Learning" Journal of Marine Science and Engineering 12, no. 2: 352. https://doi.org/10.3390/jmse12020352*

[46] *Wang Y., Wu C., Ji, Y. Tsuchiya, H. Asama, H. Yamashita, A. (2024, February 24). "2D Forward Looking Sonar Simulation with Ground Echo Modeling" (Version 2).*

A *Collected Data and Code*

A.1 Stonefish

```
void UnderwaterTestManager::SimulationStepCompleted(sf::Scalar timeStep)
{
sf::FLS* fls = (sf::FLS*)getRobot("GIRONA500")->getSensor("FLS");
if(Time > 1 && Time < 1.05) {
   std::ofstream outFile("/home/external-holo/stonefish/Simulator/Sim2/dataSim2.txt");
   std::ostream &output = outFile.is_open() ? outFile : std::cout;
   GLubyte *displayData = fls->getDisplayDataPointer();

   if ((displayData != nullptr) && (outFile.is_open())) {

   unsigned int width = 480;
   unsigned int height = 264;

   for (unsigned int y = 0; y < height; ++y) {
      outFile << std::endl;
      for (unsigned int x = 0; x < width; ++x) {

         unsigned int index = (y * width + x) * 3;
         GLubyte red = displayData[index];
         GLubyte green = displayData[index + 1];
         GLubyte blue = displayData[index + 2];
         output << "(" << x << ", " << y << "): R=" << (int) red << ", G=" << (int) green
             << ", B="
         << (int) blue << " ";
         std::cout << "(" << x << ", " << y << "): R=" << (int) red << ", G=" << (int)
             green << ", B="
         << (int) blue << std::endl;
      }
   }
   std::cout << "Pixel data has been written to pixel_data.txt." << std::endl;
   } else {
   std::cerr << "Display data pointer is null." << std::endl;
   }

   if (outFile.is_open()) {
   outFile.close();
           }
   }
}
```

Listing 1: Code implemented to collect the sonar image data

Measurements from FLS
Number of channels: *3*
Number of samples: *88800*
Number of beams: *256*
Horizontal Field of View: *150*
Vertical Field of View: *30*
Max Range: *20*
Display On Screen Scale: *0.5*

Width (i) / Height (j)	Width (0, j)	Width (1, j)	Width (2, j)	Width (3, j)	...	Width (479, j)
Height (i, 0)	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	...	...
Height (i, 1)	R=97, G=0, B=19	R=0, G=96, B=6	R=5, G=74, B=5	R=40, G=70, B=0	...	...
Height (i, 2)	R=1, G=60, B=16	R=128, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	...	...
Height (i, 3)	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=91	R=148, G=11, B=224	...	...
Height (i, 4)	R=60, G=238, B=45	R=157, G=0, B=0	R=0, G=0, B=204	R=139, G=176, B=72	...	...
Height (i, 5)	R=60, G=226, B=45	R=157, G=0, B=0	R=0, G=0, B=204	R=184, G=176, B=72	...	...
Height (i, 6)	R=125, G=83, B=42	R=157, G=0, B=0	R=0, G=0, B=132	R=145, G=175, B=72	...	...
Height (i, 7)	R=104, G=0, B=0	R=3, G=5, B=0	R=0, G=0, B=76	R=0, G=0, B=2	...	...
Height (i, 8)	R=104, G=0, B=0	R=3, G=5, B=0	R=0, G=0, B=76	R=0, G=0, B=2	...	...
Height (i, 9)	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	...	...
Height (i, 10)	R=158, G=248, B=45	R=157, G=0, B=0	R=0, G=0, B=236	R=212, G=176, B=72	...	...
Height (i, 11)	R=16, G=6, B=16	R=6, G=16, B=6	R=16, G=6, B=16	R=6, G=16, B=6	...	...
Height (i, 12)	R=22, G=249, B=45	R=157, G=0, B=0	R=0, G=0, B=172	R=210, G=176, B=72	...	...
Height (i, 13)	R=97, G=120, B=45	R=157, G=0, B=0	R=0, G=0, B=44	R=247, G=178, B=72	...	...
Height (i, 14)	R=30, G=71, B=45	R=157, G=0, B=0	R=0, G=0, B=12	R=5, G=178, B=72	...	...
Height (i, 15)	R=143, G=247, B=45	R=157, G=0, B=0	R=0, G=0, B=108	R=235, G=176, B=72	...	...
Height (i, 16)	R=105, G=234, B=45	R=157, G=0, B=0	R=0, G=0, B=140	R=155, G=176, B=72	...	...
Height (i, 17)	R=236, G=85, B=42	R=157, G=0, B=0	R=0, G=0, B=100	R=138, G=175, B=72	...	...
Height (i, 18)	R=40, G=86, B=42	R=157, G=0, B=0	R=0, G=0, B=132	R=133, G=175, B=72	...	...
Height (i, 19)	R=0, G=0, B=0	R=0, G=0, B=255	R=0, G=255, B=2	R=1, G=1, B=1	...	...
Height (i, 20)	R=255, G=254, B=1	R=0, G=0, B=0	R=255, G=0, B=0	R=255, G=1, B=0	...	...
Height (i, 21)	R=3, G=255, B=0	R=0, G=255, B=0	R=254, G=1, B=255	R=255, G=255, B=0	...	...
Height (i, 22)	R=3, G=0, B=0	R=2, G=0, B=1	R=2, G=0, B=0	R=0, G=255, B=0	...	...
Height (i, 23)	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=1, B=0	R=0, G=0, B=0	...	...
Height (i, 24)	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	R=0, G=0, B=0	...	...
Height (i, 25)	R=0, G=1, B=255	R=0, G=1, B=254	R=0, G=0, B=254	R=0, G=0, B=255	...	...
Height (i, 26)	R=0, G=1, B=0	R=1, G=0, B=1	R=0, G=0, B=1	R=0, G=1, B=0	...	...
Height (i, 27)	R=2, G=0, B=1	R=3, G=255, B=0	R=1, G=0, B=2	R=255, G=0, B=254	...	...
Height (i, 28)	R=255, G=0, B=255	R=254, G=0, B=2	R=1, G=0, B=0	R=255, G=0, B=1	...	...
Height (i, j)	...	...	...	...	...	...
...	...	...	...	...	...	...
Height (i, 263)	...	...	...	...	...	...

Table 4: An example of the data collected using pointers to access the information of each pixel displayed in the sonar image. The values for width and height are defined based on the scale option, along with other properties such as maximum range and horizontal field of view.

```
void UnderwaterTestManager::SimulationStepCompleted(sf::Scalar timeStep)
{
   cInfo("Simulation time: %1.3lf", getSimulationTime());

   sf::IMU* imu = (sf::IMU*)getRobot("GIRONA500")->getSensor("IMU");
   sf::DVL* dvl = (sf::DVL*)getRobot("GIRONA500")->getSensor("DVL");

   sf::Thruster* th_r =
       (sf::Thruster*)getRobot("GIRONA500")->getActuator("ThrusterSurgePort");
   sf::Thruster* th_l =
       (sf::Thruster*)getRobot("GIRONA500")->getActuator("ThrusterSurgeStarboard");

   std::cout << "x-axis rotation (Roll): " << imu->getValue(0, 0) << ", y-axis rotation
       (Pitch): "
   << imu->getValue(0,1) << ", z-axis rotation(Yaw): " << imu->getValue(0,2) << std::endl;
   std::cout << "Angular Velocity X: " << imu->getValue(0,3) << ", Angular Velocity Y: "
   << imu->getValue(0, 4) << ", Angular Velocity Z: " << imu->getValue(0, 5) <<std::endl;
   std::cout << "Linear Acceleration X: " << imu->getValue(0,6) << ", Linear Acceleration
       Y: "
   << imu->getValue(0, 7) << ", Linear Acceleration Z: " << imu->getValue(0, 8)
       <<std::endl;

   imu->SaveMeasurementsToTextFile(
   "/home/external-holo/stonefish/SimTry/Simulation/IMU.txt", true, 3);
   printf("BeamAngle: %1.3lf\n", dvl->getBeamAngle());

   std::cout << "Velocity: (X:" << dvl->getValue(0, 0) << ", Y:" << dvl->getValue(0,1) <<
       ", Z:" <<
   dvl->getValue(0,2) << "), " << "Altitude: " << dvl->getValue(0,3) << ", Water
       Velocity: (X: " <<
   dvl->getValue(0,4) << ", Y: " << dvl->getValue(0,5) << ", Z: "<< dvl->getValue(0,6) <<
       ")" << std::endl;
   dvl->SaveMeasurementsToTextFile(
   "/home/external-holo/stonefish/SimTry/Simulation/DVL.txt", true, 5);

   printf("Setpoint: %1.3lf Thrust: %1.3lf Torque: %1.3lf\n Setpoint: %1.3lf Thrust:
       %1.3lf Torque: %1.3lf\n", th_r->getSetpoint(), th_r->getThrust(),
       th_r->getTorque(), th_l->getSetpoint(), th_l->getThrust(), th_l->getTorque());
        std::cout << std::endl;

   std::ofstream outfile("/home/external-holo/stonefish/SimTry/Simulation/Thruster.txt",
       std::ios::app);
   std::ostream& output = outfile.is_open() ? outfile : std::cout;
      output <<"Simulation time: " << getSimulationTime()
   << "\n Setpoint (RT): " << th_r->getSetpoint() << ", Thrust (RT): " <<
       th_r->getThrust() << ", Torque (RT): " << th_r->getTorque() << " Setpoint (LT): "
       << th_l->getSetpoint() << ", Thrust (LT): " << th_l->getThrust() << ", Torque
       (LT): " << th_l->getTorque() << std::endl;

   if (outfile.is_open()) {
      outfile.close();
   }
}
```

Listing 2: Code implemented to collect Doppler Velocity Log, Inertial Measurement Unit and Thruster Force data.

Measurements from IMU
Number of channels: *9*
Number of samples: *2083*
Frequency: *200.000 Hz*
Unit System: *SI*

Time	Roll	Pitch	Yaw	Ang Vel X	Ang Vel Y	Ang Vel Z	Lin Accel X	Lin Accel Y	Lin Accel Z
0.000	0.000	0.000	0.000	-0.007	0.008	-0.041	-0.010	0.007	-0.500
0.010	-0.000	0.000	-0.000	0.003	0.069	-0.099	0.098	-0.004	-0.500
0.020	-0.000	0.000	-0.000	0.006	-0.014	-0.141	0.067	-0.004	-0.500
0.030	-0.000	0.000	-0.000	-0.106	0.055	-0.026	0.092	0.000	-0.500
0.040	-0.000	0.000	-0.000	0.022	-0.090	0.080	0.086	-0.010	-0.500
0.050	-0.000	0.001	-0.000	-0.059	0.004	0.054	0.072	-0.012	-0.500
0.060	-0.000	0.001	-0.000	0.025	0.014	-0.058	0.080	0.002	-0.500
0.070	-0.000	0.001	-0.000	0.014	0.075	0.033	0.082	-0.010	-0.500
0.080	-0.000	0.001	-0.000	-0.053	0.017	-0.015	0.084	-0.000	-0.500
0.090	-0.000	0.002	-0.000	0.017	-0.051	-0.027	0.083	0.008	-0.500
0.100	-0.000	0.002	-0.000	0.019	-0.057	0.174	0.097	-0.008	-0.500
0.110	-0.000	0.002	-0.000	-0.021	0.066	0.040	0.115	-0.003	-0.500
0.120	-0.000	0.003	-0.000	0.006	0.077	0.014	0.118	0.019	-0.500
0.130	-0.000	0.003	-0.000	0.038	0.012	-0.066	0.115	-0.007	-0.500
0.140	-0.000	0.004	-0.000	0.000	0.010	0.054	0.118	0.005	-0.500
0.150	-0.000	0.004	-0.000	-0.096	0.094	0.152	0.118	-0.007	-0.500
...	...	...	...	...	...	...	...	...	...
11.405	-0.000	-0.001	0.075	0.034	-0.093	0.150	-0.204	-0.005	-0.500

Table 5: Data collected by the Inertial Measurement Unit sensor during Testing 2, all nine channels are displayed with their corresponding values at the current time.

Measurements from Joint Sensor
Right Thruster: (RT)
Left Thruster: (LT)

Simulation Time	Setpoint (RT)	Thrust (RT)	Torque (RT)	Setpoint (LT)	Thrust (LT)	Torque (LT)
0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.500	-0.000	0.000	-0.000	0.000	0.000	0.000
1.000	-0.000	0.000	-0.000	0.000	0.000	0.000
1.500	-0.000	0.000	-0.000	0.000	0.000	0.000
2.000	-0.000	0.000	-0.000	0.000	0.000	0.000
2.500	-0.000	0.000	-0.000	0.000	0.000	0.000
...	...	...	...	...	...	...
11.390	-0.000	0.000	-0.000	0.000	0.000	0.000
11.400	-0.000	0.000	-0.000	0.000	0.000	0.000

Table 6: Data collected by the joint sensor attached to the thruster during Testing 2. Since both thrusters were set to 0 to observe the current and ocean properties acting on the GIRONA 500, the measurements written in the file confirm that the AUV did not change state by itself, but rather due to the physical properties of the ocean.

Measurements from Odometry and Compass (Direction)
Number of channels: *13*
Number of samples: *4683*
Frequency: *200.000 Hz*
Unit System: *SI*

Time	Position X	Position Y	Compass
0.000	0.000125204	3.24758e-09	0.0161785
2.500	0.0747431	0.000425169	0.0525263
5.000	0.207223	0.00424298	0.0779986
7.500	0.380264	0.0161407	0.17265
10.000	1.46032	0.236123	0.269442
12.500	3.03649	0.616632	0.288383
15.000	4.11568	0.911117	0.34818
17.500	4.70733	1.07681	0.393466
20.000	5.0956	1.18861	0.370928
22.500	5.40722	1.28324	0.348056
25.000	5.76437	1.39796	0.170724
27.500	6.2161	1.45287	-0.82747
30.000	6.57185	1.23623	-2.05492
32.500	6.54336	0.669895	-2.1509
35.000	6.24915	-0.031872	-2.06853
37.500	4.09115	-3.90128	-2.02344
40.000	3.66521	-4.72705	-1.96855
42.500	3.36482	-5.34978	-1.97948
45.000	3.13247	-5.8466	-1.99104
47.500	2.98332	-6.34892	-0.515734
50.000	3.23552	-6.58378	1.23272
52.500	3.35083	-6.36661	3.02316
55.000	2.68906	-6.03639	2.77125
57.500	1.50454	-5.5694	2.71135
60.000	0.589857	-5.20404	2.78084
62.500	0.00585696	-4.97058	2.36746
...	...	...	...
248.000	2.94036	0.585454	0.313393

Table 7: Data collected using the Compass and Odometry sensors so it can be used to determine the position of the GIRONA 500 along the simulation (Testing 4).

A.2 HoloOcean

```
config = cfg['agents'][0]['sensors'][-1]["configuration"]
azi = config['Azimuth']
minR = config['RangeMin']
maxR = config['RangeMax']
binsR = config['RangeBins']
binsA = config['AzimuthBins']

plt.ion()
fig, ax = plt.subplots(subplot_kw=dict(projection='polar'), figsize=(8,5))
ax.set_theta_zero_location("N")
ax.set_thetamin(-azi/2)
ax.set_thetamax(azi/2)

theta = np.linspace(-azi/2, azi/2, binsA)*np.pi/180
r = np.linspace(minR, maxR, binsR)
T, R = np.meshgrid(theta, r)
z = np.zeros_like(T)

plt.grid(False)
plot = ax.pcolormesh(T, R, z, cmap='gray', shading='auto', vmin=0, vmax=1)
plt.tight_layout()
fig.canvas.draw()
fig.canvas.flush_events()

command = np.array([-10,-10,-10,-10,0,0,0,0])
with holoocean.make(scenario_cfg=cfg) as env:

   env.spawn_prop(prop_type='sphere', location=[0,-2,-5], rotation=None, scale=1,
              sim_physics=False, material="white", tag="figure")

   env.spawn_prop(prop_type='sphere', location=[0,2,-5], rotation=None, scale=1,
              sim_physics=False, material="white", tag="figure")

   for i in range(1000):
      env.act("auv0", command)
      state = env.tick()

      if 'ImagingSonar' in state:
         s = state['ImagingSonar']
         print(s.ravel())
         with open("RGBData.txt", "a") as file:
            file.write("Imaging Sonar:\n")
            file.write(np.array2string(s.ravel(), separator=', ') + "\n\n")
         #Pixel-by-pixel RGB combination
         plot.set_array(s.ravel())

         fig.canvas.draw()
         fig.canvas.flush_events()

print("Finished Simulation!")
plt.ioff()
plt.show()
```

Listing 3: The code was modified from one of the predefined worlds available for download in HoloOcean to incorporate the Forward Looking Sonar and obtain the RGB combination (Testing 1).

```
with HoloOceanEnvironment(scenario=config, start_world=False) as env:
   while True:

      command = parse_keys(pressed_keys, force)
      env.act("auv0", command)
      state = env.tick()
      RotationSensor = state["RotationSensor"]
      Location = state["LocationSensor"]
      IMU = state["IMUSensor"]
      DVL = state["DVLSensor"]

      counter += 1

      print("This is the Rotation Sensor \n",RotationSensor)
      print("This is the Location \n",Location)
      print("This is the IMU \n", IMU)
      print("This is the DVL \n", DVL)

      with open("SensorMeasurements.txt", "a") as file:
         file.write(f"Measurement {counter}:\n")
         file.write("Rotation Sensor:\n")
         file.write(str(RotationSensor) + "\n\n")
         file.write("IMU:\n")
         file.write(str(IMU) + "\n\n")
         file.write("DVL:\n")
         file.write(str(DVL) + "\n\n")
         file.write("Location:\n")
         file.write(str(Location) + "\n\n")
```

Listing 4: Code implemented to collect Doppler Velocity Log, Inertial Measurement Unit and Pose Sensor data (Testing 2).

Measurement 1:				
Rotation Sensor:				
[-179.9997	-0.000	0.000]		
IMU:				
[[-4.1144827e-04	3.5459730e-03	-1.0944549e+00]		
[4.6713068e-03	9.8938849e-03	3.6378473e-03]		
[-5.1847938e-04	6.0697709e-04	-2.0389778e-03]		
[5.6857578e-03	8.8871308e-03	4.3000188e-03]]		
DVL:				
[9.6999211e-03	2.0681279e-02	1.5014126e+00	5.0094421e+01	4.9956100e+01
5.0049618e+01	5.0138783e+01]			
Location:				
[9.842999	-0.04999999	8.316351]		
Measurement 2:				
Rotation Sensor:				
[-1.7999936e+02	-2.7320753e-05	-5.4696848e-06]		
...				

Table 8: Data collected by the DVL, IMU, and Pose Sensor during Testing 2.

```
import cv2
import numpy as np
from pynput import keyboard
import matplotlib.pyplot as plt
from holoocean.environments import HoloOceanEnvironment

config ={  "name": "HoveringImagingSonar",
   "world": "SimpleUnderwater",
   "main_agent": "auv0",
   "ticks_per_sec": 200,
   "frames_per_sec": True,
   "octree_min": 0.02,
   "octree_max": 5.0,
   "agents":[
   {
      "agent_name": "auv0",
      "agent_type": "HoveringAUV",
      "sensors": [
      {
         "sensor_type": "ImagingSonar",
         "socket": "SonarSocket",
         "Hz": 1,
         "configuration": {
            "RangeBins": 256,
            "AzimuthBins": 64,
            "RangeMin": 1,
            "RangeMax": 30,
            "InitOctreeRange": 50,
            "Elevation": 20,
            "Azimuth": 120,
            "AzimuthStreaks": -1,
            "ScaleNoise": True,
            "AddSigma": 0.08,
            "MultSigma": 0.08,
            "RangeSigma": 0.1,
            "MultiPath": True,
            "ViewRegion": True,
            "ViewOctree": -1
         }
      }
      ],
      "control_scheme": 0,
      "location": [-2,0,8],
      "rotation": [0.0, 0.0, 0.0]
   }
   ],
   "window_width": 800,
   "window_height": 1000
}

sonar_config = config['agents'][0]['sensors'][-1]["configuration"]
azi = sonar_config['Azimuth']
minR = sonar_config['RangeMin']
maxR = sonar_config['RangeMax']
binsR = sonar_config['RangeBins']
binsA = sonar_config['AzimuthBins']
plt.ion()
fig, ax = plt.subplots(subplot_kw=dict(projection='polar'), figsize=(8,5))
ax.set_theta_zero_location("N")
ax.set_thetamin(-azi/2)
```

```
ax.set_thetamax(azi/2)
theta = np.linspace(-azi/2, azi/2, binsA)*np.pi/180
r = np.linspace(minR, maxR, binsR)
T, R = np.meshgrid(theta, r)
z = np.zeros_like(T)
plt.grid(False)
plot = ax.pcolormesh(T, R, z, cmap='gray', shading='auto', vmin=0, vmax=1)
plt.tight_layout()
fig.canvas.draw()
fig.canvas.flush_events()

command = np.array([0,0,0,0,0,0,0,0])
with HoloOceanEnvironment(scenario=config, start_world=False) as env:
   for i in range(800):
      env.act("auv0", command)
      state = env.tick()

      if 'ImagingSonar' in state:
         s = state['ImagingSonar']
         plot.set_array(s.ravel())

         fig.canvas.draw()
         fig.canvas.flush_events()

print("Finished Simulation!")
plt.ioff()
plt.show()
```

Listing 5: Code used to implement the HoveringAUV and Forward-Looking Sonar into the Unreal Engine scenario (Testing 3).

B *Additional Tools*

B.1 Matlab2023b

```
filename = "Path3.xlsx";
encoders = xlsread(filename);
Position_X = encoders(:, 1);
Position_Y = encoders(:, 2);

% Select every 500th element from Position_X and Position_Y
Position_X_sampled = el(1:500:end);
Position_X_sampled = er(1:500:end);

% Plot the sampled data
plot(Position_X_sampled, Position_Y_sampled, '-');
hold on;

% Add a marker that represents the location of the spheres
marker_x = [-2, 8, 2, -8]; % X coordinates of the points
marker_y = [8, 2, -8, -2]; % Y coordinates of the points

plot(marker_x, marker_y, 'r.', 'MarkerSize', 20);
title("Trajectory of the robot", "FontSize", 8);
xlabel("X", "FontSize", 8);
ylabel("Y", "FontSize", 8);
grid on;
axis([-12 12 -12 12]);


for i = 1:numel(el_sampled)-1
% Update robot's position
robot_x = Position_X_sampled(i+1);
robot_y = Position_Y_sampled(i+1);

% Plot robot's updated position
quiver(robot_x, robot_y, cos(yaw_sampled(i+1)), sin(yaw_sampled(i+1)), 'Color',
    'b','ShowArrowHead','on','LineWidth',1); % Plot heading
plot(robot_x, robot_y, 'b.', 'MarkerSize', 15); % Plot robot's position

% Pause to visualize movement
pause(0.5);

% Update plot
drawnow;
end
```

Listing 6: Code implemented in MATLAB 2023b to plot the position, the spheres and direction of the AUV.