ACIT5930

MASTER'S THESIS

in

Applied Computer and Information Technology (ACIT)

May 2023

Robotics and Control

Visual-inertial-pressure SLAM for Low-cost Remotely Operated Vehicles

Einar Sobrevega Tomter

Department of Computer Science
Faculty of Technology, Art, and Design

OSLOMET

Preface

I have always liked building and experimenting with things ever since I was little. However, I did not know what the term was for those who built all these amazing inventions and technologies. In fact, it was not until the age of 16 that I realized what the term "Engineer" really meant. That is also when my journey into the realm of technology began. During my bachelor's study at OsloMet, I was fortunate enough to come across like-minded individuals who also had a passion for learning and discovery, from my fellow classmates and professors alike. That was also when I first encountered robotics, and that is when I knew exactly what I wanted to pursue.

For my project, I wanted something that involved math (I know) and was visually intriguing. When I discovered what SLAM was, I knew I wanted to learn more about it and see how far I could take it.

With that, I would like to thank firstly my supervisor, Alex Alcocer, for encouraging me not just in my thesis, but in all things robotics. Secondly, I would also like to thank my classmates and fellow researchers in Oceanlab, especially Erik skultety, Pierre Odin Boniface, Jan-Philip Radicke, and Aksel Johan Frafjord. The course would not have been as enjoyable without you guys. Thirdly, I want to thank Daniel Gomez Ibanez, my earlier co-supervisor, who helped me in writing my paper even after he returned to the US. Finally, but most certainly not least, I would like to thank my closest friends and family, who have supported me to their fullest extent and helped me get through the hardest of times.

Einar Tomter

Enebakkveien, 15.05.2023

Abstract

In recent years, underwater robots have become cheaper and more readily accessible, allowing smaller companies and even private individuals to conduct their own research and explore the underwater domain.

This thesis presents two main contributions to the field of underwater robotics. The first contribution is the development of an underwater simulation environment using Unreal Engine and Gazebo, designed for visual SLAM and other robotic applications. The environment generates visually realistic scenarios that closely resemble real-world underwater conditions, serving as a valuable tool for research and development in robotics. The second contribution is the adaptation of ORB-SLAM3 for underwater environments using visual-pressure and visual-inertial-pressure configurations. These configurations demonstrate improved performance in the underwater domain through modifications to the visual component, pose estimation, local bundle adjustment algorithms, new vocabularies, and various initialization procedures.

While both the simulation environment and the modified ORB-SLAM configurations show promising results, there remain several challenges and opportunities for future work. For the simulation environment, usability and sensor data synchronization are areas for improvement. Expanding the evaluation, implementing new environments, additional sensors, and a GUI would further enhance the simulation's value. For the adapted ORB-SLAM configurations, refining scale estimation, addressing trajectory drift, and improving loop closure capabilities are crucial next steps. More tests in different environments and potential applications of deep learning techniques can also enhance the system's performance.

By making the simulation and SLAM system implementations openly available online, this thesis aims to increase accessibility to underwater robotics for the public, fostering continued development and innovation in the field.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Our oceans, covering more than 70% of the Earth's surface, represent a vast and largely unexplored frontier. In recent years, advances in underwater technology have enabled us to better understand and monitor the underwater world, allowing us to manage and utilize ocean resources more responsibly. Underwater robotics, including Remotely Operated Vehicles (ROVs) and Autonomous Underwater Vehicles (AUVs), have played a crucial role in this progress, allowing us to explore and conduct research in places that were once inaccessible or too dangerous for human divers.

1.1.1 Underwater robots

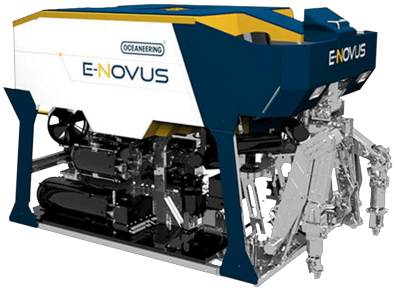
ROVs and AUVs are used to survey seabeds, monitor the environment, perform maintenance on man-made structures and perform other similar tasks. They are also better designed to navigate difficult environments with no risk to human safety. ROVs are versatile robots tethered to a topside unit and can be tele-operated. AUVs on the other hand operate autonomously using onboard sensors and run on their own power sources. Both classes of underwater robots can be configured in various ways to suit the specific tasks they are designed to accomplish.

ROVs are more commonly found due to their tele-operability, making them more versatile towards a wider variety of tasks. They will typically fall into one of these classes:

- Observation class: Small-sized ROVs mainly for exploration and observation of shallower waters such as the coast, rivers, and small lakes.
- Light work class: Small to medium-sized ROVs able to carry light payloads and manipulate small objects. Can be used in medium to deep waters.
- Heavy work class: Large-sized ROVs able to carry heavy payloads and several tools for manipulation. Can operate in the deepest depths and reach the ocean floor.

(a) AUV: Slocum glider (Teledyne Marine, 2023)

(b) Observer class ROV: BlueROV2 (BlueRobotics, 2023)



(c) Light work class ROV: eNovus (Oceaneering, 2023)

(d) Heavy work class ROV: Schilling UHD III (ROVOP, 2023)

Figure 1.1: Examples of underwater robots

Observation class and light work class ROVs are easier to find and acquire commercially, whereas heavy work class ROVs are typically only found within oil and gas industries. The relatively low cost of smaller ROVs have enabled smaller

companies, research groups and even private individuals to acquire these robots and perform their own experiments.

1.1.2 Underwater localization

Localization is crucial in robotics for proper navigation within the environment. Underwater, localization is usually achieved using acoustic positioning systems (APS) or by dead reckoning (DR). Acoustic positioning systems use beacons spaced away from each other to trilaterate the position of the robot, which will typically have a transponder mounted. Dead reckoning involves the estimation of the robot's state using onboard sensors to perform localization. The following section discusses the most commonly used instrumentation for underwater localization. Most of the information here originates from (Paull et al., 2014). We highly advise interested readers to check the authors' work as it contains more information regarding this topic.

1.1.3 Instrumentation for underwater localization

Global and Acoustic Positioning Systems

Global Positioning Systems can estimate the position of a vessel by using the time-of-flight signals from visible satellites. GPS signals however cannot penetrate deep into the water. Acoustic positioning systems function as the underwater equivalent to these and use a similar working principle. Typically, an acoustic position system will make use of at least 3 beacons to trilaterate the position of a robot, where a transponder is typically mounted on the robot.

Acoustic positioning systems can generally be split into 3 categories; Long Baseline (LBL), Short Baseline (SBL), and Ultra Short Baseline (USBL). LBL systems typically install fixed beacons that are widely spaced from one another. The position of these beacons are usually known during operation. SBL and USBL systems function similarly to LBLs by measuring the time of flight and phase shift between measurements. As their name implies, they have a much shorter baseline between beacons, with USBLs having the shortest baselines. The shorter the baseline is, the lower the accuracy, but it is also cheaper to set up compared to LBLs and can be mounted on a mobile platform such as a ship.

Proprioceptive sensors

Proprioceptive sensors provide measurements regarding the internal state of the robot and can be used for localization. Here, we refer to proprioceptive sensors as the instrumentation onboard the robot. These types of sensors do not gather information regarding the surrounding environment. Proprioceptive sensors by themselves are sufficient in providing localization estimates, but suffer from unbounded drift the longer they operate.

Pressure sensor Pressure sensors measure the amount of pressure applied by the water column above it, which scale linearly as depth increases. The pressure gradient in water is steeper than in air, which results in high accuracy in the measurements. In practical applications we are often more interested in knowing the depth measurement, which can easily be computed due to its linear relationship with pressure.

Inertial measurement unit Inertial measurement units (IMUs) are typically made up of an accelerometer and a gyroscope. Accelerometers measure the linear acceleration of the system while gyroscopes measure the angular velocity. These measurements can be used to calculate the position, speed, and orientation of an object in motion. IMUs are subject to drift over a period of time and sensor noise. To recover the position estimate, it is necessary to integrate the accelerometer twice. The noise and drift will therefore lead to a rapid accumulation of error in the estimate. They typically have to be paired with other sensors to compensate for this.

Compass Some IMUs also have a magnetometer integrated as part of the package. Magnetometers measures the magnetic field of the environment and can be used to estimate the global heading of the system, ultimately functioning as a compass. However, they are very susceptible to magnetic disturbances that may be present in the environment.

Doppler velocity log Doppler Velocity Logs are sensors, usually with 3 or 4 beams, that emit acoustic pulses towards a static object, such as the seabed. By measuring the wave's shift in frequency, the sensor is able to determine its velocity.

Exteroceptive sensors

Unlike proprioceptive sensors, exteroceptive sensors measure the state of the environment, which can for example be used for mapping. LiDAR, sonar, and cameras are the most commonly used to do this, although LiDAR is used to a more limited extent underwater. Range scan sensors like LiDAR and sonar can directly measure the geometry of the environment.

Sonar Sonars use sound to detect and locate objects underwater. In the context of localization and mapping, we generally refer to the use of active sonars to produce an image of the surrounding environment. Active sonars emit sound pulses and measure the echoes to detect and locate objects in the environment. This is in contrast to passive sonars which only listen to the sound in the environment. These sensors come in many forms, such as forward-looking sonars, side-scan sonars and many more, but all operate under the same principle. Compared to cameras, sonars are not affected by water turbidity. However, they generally have lower acquisition rates and produce less detailed images.

Camera Cameras function by allowing light to pass through an optical lens which can then be captured by a light-sensitive surface, thus generating an image. They come in a variety of configurations and specifications to best tackle a specific task. Image quality often degrades significantly underwater, due to factors such as poor lighting conditions and turbidity, resulting in limited visibility range.

1.1.4 Simultaneous localization and mapping

Acoustic positioning systems tend to be expensive to deploy and operate. They also have a limited range, only functioning when the robot is within the area covered by the beacons. Mobile solutions exist that involve mounting the beacons on support vessels, but this also increases the costs of deployment and operation. Dead reckoning solutions are able to operate using only the robot's instrumentation and are therefore cheaper. However, these solutions suffer from unbounded growth in their estimation error due to uncertainties from measurement noise and biases.

Simultaneous Localization and Mapping (SLAM) offers an alternative to dead reckoning that limits the growth in uncertainties while maintaining independence to the use of external instrumentation. SLAM algorithms estimate the robot's pose in a manner similar to dead reckoning solutions. At the same time, it constructs a map of its surroundings. If the system is able to recognize a previously explored location, it can correct the estimated trajectory and any drift caused by measurement uncertainties. This process is known as loop-closure. A lot of attention has been directed towards the development of SLAM as it has been considered a keystone for enabling fully autonomous navigation.

SLAM formulation

SLAM is typically formulated using two models: the motion model, which describes the state of the robot, and the observation model, which describes the environment. The motion model can be estimated using the robot's proprioceptive sensors such as an IMU or by observing known landmarks and calculating its position relative to these landmarks. Conversely, the observation model can be constructed using the robot's exteroceptive sensors such as cameras or raw range scan sensors like LiDAR and sonar. A map of the robot's surroundings can be gradually built up, provided that the same landmarks can be re-observed. Both models are updated iteratively and also use the output of the other as input. Each model introduces uncertainties to the system due to sensor noise and inaccuracies in the model estimations. Since neither the motion nor the observations are actually known and are instead calculated with a certain probability, SLAM is probabilistic in nature and is often described as a probability distribution.

$$P(x_k, m | Z_{0:k}, U_{0:k}, x_0) \tag{1.1}$$

Equation 1.1 describes the probability distribution of the state of the robot x_k computed at time k and the map m given the previous observations $Z_{0:k}$, input controls $U_{0:k}$, and the initial state of the vehicle x_0 (Bailey & Durrant-Whyte, 2006). Figure 1.2 provides a graphical representation of the problem.

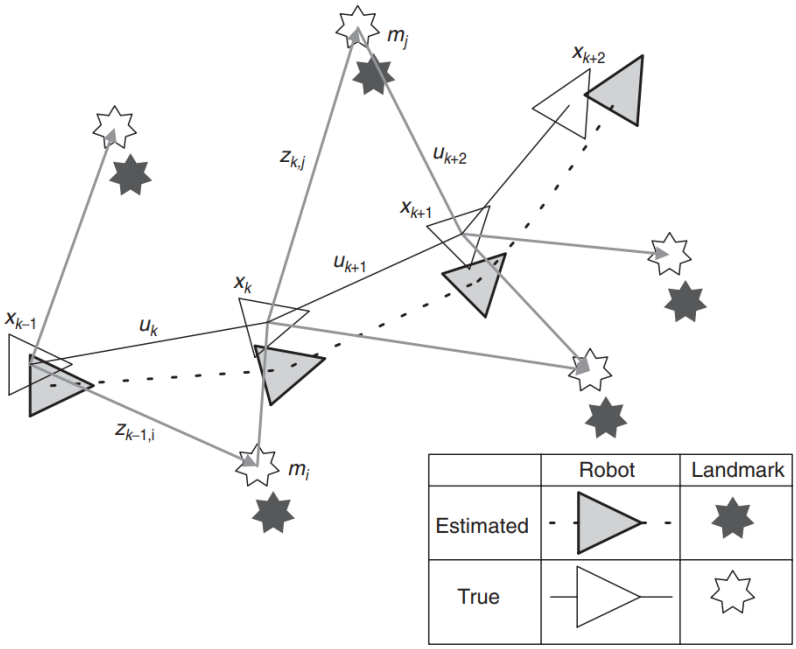


Figure 1.2: Graphical representation of the SLAM problem. Source (Bailey & Durrant-Whyte, 2006)

In this context, x_k represents the state vector of the robot, u_k is the control vector applied at time $k-1$ to move the robot to the state x_k, m_i is a vector containing information about the i^{th} landmark, and z_{ik} is the observation of the i^{th} landmark at time k. Many if not all SLAM algorithms in the literature are based on a probabilistic Bayesian approach.

Visual SLAM

Cameras have no way of directly measuring depth by themselves. Instead, camera-based SLAM, more commonly referred to as Visual SLAM (VSLAM), rely on structure from motion (SfM) to recover depth from the images. However, monocular cameras still cannot properly recover scale, which must be arbitrarily set during initialization. Cameras would have to be either paired up to produce stereoscopic vision or combined with additional sensors such as range sensors (e.g., RGBD cameras) to recover depth and consequently the scale of the environment. Despite this, a large body of literature has been released dedicated to solving the SLAM problem using normal monocular cameras. The main motivation is that monocular configurations are considerably

cheaper and easier to work with.

Many works have also been done to fuse cheaper sensors that are often already integrated within the robot, such as IMUs. These sensors also allow for the scale to be recovered. Many state-of-the-art systems have demonstrated that these low-cost configurations can reliably and accurately track the robot's trajectory and map the environment.

Recently, underwater SLAM has also seen this type of development and a desire for low-cost configurations. We note that most of the literature of underwater SLAM mainly use sonar, but these sensors tend to be expensive and difficult to work with in SLAM due to the low acquisition rate and resolution. Cameras have been investigated as an alternative due to their ability to provide more information at a much higher acquisition rate. Most ROVs also come configured with a camera.

However, cameras have some clear disadvantages underwater, including poor lighting conditions, low contrast, and backscattering. Many underwater environments also have repetitive features, and can make it difficult to differentiate one location from another. A study has nonetheless been conducted testing various state-of-the-art visual SLAM algorithms in underwater environments (Hidalgo Herencia, 2019). Some algorithms perform surprisingly well in structured environments like harbors but need to stay close to these structures or the seabed to maintain tracking. It is important to note that these algorithms were originally designed for aerial and terrestrial environments.

Building on the idea of using low-cost sensors for SLAM, we also consider incorporating depth measurements from pressure sensors to further enhance localization. Many low-cost ROVs have a camera, IMU and pressure sensor as part of their onboard instrumentation. This approach makes the SLAM configuration suitable for a wide range of ROV configurations.

1.1.5 Simulation environment for Visual SLAM

To facilitate the development and evaluation of the underwater visual SLAM algorithm, a simulation environment will be created. This environment will serve as a platform for testing the algorithm in various underwater scenarios, providing a controlled and efficient approach to fine-tuning and analyzing the algorithm's

performance. By using the simulation environment, we can more easily replicate conditions and scenarios that would be challenging to prepare and test in the real world. The simulation environment will be a key component of the algorithm's development process and will be discussed in detail in a dedicated chapter of this thesis.

1.2 Problem statement

In this thesis, the aim is to develop a visual SLAM algorithm using a camera, IMU, and pressure sensor for the underwater domain, and to create a simulation environment to facilitate the development and evaluation of the proposed algorithm. We can split the objectives into the following:

- Develop and evaluate a simulation environment for underwater visual SLAM algorithm testing and validation
- Improve the visual pipeline of the SLAM algorithm to enhance camera performance underwater
- Implement a visual-pressure SLAM configuration
- Implement a visual-inertial-pressure SLAM configuration
- Investigate loop closure capabilities in the underwater domain

The system and its various configurations will be tested on open-source datasets as well as custom datasets acquired from live testing. Each configuration should be able to run real-time with minimal configuration needed from the operators.

1.3 Scope and limitations

We will use an existing, open-source SLAM algorithm as our base configuration and implement our own changes to it. Designing the system from scratch was considered but was later changed to better fit the scope and timeframe of the project. The system will mostly be evaluated on structured underwater environments.

1.4 Ethical Considerations

The project is mostly the development of software. Regardless, there are some ethical implications of the technology both directly and indirectly. The following are some ethical considerations associated with the project.

Environmental impact Underwater robots have the potential to disturb natural habitats and marine life during their operation. It is crucial to ensure that real-life tests and experiments related to underwater SLAM minimize any negative impact on the environment of ecosystems.

Privacy and data security The collection of data, especially images and sensor readings, during the testing and development of underwater SLAM algorithms raises concerns about privacy and data security. While it is unlikely that any sensitive data is gathered in the project, it is still important to handle all collected data responsibly, ensuring that it is stored securely and not used for purposes other than those intended in the project.

Dual-use technology While the primary goal of this thesis is to advance the field of underwater robotics for scientific, exploration, and maintenance purposes, it is essential to be aware that the technology developed could be used for other purposes, such as military or surveillance applications.

Accessibility and fairness As technology continues to develop, it is important to ensure that the benefits of these advancements are accessible to a wide range of users and applications. By focusing on low-cost sensor configurations, this project aims to make underwater SLAM more accessible and affordable to researchers and organizations with limited resources. We also make all the software developed in this project open-source and available online for the benefit of the community.

Chapter 2

Underwater Simulation Environment for Visual SLAM

2.1 Introduction

Simulations play a key role in the development and testing of robotic systems and have recently become increasingly important for training machine learning algorithms designed for robotics. This allows researchers to test different systems and algorithms without the need for expensive and time-consuming real-world deployments, in turn allowing for a more efficient development cycle. Underwater environments, in particular, pose unique challenges such as complex hydrodynamics, limited visibility, and communication constraints, making simulations even more crucial for the development of underwater robots. In this chapter, we present the development of an underwater simulation environment that can be used for visual SLAM algorithms for underwater robots. To achieve this, we integrate Unreal Engine and Gazebo.

The main objective of the simulation is to provide a framework to let other researchers build upon it, extending its capabilities to support the dynamic elements of the underwater domain. We will evaluate the simulation by creating a dataset that can be used for the development and verification of underwater visual SLAM algorithms. The visual component is the most crucial aspect of these systems, which is why we leverage the photorealistic rendering capabilities of Unreal Engine. We use it together with Gazebo, as it provides a more accurate physics-based simulation of the underwater environment.

2.2 Related works

2.2.1 Simulation platforms

Several simulation platforms have been used by the robotics community. We discuss some of the most popular platforms in the community as well as some notable implementations.

Gazebo

Gazebo is a widely-used, open-source simulator that provides realistic rendering and physics-based dynamics. Gazebo has been used for aerial, terrestrial, and underwater robot and supports a wide range of sensors and actuators (Koenig & Howard, 2004; Open Robotics, 2021). Gazebo integrates seamlessly with ROS and is part of the reason for its widespread popularity in the robotics community. Several packages also exist that further extend its capabilities, allowing for more advanced simulations.

There are numerous implementations of robotics simulators for Gazebo. One of the most well-known practical ground-based simulators is the TurtleBot simulator, presented by Open Robotics themselves (Open Robotics, 2023). The turtleBot is a low-cost, personalized robot kit often used for education and research. The simulator replicates the robot's sensors and actuators that allow for users to develop and test various robotics systems such as SLAM, navigation, object manipulation and more. For aerial simulations, two examples include the works of (Furrer et al., 2016; Meyer et al., 2012). In the first paper, the authors present a simulation framework for micro-aerial vehicles (MAVs), allowing for the development and testing of higher level tasks such as path planning, obstacle avoidance, and SLAM. The second paper presents a similar framework, but focuses on quadrotor unmanned aerial vehicles (UAVs).

Unity

Unity is a powerful game engine increasingly used for robotics simulations due to its high-quality rendering capabilities and support for various sensors and physics models. The Unity developers have also acknowledged the increased interest in its use as a robotics simulation platform and have released the Unity Robotics Hub. The

Robotics Hub is a collection of tools, tutorials, and resources for robotics simulations in Unity (Unity Technologies, 2022), and enables ROS integration in the engine.

The most well known robotics implementation using Unity is Isaac Sim developed by NVIDIA (NVIDIA Corporation, 2023b). It offers realistic rendering, physics, and perception capabilities for various robotic platforms, including mobile robots, manipulators, and drones. It has been designed to function seamlessly with NVIDIA's Isaac SDK, an SDK for developing AI-powered robotic applications (NVIDIA Corporation, 2023a).

Unreal Engine

Unreal Engine is another popular game engine that offers photorealistic rendering and physics-based dynamics. Unreal Engine has also seen use in simulating robots and vehicles. As of the time of writing, no official tools have been released that provide an easy way of integrating the engine with ROS, but third-party plugins have been developed to accomplish this.

Two notable simulators use Unreal Engine, namely Microsoft's AirSim and CARLA. AirSim is an open-source simulator designed for the development of autonomous systems for aerial vehicles (Shah et al., 2018). CARLA (Car Learning to Act) is used primarily for research in autonomous driving (Dosovitskiy et al., 2017). Both of these simulators offer high fidelity in visual and behavioral realism and support a large set of sensor modalities. They also support ROS integration.

2.2.2 Underwater simulators

One of the earliest underwater simulators to gain traction is UWSim (Prats et al., 2012), which is built on top of OpenSceneGraph and integrated with ROS. The simulator supports the addition of various scene elements as well as multiple underwater vehicles and manipulators and also offer support for multiple sensor modalities. The simulator is somewhat limited because it cannot be easily extended with custom sensors and interfaces. Additionally, setting up a new simulation is a time consuming task, as everything needs to be defined in a single XML description file (Manhães et al., 2016).

UUV Simulator introduces an underwater vehicle simulator based on ROS and

Gazebo (Manhães et al., 2016). It contains a set of plugins for simulating underwater dynamics and sensor models in Gazebo. Perhaps one of the most attractive features of this simulator is its support for Fossen's equations of motion for marine vehicles (Fossen, 2011). This set of equations is often used in modelling the hydrodynamics of marine vehicles and its inclusion is therefore an attractive feature to those familiar with the field. As of the time of writing, this simulator has not received any updates since June 2020. Project Dave can be thought of as an unofficial successor to UUV Simulator, further extending its feature set with support for a greater variety of vehicles and manipulator configurations, better simulations of underwater specific sensors, and parameterized representations of the ocean environment (Field Robotics Lab, 2022). Project Dave has also been developed to support the latest release of ROS1, ROS Noetic. The greatest limitation of these simulators is the visual fidelity which often do not reflect how underwater environments look in the real-world.

Underwater simulations have also been developed using Unity. The first example of this is presented by (Katara et al., 2019). Here, the authors use Unity for rendering and modelling the physics of an underwater environment as well as certain sensor modalities. It communicates with ROS using ROSBridge and allows the simulator to interface with other systems (e.g., path planning algorithms). (Chaudhary et al., 2021) present a similar framework, where Unity is also responsible for rendering the environment and simulating the hydrodynamics. Additionally, they provide support for multibeam SONAR. Both simulators offer higher visual fidelity than those previously mentioned. However, they do not provide a "universal" parameterization of the vehicle's hydrodynamic properties such as presented in UUV simulator.

2.2.3 Summary and project direction

To address the challenges associated with underwater simulations, our project aims to combine the strengths of both Gazebo and Unreal Engine. We use Gazebo for its accurate physics-based simulations and ROS integration, with the main attraction being Project Dave and its implementation of Fossen's equations. We then pair this with Unreal Engine for its photorealistic rendering. This combination enables the development of a more realistic and versatile underwater simulation environment.

2.3 Methodology

The simulation comprises of two primary components: the back-end and the front-end. The back-end, running on Ubuntu Linux, leverages Gazebo for physics simulation and generating measurements for proprioceptive sensors attached to the robot. The front-end, running on Windows, utilizes Unreal Engine to enable direct user control and display the camera feed from the robot's perspective. The simulation was earlier designed to run on two separate PCs, with the front-end on one machine and the back-end on another. The newest iteration allows the user to run the entire setup on a single PC, but requires WSL2 to run Ubuntu. Figure 2.1 shows the system architecture of the simulation. We first present the hardware and software configuration used in developing the simulation. Next, we discuss the implementation of the back-end, followed by the implementation of the front-end. Finally, we describe the practical implementation of a simulation environment and create a dataset that can be used for the development and evaluation of visual SLAM.

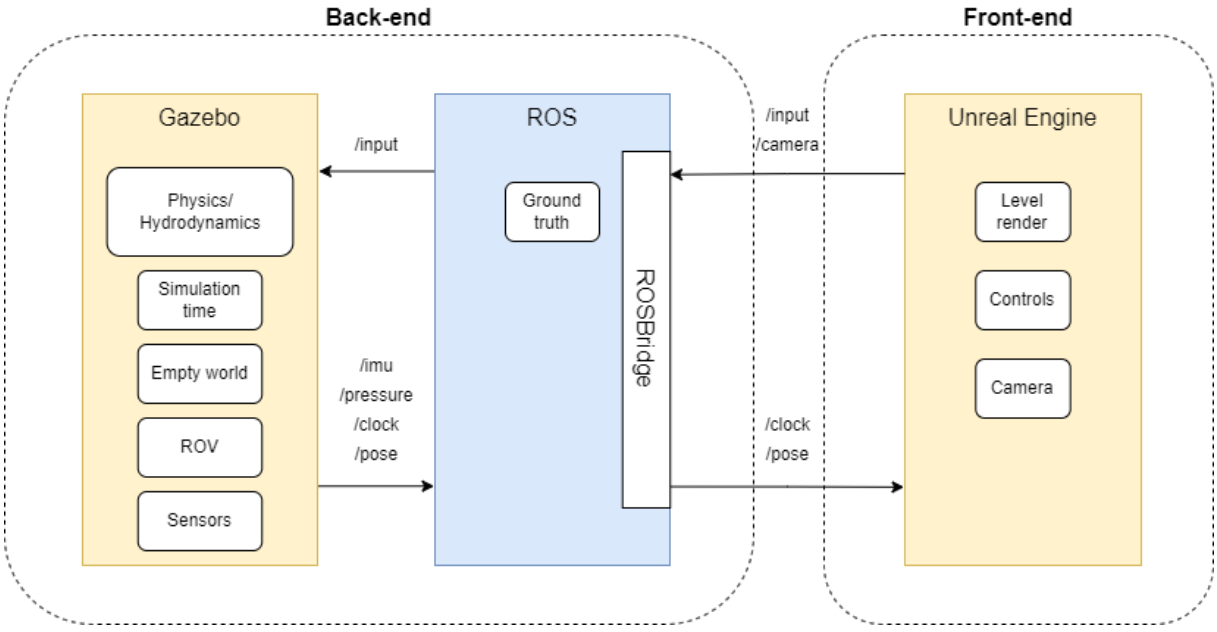


Figure 2.1: Simulation overview

2.3.1 Hardware and software configuration

The simulator has been tested and developed on two main PC configurations as shown in Table 2.1.

Component	PC1 (stationary)	PC2 (laptop)
CPU	AMD Ryzen 2600x	Intel i5-8300H
RAM	16 GB	16 GB
GPU	Nvidia Gtx 1060 6 GB	Nvidia Gtx 1050Ti 4 GB

Table 2.1: PC specifications

The stationary PC (PC1) is primarily used for development due to the simulator's higher computational demands. The laptop (PC2) is used to run the front-end and back-end on separate machines, with the front-end on PC1 and the back-end on PC2.

The software specifications for the back-end and front-end are shown in Tables 2.2 and 2.3, respectively.

Software	Version
Ubuntu	20.04
ROS	Noetic
Project Dave	4.3.1
Gazebo	11.12
ROSBridge	0.11.16

Table 2.2: Back-end software specification

Software	Version
Windows 11	22H2
Unreal engine	4.27
Visual studio	Community 2019
ROSIntegration	custom

Table 2.3: Front-end software specification

2.3.2 Back-end

The primary component of the back-end is Gazebo. Gazebo offers tighter integration with ROS, making it easier to translate the code to the real-world counterpart of

whatever is being simulated. We extend Gazebo with the Project Dave package by adding plugins and ROS applications to simulate underwater robots. Project Dave is chosen over UUV Simulator as it has better compatibility with ROS Noetic. Keep in mind however that we mostly use the features of the original UUV Simulator.

Level setup

We set Gazebo up to open an empty underwater world in which we run our simulation. Ideally, the Gazebo world would match the level used in Unreal engine such that we can also model collisions and interactions with the environment. However, there is no easy way to translate the levels between the two engines. Since everything we see is in Unreal engine, we only model the level within there. This remains an open topic left for future work.

With the world ready, we can spawn in the desired vehicle to operate in the environment. The vehicles that come with Project Dave have been designed to receive either a twist or wrench vector as input used for motion. These vectors are translated into the the thrust forces f_t for each individual thruster according to the vehicle's thrust allocation matrix (TAM). The back-end itself does not generate the input messages. Rather, it waits for the input from the front-end which are further sent to Gazebo.

Gazebo then calculates the necessary parameters for motion to occur. We call a service request to Gazebo to fetch the pose of the robot. This needs to be converted into a pose message, which we then send to the front-end.

The measurements coming from the sensors are directly published to ROS. The topics from Gazebo relevant to the project are shown below. Topics above the line are what Gazebo subscribes to, while topics below the line are what Gazebo publishes. To ensure that both systems are synchronized as closely as possible, we also publish the simulation time used in Gazebo (also referred to as the clock).

```
geometry_msgs/Twist         /cmd_vel
---
rosgraph_msgs/Clock         /clock
geometry_msgs/Pose          /x3/pose_raw
sensor_msgs/Imu             /x3/imu
sensor_msgs/FluidPressure   /x3/pressure
```

Ground truth

The same transform message that we send to the front-end is also used for generating the ground truth of the robot. We perform further calculations to the pose such that it is viewed relative to the initial position of the vehicle. We can view this in Rviz or similar software to visualize the robot's pose. In addition, we use a ROS library, hector-trajectory-server, that keeps track of the entire trajectory. We use this package for visualizing both the ground truth as well as the estimated trajectory generated by the SLAM system.

ROSBridge

Communication is established using ROSbridge. ROSbridge provides an API to ROS functionality for non-ROS programs. We establish a ROSbridge server that opens up the back-end to the front-end and enables two-way communication using TCP. We enable BSON in ROSBridge. BSON is a binary encoded version of the popular JSON (Javascript Object Notation) format and is more efficient as it uses less space. It also improves performance by reducing the overhead in parsing and encoding the data.

Sequence of operations

The back-end involves several steps that are not as straightforward to the unfamiliar user. Therefore, we summarize here the procedure to set up and run the back-end of the simulation.

1. Launch ROSBridge server with BSON enabled.

2. Open an empty underwater world in Gazebo.

3. Upload desired underwater vehicle to Gazebo.

4. Run thrust allocator converter for the vehicle.

5. Run python node to transform Gazebo pose message to a transform message.

2.3.3 Front-end

The front-end of the simulation is built using Unreal Engine, which is responsible for providing the user with direct control and displaying the camera feed from the robot's perspective. Unreal Engine is chosen for its high-quality rendering capabilities, making the simulation visually more realistic and better suited for testing visual SLAM algorithms.

ROSIntegration

We use the ROSIntegration Plugin for Unreal Engine to support ROS features (code-iai, 2023). We are using a custom version of this plugin to fix some of the issues that we encountered, which we have included in the appendix. The main issue was an error in the "camera info" message topic, which stopped the message from being published to ROS.

The connection itself is done using the ROSBridge suite as described in Section 2.3.2. ROSIntegration only supports Unreal Engine 4 and is the main reason for choosing Unreal Engine 4 over the newer and more photorealistic Unreal Engine 5.

In the engine itself, we need to create a custom game instance provided by the plugin, specifically a ROSIntegrationGameInstance. This instance handles communication with ROS and also allows for the engine to publish the simulation time. In our case, we use the clock published by Gazebo in the back-end.

ROS pawn

We created a custom pawn using the base Pawn class in Unreal Engine. Any sensors the user wishes to model can be attached to this class. The ROS Pawn is subscribed to a transform topic that is used to update the pose and orientation of the robot within Unreal Engine. This ensures that the visual representation in Unreal Engine is consistent with the simulated physics in Gazebo. It is also subscribed to the clock such that any component attached to the pawn can use it as a reference if necessary. All topics the pawn subscribes and publishes to are shown below.

```
geometry_msgs/Pose          /x3/pose_raw
rosgraph_msgs/Clock         /clock
---
```

```
geometry_msgs/Twist          /cmd_vel
sensor_msgs/Image            /unreal_ros/image_color
```

Control input The ROS Pawn contains a component that provides a control interface for the end user. The input commands are set up in Unreal Engine's settings, and each input key is converted to the appropriate value in a twist vector. We have implemented a basic control scheme which can be seen in Figure 2.2. The mouse can also be used to rotate around the pawn's yaw.

Figure 2.2: Control scheme

Camera For each sensor attached to the ROS Pawn, we create a corresponding ROS publisher. The sensor data generated in the Unreal Engine simulation is converted into the appropriate ROS message type and published on a dedicated topic. We use the subscribed clock topic when creating timestamps.

In our case, we only implement a monocular camera attached to the ROS pawn that captures the images from the perspective of the robot. These images are converted into ROS image messages and published together with the camera information. To minimize latency and maximize performance, we limit the images to their grayscale equivalent using OpenCV's RGB-to-grayscale conversion (OpenCV, 2023). We can see this in Equation 2.1:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{2.1}$$

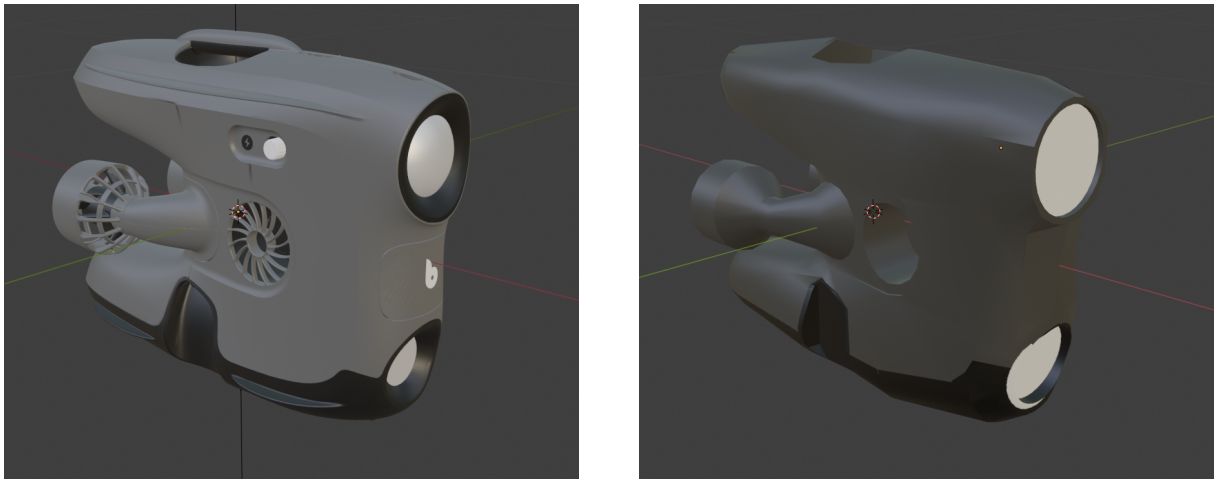
where Y is the grayscale value, and R, G, B are the red, green, and blue pixel intensities respectively for the given pixel.

2.3.4 Simulation environment

This section discusses the practical implementation of the simulation environment, including the ROV and level used for analysis.

ROV implementation

We implement a custom ROV based on Blueye Robotics' X3 ROV. The hydrodynamic parameters of the ROV are not readily available, so the values used here are only approximations. We use the estimated model parameters of its predecessor, the Blueye Pioneer, presented by (Bellingmo, 2020) as our baseline. However, some values are missing, notably values related to the roll and pitch axes, as they were not relevant to the authors. We derived these values through empirical testing and by looking at the URDF descriptions of the vehicles included in UUV Simulator.



(a) Official render of the X3 ROV

(b) Low poly version

Figure 2.3: X3 ROV. Gazebo uses the low-poly version to reduce computational load

The vehicle description is provided in a standard ROS URDF format using plugins provided by UUV Simulator to model hydrodynamics, actuators, and sensors. The sensor suite includes a camera, a pressure sensor, and an IMU. Their specifications can be found in Table 2.4. The camera is the only sensor implemented in Unreal Engine, while the rest are modeled in Gazebo. We highly recommend interested readers to read the original publication of (Fossen, 2011) for detailed information regarding hydrodynamic parameters. A detailed description of the ROV is provided in Appendix Section A.2 for those interested in specific implementations. Here, we only provide the

sensor specifications. These values do not reflect the actual sensor specifications of the X3. The values have changed during the project as part of the testing and development process, and the values shown in the table represent the latest iteration.

Camera	
Resolution	960×540 px
Frames per second	20 fps
Field of view	90°
Pressure sensor	
Frequency	10 Hz
Noise	$1 \times 10^{-3}\ m$
Inertial measurement unit	
Gyroscope frequency	200 Hz
Gyroscope noise	$3.08 \times 10^{-5}\ rad/s$
Gyroscope bias	$1.7 \times 10^{-2}\ rad/s$
Accelerometer frequency	200 Hz
Accelerometer noise	$3.08 \times 10^{-2}\ m/s^2$
Accelerometer bias	$6.8 \times 10^{-6}\ m/s^2$

Table 2.4: Sensor specifications

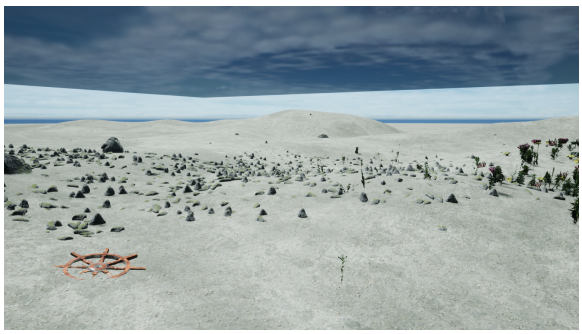
Scene and level design

In Unreal Engine, a level is a self-contained environment in which the user can interact with objects and simulate physics. For our underwater simulation, we create a custom level that represents the desired underwater environment.

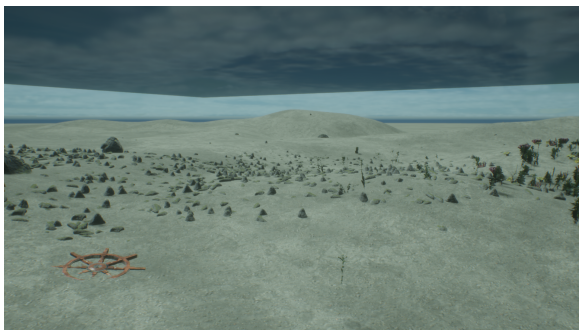
The level presented here has been designed with consideration for the specific task of visual SLAM. We have therefore included various environmental features, such as rocks, seaweed, corals, and a few man-made debris. We have split these into distinct areas in the level to accommodate several typical environments. Additionally, we have added particle effects in the water to provide a more dynamic scene.

We have also changed the lighting and post-processing effects to simulate different underwater conditions. We can see how they affect the world in Figure 2.4. The

lighting and post-processing techniques used are key for replicating an underwater environment as accurately as possible.



(a) Base landscape



(b) Post processing enabled

(c) Post processing + exponential height fog

Figure 2.4: UE effects to simulate an underwater environment

Dataset creation

Two sequences are created for the dataset. We use the same level but model different circumstances. Both sequences are recorded using PC1 with a 1-PC configuration and the back-end running on WSL2. The data is recorded into rosbags with the following topics:

`rosgraph_msgs/Clock`	`/clock`
`geometry_msgs/Pose`	`/x3/pose_raw`
`geometry_msgs/Twist`	`/cmd_vel`
`sensor_msgs/Imu`	`/x3/imu`
`sensor_msgs/FluidPressure`	`/x3/pressure`
`sensor_msgs/Image`	`/unreal_ros/image_color`

The 1-PC configuration offers less latency compared to running the front-end and back-end on separate machines. We further process the rosbags to better synchronize

the sensor measurements, namely the image feed from Unreal Engine. We replace the timestamps of the images to instead use the time at which they were published to ROS. We further offset the time by a set value to account for the time from when the image was generated to when it gets published to ROS. We discuss this further in the results section.

Sequence 1 Sequence 1 simulates a situation with greater turbidity and, thus, a lower range of vision. The overall environment is darker and offers less contrast. Outside of moving particles and seagrass, the environment is relatively static. The total length of the trajectory is 155 meters, with a duration of 200 seconds. The trajectory starts and ends within the same location and in view of an Apriltag placed in the level. Apriltags are usually used to calibrate cameras, but in this case it simply serves as a landmark. The motion of the ROV is relatively close to the seafloor to provide the best opportunity for identifying features in the environment. The start contains motion along the depth-aligned axis, which should make it suitable for initialization procedures using depth sensors.

Sequence 2 Sequence 2 features less turbidity and more illumination. However, this causes the particles in the water to be more visible. Additionally, we add caustics to the level. This results in a highly dynamic and challenging environment for visual SLAM algorithms. The trajectory has a total length of 120 meters and a duration of 157 seconds. Similar to the previous environment, the trajectory starts and ends near the same area with an observable Apriltag for reference. We keep the ROV motion close to the seafloor, as in the previous environment, and also introduce motion along the depth-aligned axis during the start of the trajectory.

Sequence	**Length**	**Duration**	**Comments**
Sequence 1	155 m	200 s	Low visibililty, relatively static
Sequence 2	120 m	157 s	Higher visibility and caustics, highly dynamic

Table 2.5: Details on underwater sequences

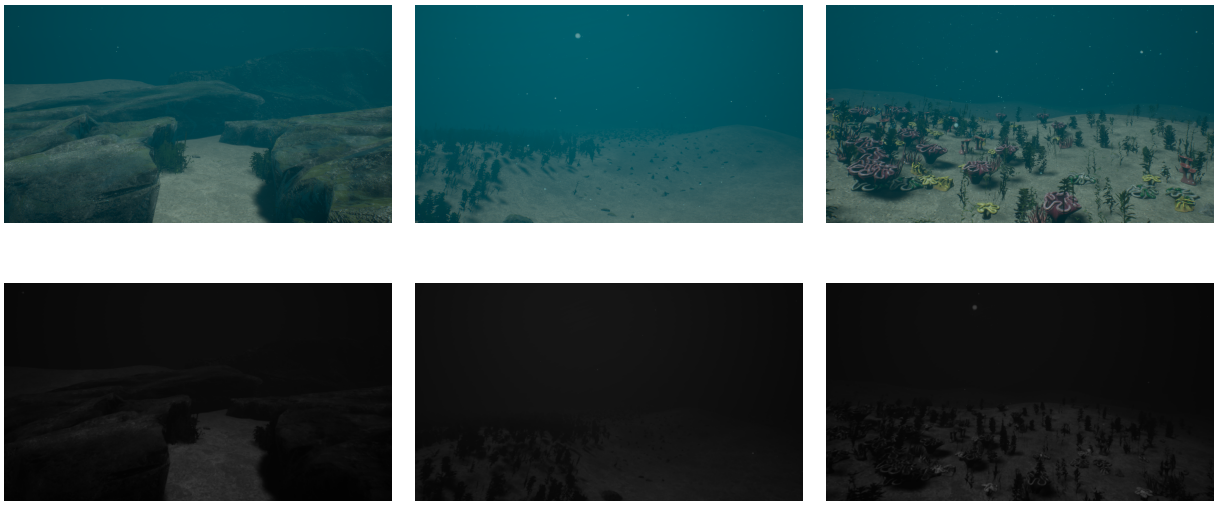


Figure 2.5: Images from sequence 1. Top row: RGB, bottom row: grayscale

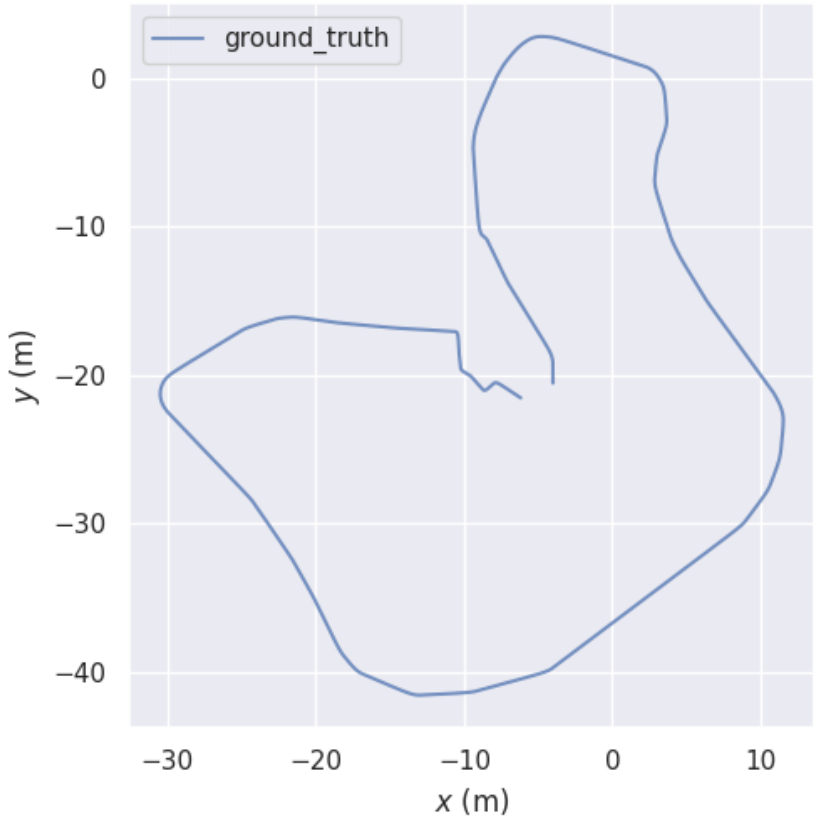


Figure 2.6: Sequence 1 trajectory

Figure 2.7: Images from sequence 2. Top row: RGB, bottom row: grayscale

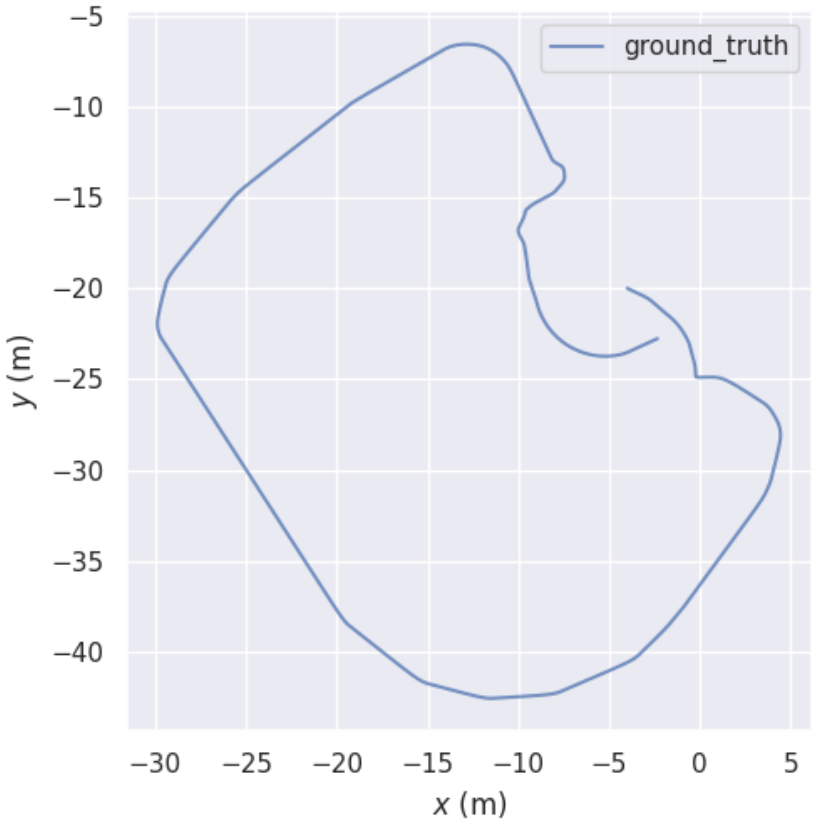


Figure 2.8: Sequence 2 trajectory

2.4 Results

In this section, we present the results of the evaluation of the underwater simulation implementation. Our focus is on assessing the performance, user experience, and usability of the simulation in the two created environments, as well as discussing any limitations and potential future work. Most of the results here are qualitative in nature. The evaluation is conducted mainly for the 1-PC configuration unless otherwise specified, with the back-end running on WSL2.

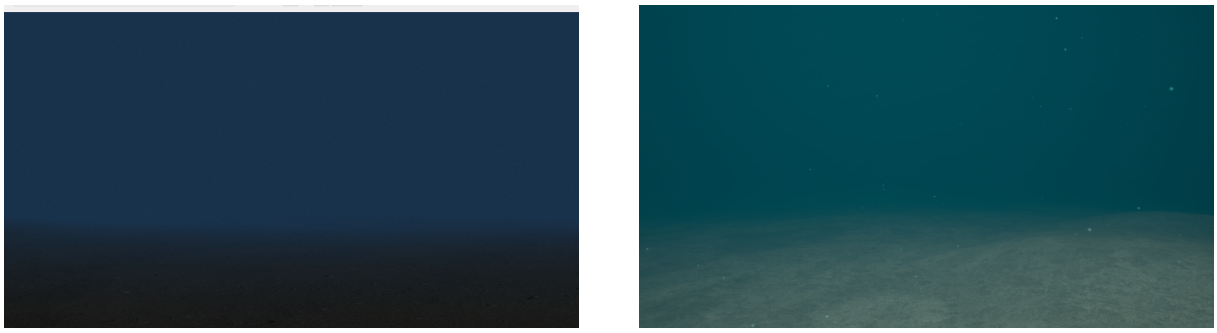
2.4.1 Simulation performance

Simulation performance is evaluated by looking at each sequence. First, we examine how accurately the environment represents an actual scenario in real-life. This mostly comes down to visual inspection of the camera feed. Next, we examine the quality of the sensor data and if they accurately model their real-life equivalent. We then discuss the user experience for operating the simulation. Finally, we look at some of the technical aspects of the sequence such as the framerate and any latency issues.

Sequence 1

In Sequence 1, the environment sufficiently represents typical underwater areas. Located within a relatively small level, it is easy to test the performance of visual SLAM algorithms against these common scenarios without reloading the simulation. The visuals are enough to emulate the underwater domain, and somewhat ironically, the low-resolution camera enhances the environment's realism. It can however be argued that the visuals are too dark. Figure 2.9 shows a significant improvement in visuals compared to Gazebo. The sensor data generated in this environment provides a good representation of the challenging conditions, with realistic noise and degradation in the camera feed.

The control inputs from the user are responsive and accurately translates into the simulated robot's movements. However, some issues with lateral movement arises when combined with other inputs, causing the ROV to rotate instead of moving laterally. This seems to be an issue related to the accuracy of the hydrodynamic parameters in the X3's description.



(a) Gazebo (b) Unreal Engine

Figure 2.9: Comparison between relatively feature-less areas in both engines

The simulation demonstrates stable performance with minimal frame drops or latency issues for most of the trajectory. It is important to note that the target framerate is limited to the camera acquisition rate, as described in table 2.4. We tested emulating a camera with a higher framerate, but the hardware configuration for PC1 could not keep up, especially in areas with a higher particle count.

The synchronization between the back-end and front-end did not work as expected. We noticed that the Gazebo clock being published was not sent at regular intervals. Although the image feed was published consistently, their timestamps made it seem as though they were published in bulk. This is problematic as their timestamps no longer represent their time of acquisition. Further processing was necessary to fix these issues, as described in the dataset creation section. Figure 2.10 showcases a part of the synchronization issues.

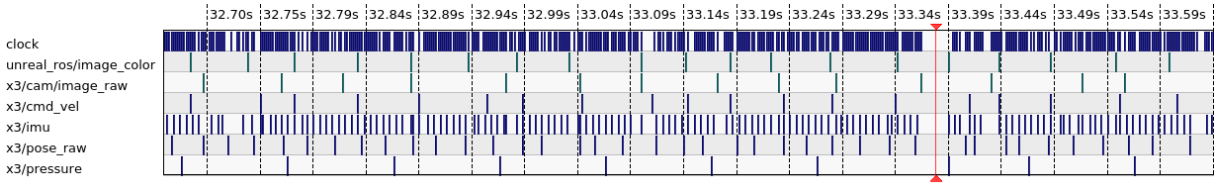
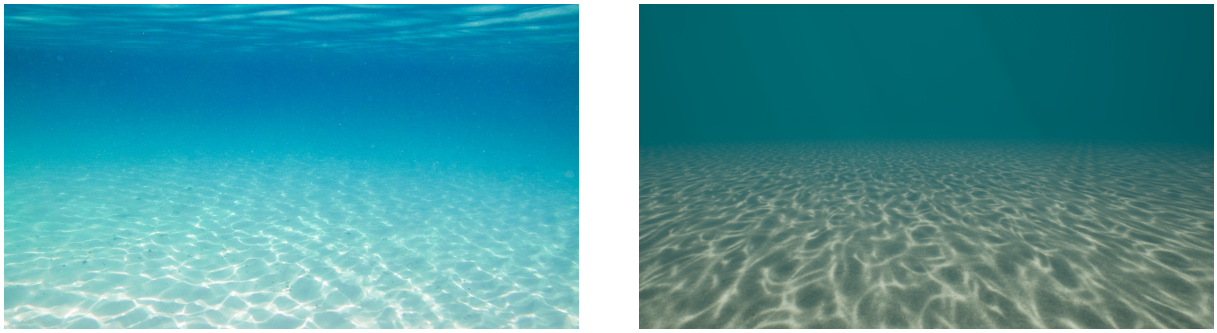


Figure 2.10: Synchronization issue as seen in Rqt Bag. Blue vertical lines indicate when the message is published to ROS. The red line highlights a section where the clock cycle is not published at a regular interval.

Sequence 2

For sequence 2, we only discuss the implementation of the environment and the simulation performance. All other aspects remain the same since we use the same ROV configuration.

For this sequence, the visuals are not as representative of underwater environments, as caustics are typically found only in shallower waters. In the simulation environment, not enough attention is given to the rest of the level that would better reflect a shallow environment outside of the use of caustics. Notably, the upper regions are darker than what they would be like in the real world, and the water surface is not visible.



(a) Real-world (b) Simulation

Figure 2.11: Comparison between caustics in the real-world and simulation. Real-world image sourced online

The simulation performance remained stable in this sequence, with only a slight increase in computational load due to the improved visibility. The caustics did not cause any noticeable performance drops.

2.4.2 User Experience and Usability

The user experience of the simulation is as a whole quite challenging, especially for those who want to make any changes. The user is expected to have knowledge of all the different components, meaning they need to be familiar with basic Linux commands (especially when running WSL2), ROS, Gazebo, the Project Dave package, and Unreal Engine. Additionally, to successfully implement a custom ROV, the user needs some knowledge of vehicle hydrodynamics. If users want to generate a reusable

dataset (e.g., creating a rosbag), extra steps must be taken. As such, it is not a simple plug-and-play solution even for experienced developers.

For those who simply wish to run the simulation and leave everything in the default configuration, the process is still cumbersome. To start the back-end, the user needs to open ROSBridge and run the launch file to initiate the Gazebo simulation. On the front-end, the user needs to start the level once the back-end is up and running.

Once the simulation is running, the user experience is fairly acceptable. The controls are intuitive, and the low latency makes it easy to operate the robot and navigate in the environment, despite the issues related to lateral movement. However, there is no built-in UI to provide information to the user. Visualization of the trajectory or reading sensor measurements must instead be viewed using external software such as Rviz or Rqt.

Overall, the user experience and usability of the simulation could be significantly improved with more user-friendly interfaces and streamlined setup processes. This would make the system more accessible to a wider range of users, including those with limited experience in the various components required to run the simulation. Furthermore, resolving issues with synchronization and lateral movement would enhance the performance and reliability of the simulation, creating a more robust and practical tool for researchers and developers in the underwater robotics field.

2.5 Discussion

Overall, the results indicate that the simulation is able to provide a realistic and challenging environment for evaluating visual SLAM algorithms and other underwater robotic applications. The visuals, such as the underwater lighting, particle effects, terrain, and sensor data generated are largely representative of real-world conditions. The low resolution of the camera and realistic noise levels contribute to the challenging nature of the simulation. This supports the potential use of the simulation for research and development purposes in the field of underwater robotics. We believe that the work done here serves as a good groundwork for further development.

Looking closer at Sequence 2, we note that the environment itself is not as accurate of a representation of an underwater scenario. The main reason for its creation is to make use of caustics which we typically do not see in similar datasets. Underwater datasets for SLAM are already limited, and to the best of the author's knowledge, no other datasets include caustic disturbances. We believe that this is still valuable for developing visual SLAM systems that are robust towards any and all disturbances that can be found underwater.

By far the most lacking aspect of the simulation is in its usability, both for end users and developers alike. The development pipeline is split between the back-end and front-end which is inefficient and requires a broad range of understanding of all the different parts to implement any significant changes. Moreover, further processing of a dataset is necessary for it to be fully usable. Ideally, development would only be required for one side of the simulation. This split between the two systems also introduces heavy latency due to the encoding and decoding of data. The Unity Robotics Hub developers have stated that using their custom TCP-endpoint reduced latency by an order of magnitude (Unity Technologies, 2022).

The end user would also benefit from having a simplified system for launching the simulation. Additionally, the inclusion of a GUI would make the simulation more intuitive.

2.5.1 Limitations

Some limitations of our study include the qualitative nature of the results, the hardware configuration used for the evaluation, and the scope of the implementation. The qualitative assessment may not capture all aspects of the simulation's performance and may be subject to personal interpretation. The hardware configuration, specifically the use of the 1-PC setup and WSL2, are likely to have affected the results, as different configurations could yield different performance characteristics. Additionally, our implementation focused on a limited set of features and scenarios specifically catered to visual SLAM.

2.5.2 Suggestions for future work

To address the limitations and further improve the simulation, several avenues for future work can be explored. This includes a more thorough quantitative evaluation of the simulation performance and testing on different hardware configurations. Specifically for quantitative evaluation, the computational performance of the system is key. Another metric that is important is the latency between the simulation and ROS. The simulation should also be expanded to cover new environments and support additional sensors such as sonar and LiDAR. A GUI would also be beneficial for the end-users.

Another avenue is to move the entire simulation to Unreal Engine exclusively. This would eliminate many issues related to sensor synchronization and reduce computational demand since the computer would only require running one engine. Doing so would also bypass the need for ROSIntegration, and the simulation can instead be implemented in Unreal Engine 5. To allow the simulation to interface better with other robotics systems, it should still maintain proper ROS integration.

For other applications, this type of simulation environment is highly beneficial to facilitate the training of machine learning techniques such as reinforcement learning for underwater robots.

2.6 Conclusion

In this chapter, we presented the design, implementation, and evaluation of an underwater simulation environment for visual SLAM and other robotic applications. The simulation was developed using Unreal Engine and Gazebo and focused on providing realistic visuals and challenging conditions that mimic real-world underwater scenarios. Our evaluation covered the performance, user experience, usability, and limitations of the simulation in two different environments.

The results show that the simulation is capable of generating realistic and challenging environments for evaluating visual SLAM algorithms. The visuals and sensor data closely resemble real-world underwater conditions, supporting the potential use of the simulation for research and development in the field of underwater robotics. However, usability remains a significant challenge, both for end users and developers, due to the split between the back-end and front-end and the broad range of knowledge required to implement changes.

There are several avenues that can be taken for future work, including quantitative evaluations, testing on different hardware configurations, expanding the evaluation to cover a wider range of scenarios and features, and implementing new environments, additional sensors, and a GUI. Moving the entire simulation to Unreal Engine exclusively while maintaining ROS integration is another possibility to streamline the development pipeline and reduce computational demand.

Chapter 3

Underwater Visual SLAM

3.1 Introduction

Simultaneous Localization and Mapping (SLAM) is a critical component in the development of autonomous robotics. It enables a robot to estimate its position and orientation in an unknown environment while simultaneously constructing a map of its surroundings. In the context of underwater robotics, SLAM algorithms face unique challenges due to the dynamic nature of the underwater environment and the inherent limitations of underwater sensors. Low-cost sensor configurations are an especially interesting category for underwater SLAM, one that has not been explored to the same extent as other configurations.

In this chapter, we investigate the adaptation of an existing visual SLAM algorithm, namely ORB-SLAM, for visual-pressure and visual-inertial-pressure SLAM configurations. We aim to accomplish the following objectives from our project description:

- Improve the visual pipeline of the SLAM algorithm to enhance camera performance underwater
- Implement a visual-pressure SLAM configuration
- Implement a visual-inertial-pressure SLAM configuration
- Investigate loop closure capabilities in the underwater domain

The chapter can be outlined as follows:

1. Introduction (this section): We provide a basic introduction to the subject and outline the rest of the chapter.

2. Background and related works: We provide an in-depth discussion on the principles of SLAM and its evolution over the years. We also provide examples of SLAM used in underwater robotics. This section also covers topics such as image processing and loop closure for the underwater domain.

3. Theory: We delve into the most important theoretical concepts that underpin the SLAM implementations in this chapter. The topics include coordinate frame notation, factor graphs, and least squares optimization.

4. Methodology: We discuss the adaptation of the ORB-SLAM algorithm for visual-pressure and visual-inertial-pressure SLAM configurations, detailing the steps taken to enhance the algorithm's performance in the underwater domain.

5. Results: We present the results of our implementation of the visual-pressure and visual-inertial-pressure SLAM configurations, comparing their performance to that of the original ORB SLAM.

6. Discussion: We delve deeper into the results, analyzing the performance of our proposed configurations and how they compare to existing literature.

7. Conclusion: We summarize our findings, discuss the implications of our work, and suggest possible avenues for future research in the field of underwater SLAM using low-cost sensor configurations.

3.2 Background and related Works

This Related works section aims to provide a comprehensive review of the existing literature and state-of-the-art techniques in the field of underwater SLAM. The main topics covered in this section include various SLAM frameworks such as Extended Kalman Filter (EKF) and Graph SLAM, visual SLAM techniques and image processing methods to enhance the quality of visual data; and loop closure algorithms for reliable place recognition.

3.2.1 Underwater SLAM

Underwater SLAM presents unique challenges that differentiate it from aerial and terrestrial SLAM applications. Some underwater environments are highly dynamic due to moving elements such as seagrass and dust particles kicked up from the seabed. Other environments have very uniform bathymetry and few distinct features, making it challenging to differentiate one area from another. Despite these challenges, significant progress has been made in developing SLAM algorithms specifically tailored for the underwater domain. The SLAM algorithms generally fall under three main frameworks: Extended Kalman Filter SLAM, FastSLAM, and Graph SLAM. The main goal of this section is to provide an overview of how these frameworks have been applied in the context of underwater SLAM, without delving into the in-depth explanation of how the algorithms work. For more detailed explanations, we refer the interested reader to (Hidalgo & Bräunl, 2015).

EKF SLAM

Extended Kalman Filter (EKF) SLAM has been widely used in the literature due to the extensive body of work on EKFs and its many variations. Localization and mapping estimates are stored in a state space model. This framework has 2 major drawbacks. Firstly, its computational complexity scales quadratically with the number of landmarks stored in the map. Secondly, the integration of linearization errors can cause the system to drift over time. Although studies continue to improve EKF approaches, they have fallen out of favor in recent years due to these limitations.

Some examples of its application underwater are presented in (E. Chen & Guo,

2014; Ribas et al., 2006). In (Ribas et al., 2006), the authors propose a two-stage EKF SLAM framework using an imaging sonar for partially structured underwater environments. The paper by (E. Chen & Guo, 2014) presents a real-time map generation method for unmanned underwater vehicles using sidescan sonar for estimating the vehicle's trajectory and creating a consistent map of the environment.

FastSLAM (Particle SLAM)

FastSLAM algorithms use particle filters in their design to address some of the limitations of EKF SLAM and was first introduced by (Montemerlo et al., 2002, 2003). Localization and mapping are estimated through particles, while the map itself is represented and updated using a separate Extended Kalman filter. Computational complexity increases by $N \cdot log(m)$, where N is the number of particles and m is the number of landmarks. FastSLAM approaches do not require linearization, but are prone to sampling degeneracy over time.

Some underwater implementations include and (L. Chen et al., 2020; He et al., 2012). In (He et al., 2012), the authors present a modified FastSLAM algorithm using a mechanical scanning forward-looking sonar, showing improved results over the traditional FastSLAM algorithm. (L. Chen et al., 2020) introduces a Rao-Blackwellized Particle Filter (RBPF) for SLAM using a slow mechanical scanning imaging sonar, providing accurate localization and a consistent map of the environment.

Graph SLAM

Graph SLAM introduces an intuitive method for solving the SLAM problem, with localization and mapping represented as nodes in a graph and edges as constraints based on sensor measurements and landmark observations. Current estimates take into account previous estimations, limiting system drift. The graph is solved by minimizing a cost function through a non-linear least squares problem. Graph SLAM is more computationally expensive than filter-based approaches but can produce more accurate maps. Its flexibility allows for easy integration of new sensor modalities without major redesign. Computational complexity scales linearly with the number of nodes and edges rather than landmarks, allowing for the possibility of solving a sparser graph to reduce computational load at the expense of accuracy. Early

works generally declared graph SLAM as an offline-only algorithm due to how computationally expensive it is.

(Klein & Murray, 2007) presents a parallel tracking and mapping (PTAM) algorithm for small AR workspaces, which laid the foundation for real-time graph SLAM algorithms. PTAM splits the system into a tracking and mapping thread. The tracking thread operates in real-time and processes new sensor information. A simplified motion model is also calculated to track the current pose and velocity. The mapping thread handles the computationally demanding task of correcting the map as new information is received. Today, many start-of-the-art SLAM algorithms use the concepts behind PTAM and graph SLAM in their system.

In underwater robotics, one notable work includes (Rahman, Li et al., 2018, 2019). In their first publication, (Rahman, Li et al., 2018) introduces a tightly-coupled multi-sensor fusion-based SLAM framework that combines monocular visual, inertial, and sonar data for underwater environments, employing a factor graph-based optimization approach. (Rahman et al., 2019) is the continuation of the previous work, adding depth measurements, image preprocessing, and real-time loop closure and relocalization capabilities. Their work presents one of the most robust implementations of underwater SLAM thus far.

AI SLAM

Using AI in SLAM is a relatively new concept that has emerged due to the surge in AI-related studies. Many of the studies conducted here implement hybrid approaches, only replacing sections of the SLAM pipeline with machine learning techniques. Monocular visual SLAM algorithms are the most attractive candidates, as machine learning could enable the system to recover depth and scale more reliably without additional sensor fusion. This branch is currently limited by the data it can use for training as well as its real-time performance, but nonetheless remains an interesting point for future works.

There is little in terms of literature where AI SLAM has been used in underwater environments, although one is presented here by (Teixeira et al., 2020). This paper proposes a Recurrent Neural Network (RNN) using supervised learning on visual odometry and inertial measurements. At inference time, the system is only fed

the visual odometry pose estimates, which are processed by a separate estimation algorithm. The algorithm shows promising results but currently is unable to run in real-time.

3.2.2 Visual SLAM

Visual SLAM has become increasingly popular in recent years, with cameras being used as the primary sensors to perform SLAM. This branch is particularly interesting for underwater robotics, due to the increasing affordability and availability of low-cost underwater robots. As previously mentioned, using cameras as primary sensors in underwater environments presents challenges such as limited visibility, light attenuation, and image distortion caused by refraction.

Visual SLAM algorithms are typically categorized as either direct or indirect, depending on their image processing methods. Direct approaches look at the pixel intensities of an image and attempt to match it in the next frame, while indirect approaches, also known as feature-based approaches, extract distinct features from images and attempt to find and match them in the following frame.

Direct approaches tend to be more robust toward noise in images and are computationally more efficient compared to indirect solutions. This efficiency allows the system to produce denser maps. Due to the inherent challenges of underwater environments, such as noise and low contrast, direct approaches might at first glance be more appropriate for underwater VSLAM. Direct approaches are typically constructed using a basic feature detector such as the popular Harris-Stephens (Harris) (Harris, Stephens et al., 1988) and Kanade-Lucas-Tomasi (KLT) detectors (Shi & Tomasi, 1994).

On the other hand, feature descriptors in indirect approaches allow for better data association, making them more robust towards short-term visual loss and essential for recognizing loop closures. Some examples of well-known feature descriptors include BRIEF (Calonder et al., 2012), BRISK (Leutenegger et al., 2011), FAST (Rosten & Drummond, 2006), ORB (Rublee et al., 2011), SIFT (Lowe, 2004), and SURF (Bay et al., 2006).

There have been recent attempts to create feature descriptors using deep learning techniques (Georgiou et al., 2020; Noh et al., 2017; Tian et al., 2017). These show

promising results compared to classical methods but are computationally expensive and perform well only within their trained datasets. Investigating how these descriptors perform underwater when trained with an appropriate dataset would be interesting, although their real-time performance might still be limited.

Visual SLAM algorithms tend to support various camera configurations, such as monocular, stereo, and RGBD. Monocular setups are the most common due to their cost and simplicity but suffer from an inability to recover depth information from images, leading to scale drift. Stereo configurations can recover depth information by comparing images taken by each camera, while RGBD cameras combine an RGB camera with a depth sensor to recover depth information. Despite the advantages of stereo and RGBD cameras, they present additional challenges such as increased hardware complexity, synchronization issues, and potential impact on power consumption.

Visual-inertial SLAM (VISLAM) is another popular technique that fuses visual SLAM algorithms with inertial measurements. This fusion is especially beneficial for monocular setups, as it enables the system to recover the correct scale of the trajectory and align it with the world reference frame.

Keyframe-based SLAM

Modern implementations of visual SLAM are more likely to use graph-based approaches, as they are better suited for handling the large number of features that are tracked at any given moment. Many of these take a specialized "Keyframe" approach. A "Frame" is created for every image that is processed by the system containing information such as the current pose of the robot and the features observed. However, frames can contain redundant information, especially when the robot moves slowly. Saving all this information is also memory intensive and adds greater computational requirements for the system. Certain frames containing the most valuable information for a specific section of the trajectory are chosen instead. These frames are then referred to as keyframes. Almost all state-of-the-art visual SLAM algorithms fall in this category (Engel et al., 2018; Forster et al., 2017; Leutenegger et al., 2016; Mur-Artal et al., 2015; Newcombe et al., 2011; Qin et al., 2018).

Although most visual SLAM algorithms have been designed for aerial and

terrestrial robots, some research shows that they can be applied to underwater robotics as well (Hidalgo Herencia, 2019). More specialized algorithms for underwater environments have also been developed. Notably, the works by Ferrera (Ferrera, 2019) and (Rahman et al., 2019) are highly relevant to this project, as they both fuse inertial and pressure measurements. (Ferrera, 2019) proposes UW-VO, a visual-inertial-pressure SLAM configuration using direct feature extraction using KLT. The paper shows very promising results but lacks loop closure capabilities. We mentioned (Rahman et al., 2019) earlier in section 3.2.1 which uses visual, inertial, sonar, and depth measurements. However, the system is also able to operate in a visual-inertial-pressure configuration.

Image processing

Visual SLAM algorithms require high-quality visual data to function optimally. Several studies have focused on enhancing underwater images, many of which are presented in this survey (Ahamed et al., 2019). The algorithms presented here improve images by reducing noise, increasing exposure in darker regions, correcting color, and enhancing contrast to make details and edges more distinct. Many of these have not been designed to operate in real-time. Instead, many visual SLAM approaches use simple histogram equalization (HE) techniques to improve the contrast of the images, which are efficient enough to run in real-time. However, directly applying HE may cause overexposure in certain areas of the image. Contrast limited adaptive histogram equalization (CLAHE) aims to solve this by dividing the image into a grid of local patches before equalizing them, ensuring better overall contrast (Pizer et al., 1987). Keep in mind that histogram equalization will do nothing to remove noise, and will in certain cases cause it to be more prevalent in the image.

The survey also reviews machine learning techniques for image enhancement, such as in (Anwar et al., 2018; Wang et al., 2017). Their results are impressive, but they cannot run in real-time and are limited to the domain of their training data.

3.2.3 Loop closure

Loop closure refers to a system's ability to recognize previously visited locations, which is essential for correcting accumulating errors in the trajectory. Without loop

closure, a system may regard a previously visited location as new and duplicate the map. Loop closure is typically achieved using a place recognition algorithm.

An earlier study by (Williams et al., 2009) describes three approaches typically used for visual SLAM place recognition: map-to-map, image-to-image, and image-to-map. Map-to-map approaches look for correspondences between features in two submaps. Image-to-image methods compare the current image with previously seen images. Finally, image-to-map methods look for correspondences between the latest image and the features in a map. The authors conclude that image-to-image and image-to-map approaches work best in visual SLAM. Image-to-map returned the highest number of true positives, but does not scale well to larger environments due to its computational demand. Image-to-image methods work well and scale better but require additional steps to check for geometric consistency, as they are susceptible to perceptual aliasing, where similar-looking locations may be wrongly associated as being the same place.

Image-to-image place recognition algorithms

In the literature, image-to-image (or appearance-based) techniques are the most popular for visual SLAM. The basic principle behind this is to build a database from the images collected during the operation, which can later be retrieved and compared with the latest image. If the images have enough similarities, a loop closure can be formed. To make the process more efficient, images are converted into a bag-of-words (BoW) representation, allowing for the development of quick and effective matching algorithms. However, images suffer from perceptual aliasing, so appearance-based techniques require an additional geometric validation step.

By far the most popular appearance-based place recognition algorithm is DBoW2 (Gálvez-López & Tardos, 2012). DBoW2 creates a bag of binary words based on the images received. This has proven to be very efficient for place recognition, enabling real-time operation. DBoW2 is frequently found in many state-of-the-art systems such as ORB-SLAM (Mur-Artal et al., 2015) and and SVIn2 (Rahman et al., 2019).

Alternative appearance-based place recognition methods have been proposed that may be better suited for the underwater domain. The approach using DBoW2 tends to be most effective only in structured environments. A novel cluster-based approach is presented by (Negre et al., 2016). The idea is to cluster tracked features in an image

using Density-Based Spatial Clustering of Applications with Noise (DBSCAN) (Ester et al., 1996). These clusters are transformed into a global signature using hash-based loop closure detection (HALOC) (Negre Carrasco et al., 2016), in contrast to the usual bag-of-words approaches such as DBoW2. HALOC has shown to be better suited for labeling and recognizing places in underwater environments. These signatures are then used to build a database similar to the bag-of-words process.

Loop closure candidates are queried based on their proximity to the clusters in the frame and by comparing their signatures. Geometric validation is performed once again before deciding whether a valid loop closure has been identified or not. The cluster-based approach presented by (Negre et al., 2016) shows promising results in underwater environments, offering an alternative to DBoW2 for underwater SLAM systems.

3.2.4 Summary and project direction

In summary, this related works section has provided an extensive overview of various methods and techniques employed in the field of underwater SLAM. The discussion has highlighted the unique challenges posed by underwater environments and the ways in which different SLAM frameworks, visual SLAM techniques, image processing methods, and loop closure algorithms address these challenges. Building on the insights gained from this review, this project aims to develop an underwater SLAM algorithm based on ORB-SLAM. Inspired by the work of Ferrera (Ferrera, 2019), we will introduce a visual-pressure and visual-inertial-pressure configuration. For comparison, our proposed system will employ a feature-based approach using ORB descriptors, as opposed to the direct approach presented by Ferrera. Moreover, ORB-SLAM includes a loop closure system which will enhance long-term data associations. The proposed algorithm aims to provide a more accurate and reliable solution for underwater visual SLAM applications.

3.3 Theory

In this section, we introduce a brief discussion on the relevant theory used throughout the project. This is to orient the reader on the most important concepts used throughout

the rest of the paper.

3.3.1 Coordinate frame notation

We are working in a 3-dimensional coordinate frame to represent the pose of the robot and landmarks in the map. These are described with respect to a given coordinate frame, such as the world reference frame W. A 3D position p with respect to a coordinate frame A can be written as a 3-element vector p_A. A transformation between frames can be described using a homogeneous transformation matrix $T_{AB} \in SE(3)$, that transforms points from frame B to A. This transformation matrix is composed of a 3×3 rotation matrix $R_{AB} \in SO(3)$ and a 3-element translation vector t_{AB}. To summarize, we use the following notation:

- W: world reference frame
- A: given coordinate frame
- p_A: 3d position with respect to coordinate frame A
- T_{AB}: homogeneous transformation matrix from frame B to A
- R_{AB}: rotation matrix from frame B to A
- t_{AB}: translation vector from frame B to A

The pose is described using a transformation from the robot to the world reference denoted as T_{WR}. In visual SLAM, R can be either the camera frame C, which is the coordinate frame attached to the camera, resulting in the transformation T_{WC}, or the IMU, also referred to as the body frame B with transformation T_{WB}. Additionally, extrinsic calibration between the camera and IMU is required to properly perform sensor fusion, as it helps in aligning the data from both sensors. The transformation T_{BC} must therefore be found using calibration software such as Kalibr (Rehder et al., 2016).

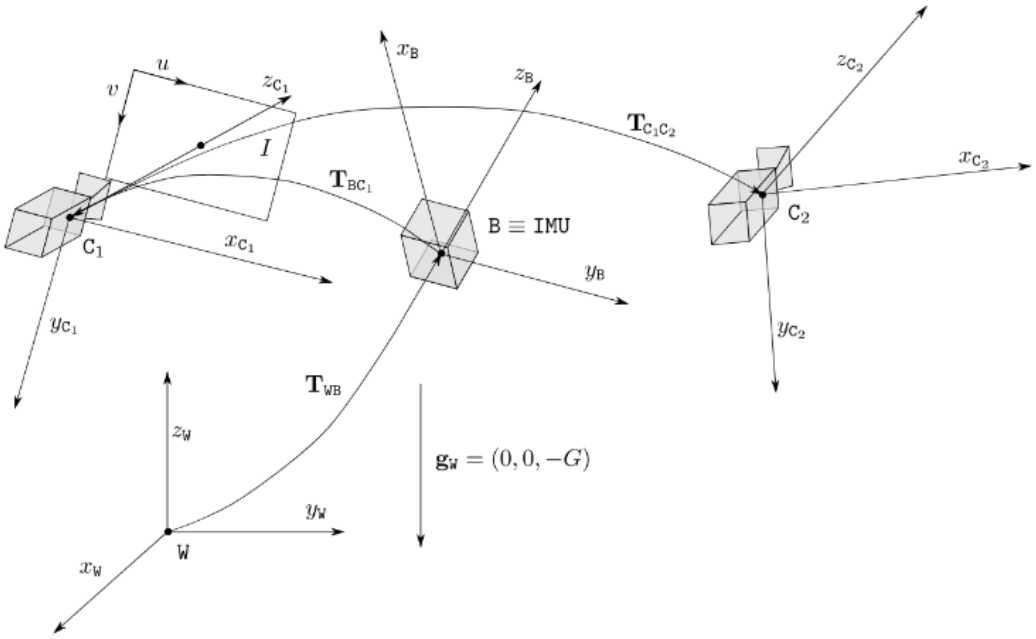


Figure 3.1: Reference system Source: (UZ-SLAMLab, 2022)

The world reference W is set as the first pose of the SLAM system when using a purely visual configuration (mono, stereo, or RGBD). With an IMU, W has its z_W axis aligned with the gravity vector g but pointing in the opposite direction. Translation and yaw are arbitrarily defined during initialization and become fixed once initialized

For the camera frame C, z_C points forward along the optical axis of the camera, y_C points down and x_C points to the right. Both are aligned with the image directions u and v.

3.3.2 Factor graphs

Graph SLAM employs the use of the concept of factor graphs to represent an optimization problem, where we maximize the posterior probability of the variables (poses and landmarks) given a set of measurements. Typically, we minimize a cost function by reducing the error between the estimates and the relevant sensor measurements. In most cases, this error term only involves a small subset of parameters of the total cost function. Factor graphs provide a very intuitive way of modeling this structure and can be used to represent a wide variety of problems. Nodes represent variables of our system that we wish to optimize, and factors or edges

connect them. These represent functions that describe the relation between the nodes. Edges can connect any number of nodes, depending on their relationships. Likewise, unary edges, which involve only one node, can also be constructed, for instance when constraining states in a node with direct measurements or a prior, which is a known probability distribution.

Different sensor modalities can also be easily added to the graph without having to change the underlying structure.

Figure 3.2 shows an example of a simple factor graph formulation. Circles represent nodes (3 pose states and 2 landmarks) while the squares represent edges linking the different nodes.

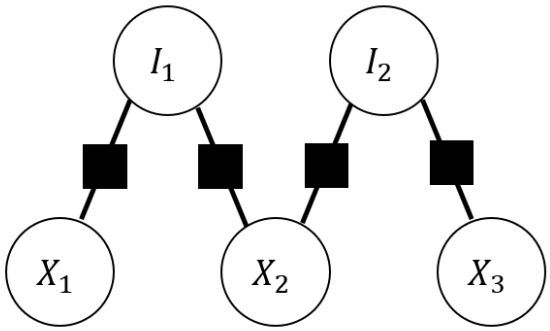


Figure 3.2: Factor graph example

In the context of our project, factor graphs are employed for pose optimization, local bundle adjustment, and global bundle adjustment. These methods are essential for refining the estimates of the robot's trajectory and the positions of the landmarks in the environment. The specific implementation details of these techniques will be further discussed in the methodology section.

3.3.3 Least-squares optimization

Factor graphs provide an intuitive way of constructing a model consisting of the robot's poses and landmarks. We can then optimize our model using a nonlinear least-squares approach. By minimizing the sum of squared errors between all observations and the predicted model, we can estimate a solution that best represents the trajectory and map. The information presented here is based on the description provided by G2O (Kümmerle et al., 2011); we refer the reader to the authors' work for a more detailed

explanation.

The function $F(x)$ that we wish to optimize can be expressed in the form:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \mathbf{F}(\mathbf{x}) \tag{3.1}$$

$$\mathbf{F}(\mathbf{x}) = \sum_{k \in \mathcal{C}} \underbrace{\mathbf{e}(\mathbf{x}_k, \mathbf{z}_k)^\top \Omega_k \mathbf{e}(\mathbf{x}_k, \mathbf{z}_k)}_{\mathbf{F}_k} \tag{3.2}$$

where:

- x: vector of parameters to be optimized.
- x_k: subset of parameters involved in the k^{th} constraint.
- z_k: constraint for the parameters x_k, usually a sensor measurement.
- Ω: information matrix of a constraint representing uncertainty in the system (e.g., sensor noise). The information matrix is also sometimes referred to as the precision or concentration matrix.
- $\mathrm{e}(\mathrm{x_k}, \mathrm{z_k})$: the error function measuring how well the parameters x_k satisfy the constraint z_k. It is equal to 0 when the parameters perfectly match the constraint. To simplify the notation, we will instead refer to this term in the form $e_k(x, z)$.

The error term e_k can also be defined as the Mahalanobis distance such that

$$e_k(x, z) = ||h(x_k) - z_k||_\Sigma \tag{3.3}$$

where h is the observation model relating the parameters x_k to the constraint z_k and Σ is the covariance of the measurement, and is equal to the inverse of the information matrix Ω. Note that this expression is identical to F_k from Equation 3.2. Both notations are used in this paper when describing error terms. When used in a factor graph, the error term is equivalent to the edge connecting relevant nodes.

Gauss-Newton

The Gauss-Newton method can be used to find the solution of a system given a good initial guess of its parameters $\breve{x}$. A good initial guess is required, as the method may

otherwise fail to converge or settle to a local minimum if the guess is too far from the true solution. We linearize the error function e_k using a first-order Taylor expansion around the initial guess such that

$$e_k(\breve{x}_k + \Delta x_k) = e_k(\breve{x} + \Delta x) \tag{3.4}$$

$$\simeq e_k + J_k \Delta x \tag{3.5}$$

where J_k is the Jacobian of e_k. Substituting in equation 3.2, we get the form:

$$F_k(\breve{x} + \Delta x) = e_k(\breve{x} + \Delta x)^T \Omega_k e_k(\breve{x} + \Delta x) \tag{3.6}$$

$$\simeq (e_k + J_k \Delta x)^T \Omega_k (e_k + J_k \Delta x) \tag{3.7}$$

$$= \underbrace{e_k^T \Omega_k e_k}_{c_k} + 2 \underbrace{e_k^T \Omega_k J_k}_{b_k} \Delta x + \Delta x^T \underbrace{J_k^T \Omega J_k}_{H_k} \Delta x \tag{3.8}$$

$$= c_k + 2 b_k \Delta x + \Delta x^T H_k \Delta x \tag{3.9}$$

Taking the derivative of 3.9 with respect to Δx, we get:

$$\frac{\partial}{\partial \Delta x}(c_k + 2 b_k \Delta x + \Delta x^T H_k \Delta x) = 0 \tag{3.10}$$

from which we can obtain the linear system:

$$H_k \Delta x^* = -b_k \tag{3.11}$$

where x^* is the updated parameters of the solution. Equation 3.9 can therefore be minimized in Δx by solving for the linear system in equation 3.11. The optimal update is found by searching the local gradient of the cost function, defined by the Hessian H_k. The linearized solution is obtained by adding the computed increments to the initial guess such that $x^* = \breve{x} + \Delta x$. In every iteration, the previous solution is used as the new linearization point.

Levenberg-Marquardt

The Gauss-Newton method requires a good initial estimate for it to converge to the correct solution. Otherwise, the system will fall into a local minimum. The Levenberg-Marquardt method attempts to solve this issue by adding a damping factor λ to control

the convergence of the system, which balances the optimization between the Gauss-Newton method and the gradient descent method.

$$(H + \lambda I)\Delta x^* = -b \tag{3.12}$$

The larger λ becomes, the smaller the change Δx will be. In practice, the optimization acts more like a gradient descent algorithm for large values in λ and more like Gauss-Newton for smaller values. If the error is lower than in the previous iteration, the lambda is decreased. Otherwise, the solution is reverted, and the lambda increased.

On-manifold optimization

Optimization over the pose's rotational component R is performed along the $SO(3)$ manifold to preserve the constraints imposed by the $SO(3)$ group, ensuring that the estimated rotations are valid. These rotations are then mapped back to the Euclidean space. The mapping functions are the following:

An exponential mapping from the Euclidean space to the manifold

$$exp : so(3) \rightarrow SO(3) \tag{3.13}$$

and a logarithm mapping from the manifold back to the Euclidean space

$$log : SO(3) \rightarrow so(3) \tag{3.14}$$

The exact details on how mapping is performed have not been included here. We refer the reader to (Blanco-Claraco, 2022; Kümmerle et al., 2011) for more information regarding this topic.

3.4 Methology: Overview

The methodology can be divided into three main parts:

1. ORB-SLAM Overview (3.5): The first part familiarizes the reader with the various components of ORB-SLAM3. This is important to understand where we make our modifications. Its general structure follow the tracking, local mapping, and loop closing systems.

2. Visual-pressure configuration (3.6): Here, we introduce our implementation of the visual-pressure configuration with ORB-SLAM as the base. It can be split into image preprocessing, tracking, local mapping, loop closing, and initialization.

3. Visual-inertial-pressure configuration (3.7): This describes the visual-inertial-pressure implementation. Here, we first give an overview of oRB-SLAM's visual-inertial configuration before discussing the implementation of tracking, local mapping, loop closing, and initialization.

3.5 ORB-SLAM3

We have included a section on ORB-SLAM for the reader to better understand the modifications that we will make on this algorithm. Specifically, we use ORB-SLAM3 as our starting point. The details have been condensed as much as possible. For more details, we highly recommend the reader to check the published literature of the original authors (ORB-SLAM (Mur-Artal et al., 2015), ORB-SLAM2 (Mur-Artal & Tardós, 2017a), ORB-SLAM-VI (Mur-Artal & Tardós, 2017b), ORB-SLAM3 (Campos et al., 2021)).

Over the years, ORB-SLAM, an open-source SLAM algorithm that uses ORB feature descriptors when processing images, has consistently proven to be among the top performers in visual SLAM. The system generates relatively sparse maps, which in turn allows for longer operations. The original ORB-SLAM algorithm only supports monocular camera configurations. ORB-SLAM2 extends this to include stereo and RGBD. ORB-SLAM-VI builds upon ORB-SLAM2 by fusing inertial measurements to the system, allowing for monocular setups to recover scale. ORB-SLAM3 refines

the inertial initialization procedure of ORB-SLAM-VI to allow for faster convergence. Additionally, the authors add a new system referred to as the atlas. Essentially, the atlas allows the system to save and reuse previously generated maps. If previously explored regions are recognized, the system will merge them with the current map.

ORB-SLAM can be split into 4 major parts: tracking, local mapping, loop closing, and the atlas, the latter being present only in ORB-SLAM3. An overview of the system is provided in Figure 3.3. We will only discuss the first 3 parts as implemented in ORB-SLAM3, as they are the ones relevant to the project. Each part runs on a separate thread. There is also an extra thread dedicated to performing global bundle adjustment which is controlled by the loop closure thread. This is done to prevent bottlenecks in the system, as the operation is costly

ORB-SLAM uses least-squares optimization across several areas of the system. We reference the cost functions for these when relevant. However, we do not provide the full details of every component as it is outside the scope of the project.

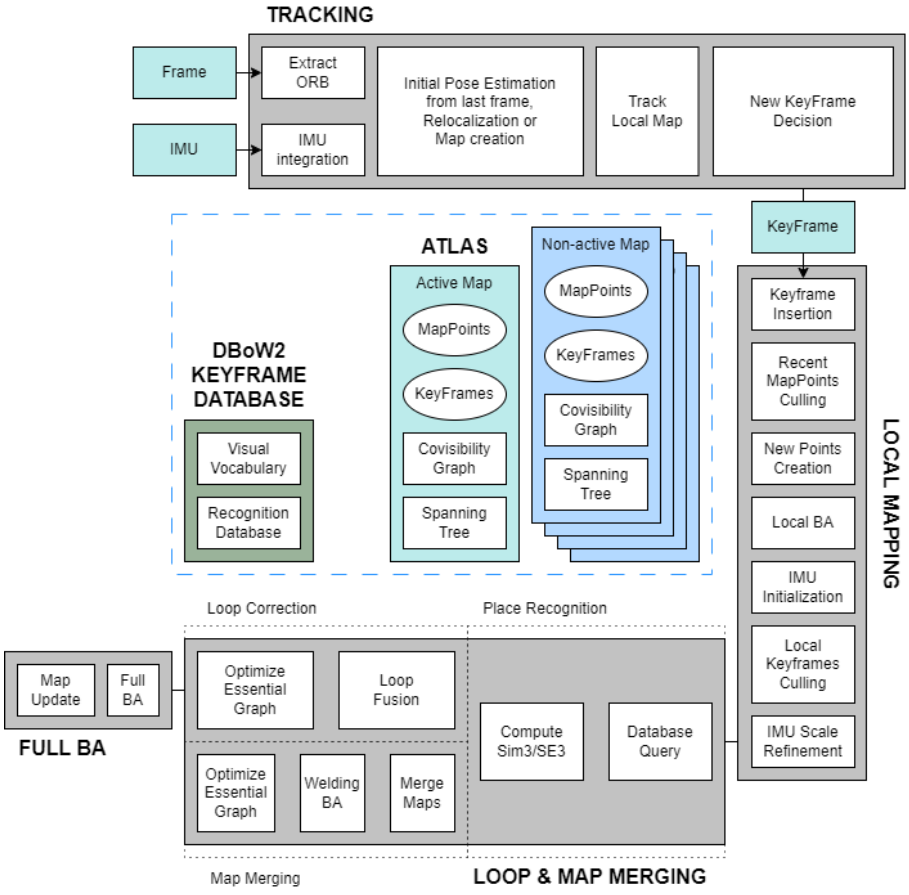


Figure 3.3: Redrawn ORB-SLAM3 overview. Original: (Campos et al., 2021)

3.5.1 Tracking

The tracking thread serves as the front-end, processing sensor data and computing the robot's pose at framerate. It decides whether to turn a frame into a keyframe, which is then sent to the local mapping thread. The tracking thread also handles map initialization and relocalization.

The tracking process starts by extracting ORB features from the image. An initial pose estimation is calculated based on the motion model of the previous frame and the features observed in the previous and current frame. A wider search window is used if not enough matches are found. In the inertial case, the initial pose estimate is instead calculated based on the preintegrated inertial measurements and bypasses the check for map point correspondences. This is computationally more efficient than the non-inertial case, as it does not require solving an optimization problem.

If the estimate still fails to meet the minimum number of matches, the thread will attempt to relocalize. The frame is converted into a bag-of-words representation and compared with the database for any keyframe candidates. If enough correspondences are found among one of the keyframes, the system will once again attempt to estimate an initial pose of the robot.

If an initial estimate is found, the system will refine its estimation by projecting a local map onto the frame to search for more correspondences. The pose will then be refined using all found map points. This process is run regardless of whether the pose was estimated using map point correspondences of the previous frame or inertial measurements.

Finally, the thread will decide whether the current frame is to be converted into a keyframe or not, based on a set of criteria: more than 20 frames must have passed since the last global relocalization (1), the local mapping thread is idle or more than 20 frames passed since the last keyframe insertion (2), the current frame tracks at least 50 points (3), and the current frame tracks at least 90% of points not found in the reference keyframe (4).

3.5.2 Local mapping

The local mapping thread is responsible for managing the local map, which consists of a set of keyframes and map points. It processes keyframes sent by the tracking thread, updating the covisibility graph and computing the bag of words vector for each keyframe. A covisibility graph connects keyframes with overlapping observations, making it easier to associate keyframes with each other. The local mapping thread then performs map point culling and creation based on their alignment with the current trajectory.

The trajectory and map are refined using local bundle adjustment, which optimizes the current keyframe, connected keyframes in the covisibility graph, and all map points observed by those keyframes. Other keyframes observing the same map points but not connected remain fixed during optimization. In the inertial case, local bundle adjustment instead takes place using a sliding window of keyframes and their points. Covisible keyframes are included but kept fixed during optimization. This sliding window optimization approach is designed to reduce computational complexity while maintaining accurate mapping.

Keyframe culling is performed to maintain a compact reconstruction and prevent the trajectory from becoming intractable. A keyframe is discarded if 90% of its map points have been seen in at least three other keyframes with the same or finer scale. This policy ensures that map points maintain keyframes from which they are measured with the highest accuracy.

In inertial mode, the system's initialization procedure, which estimates the initial inertial variables, scale, and gravity direction, is also performed within the local mapping thread. We provide more details on this in Section 3.7.5.

3.5.3 Loop closing

The loop closure thread is responsible for handling loop closures and map merging. Every time the local mapping thread processes a keyframe, the loop closure thread will try to detect a loop or a merge. This is done by computing its bag of words similarity vector with its neighbors in the covisibility graph using the DBoW2 bag-of-words place recognition system.

For each potential loop or merge candidate, a series of geometric verification steps is performed to achieve 100% precision. The geometric verification process involves computing the similarity transform to account for drift in translation and rotation (and scale in the purely monocular case), and checking for inliers with a reprojection error below a threshold. If a valid loop is found, the covisibility graph is updated to close the loop, and all keyframes in the loop including their neighbors are corrected using pose graph optimization. Finally, a full bundle adjustment process is run that includes all keyframes to further refine the estimate. This is done on a separate thread so as not to create a bottleneck in the system.

When a map merge is detected, a similar process is performed, but instead of closing a loop within the active map, a merge involves integrating multiple maps. The algorithm finds the aligning transformation between the maps and updates the covisibility graph accordingly. Subsequently, a global bundle adjustment process is run to refine the estimate, similar to the loop closure case.

3.5.4 ORB-SLAM3 setup

ORB-SLAM3 is originally designed to work with ROS Melodic, which runs on Ubuntu 18.04. A ROS-focused implementation of ORB-SLAM3 is however presented within this repository (thien94, 2023), which also happens to support Ubuntu 20.04 and ROS Noetic. The project already requires several other ROS libraries, and thus we use this version of ORB-SLAM3 in our development process. Table 3.1 lists the packages needed and the versions used in this project.

Package	**Version**
ORB-SLAM3	1.0
CMake	3.26
OpenCV	4.2
Pangolin	0.8.0
Eigen	3.3.7-2

Table 3.1: ORB-SLAM3 libraries

3.6 Visual-Pressure SLAM

In this section, we present the configuration that fuses visual and pressure sensor data to perform SLAM. The structure follows the outline of ORB-SLAM3 in 3.5, describing tracking, local mapping, and loop closing. Additionally, we include a section detailing the initialization procedure for this configuration.

Depth measurements obtained from pressure sensors provide an absolute value for the z-axis translational component, invariant to changes in time. By comparing these depth measurements to the estimated z-coordinate of the pose, we can scale the trajectory accordingly.

The current implementation of this visual-pressure SLAM system is designed to work with the monocular configuration of ORB-SLAM. In future work, this approach can be extended to accommodate stereo and RGBD configurations as well.

3.6.1 Tracking

We first make some additions to the visual component by preprocessing the images before feeding it to the rest of the system.

Improving the visual component

There are a few modifications that can be done to improve the visual component of the algorithm. One way of doing this is by preprocessing the images before feeding it to as input to SLAM. Underwater images often suffer from a lack of contrast and poor image quality, making it difficult to track distinct features. One of the most well known preprocessing techniques for underwater images is CLAHE as discussed in the related works (section 3.2.2).

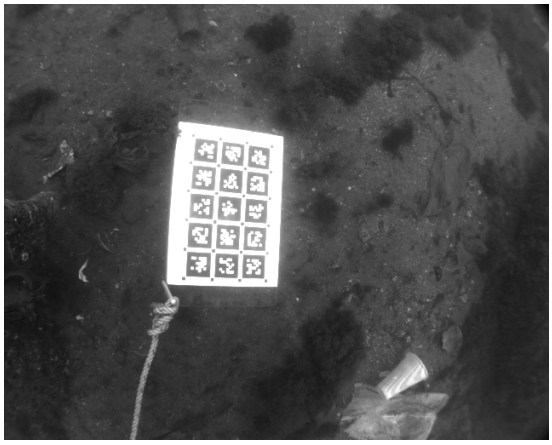
OpenCV contains a C++ library to easily implement CLAHE. There are two parameters that affect its behavior: the clipping limit and the grid size to divide the image by. The clip limit sets the threshold for the contrast enhancement. The higher this value is, the greater the contrast of the image patch. The grid size dictates how many sections to divide the image by. Each of these sections are the aforementioned patches.

There are no set rules for how to decide the values for these parameters as it is

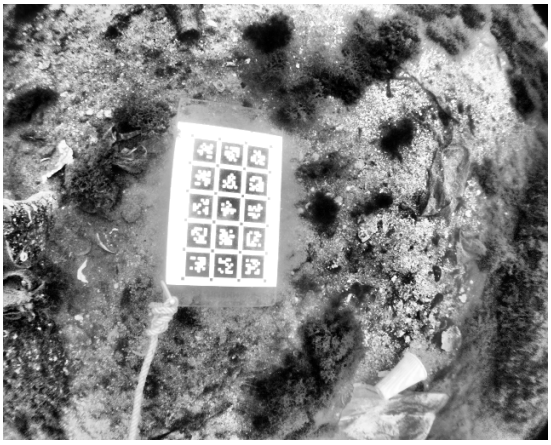
completely dependent on the images that you work with. These values have been chosen through empirical testing and by examining the number of features that the SLAM system can track. The suggested values can be seen in Table 3.2. The default values show good results across the board, but can be tweaked according to the range shown to fine-tune the system according to the environment.

Parameter	**Default**	**Operating range**
Clip limit	3	1 - 10
Grid size $(n \times n)$	6	4 - 20

Table 3.2: CLAHE values



(a) Original



(b) CLAHE clip limit 10

Figure 3.4: CLAHE

Pressure sensor

Pressure sensors measure the pressure exerted by the fluid as well as atmospheric pressure. Pressure scales linearly as we go deeper into the fluid. Depth measurements are related to pressure through the equation,

$$p = \rho \cdot g \cdot d + p_{atm} \tag{3.15}$$

where p and p_{atm} represent the measured pressure at depth d and atmospheric pressure, respectively, ρ denotes the fluid density, and g is the gravitational constant. In this model, we consider pressure measurement noise to follow a zero-mean Gaussian distribution.

$$p = p + v \quad , \quad v \sim \mathcal{N}(0,,\sigma^2) \tag{3.16}$$

By rearranging Equation3.15, we can directly obtain the expression for depth measurements. Incorporating noise from Equation3.16, we arrive at the following expression:

$$d_{raw} = \frac{p - p_{atm}}{\rho \cdot g} + v \quad , \quad v \sim \mathcal{N}(0,,\sigma^2) \tag{3.17}$$

where v' is the depth measurement noise derived from pressure measurement noise.

Since ORB-SLAM calculates its trajectory relative to its initial position, we need to compute the depth measurement relative to the initial position. Therefore, it is more convenient to refer to the depth measurement relative to its initial value when performing calculations. Thus, the final expression for depth d is given by:

$$d = d_{raw} - d_0 \tag{3.18}$$

Fetching depth measurements

Ideally, every captured image is associated with a unique depth measurement. However, this may not always be possible, primarily due to a mismatch in the sampling rate of the sensors. Typically, cameras have a higher sampling rate than pressure sensors. To assign depth measurements to each image, we make a few assumptions. The first assumption is that, although pressures sensor and cameras may have different sampling rates, they are likely to be within the same range. For example, the pressure sensor and camera may have sampling rates of 20 Hz and 30 Hz, respectively. Additionally, we assume that, due to the relatively slow motions of ROVs underwater, the changes between sensor measurements are relatively small.

In cases where an image cannot be associated with a unique depth measurement, we can simply assign the previous depth measurement taken at timestep $t - 1$ to the current image at timestep t. Practically speaking, the error between the copied measurement and what would have been the actual measurement is so small that it should have no noticeable impact in the system's performance. If the sampling rate of the pressure sensor is too low (e.g., 1 Hz), this method will not work. If the

sampling rate of the pressure sensor is greater than the camera and more than one depth measurement is taken between images, we take the newest measurement.

The practical implementation of this process is done using a ROS node. A subscriber listens to an image topic (the camera feed) and a pressure topic (the pressure sensor). Pressure sensor readings are saved as a vector and retrieved in a first-in-first-out (FIFO) manner. Whenever an image is received, we check the pressure sensor vector and use the first measurement with a timestamp later than the image. As mentioned earlier, if no such measurement is found in the vector, we instead reuse the previous measurement. With this approach, we can synchronize the sensors. The synchronized data is then passed on to ORB-SLAM.

The node supports both depth measurements and raw pressure measurements but needs to be specified beforehand.

The data received from ROS is passed on to the tracking thread and used to generate a frame. If the system is passed pressure readings, we first perform the conversion to depth here. The depth d saved per frame is relative to the initial depth as mentioned earlier. If the map has not been initialized yet, we set the initial depth d_0 equal to the raw depth d_{raw}.

Pose estimation

We fuse depth measurements during both pose estimation procedures in the tracking step, resulting in a tightly coupled fusion. The original cost function from ORB-SLAM estimates the pose X and follow the form:

$$\mathbf{X}_i^* = \underset{\mathbf{X}_i}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}_i) \right) \tag{3.19}$$

where $e_{visual}(\mathbf{X}_i)$ is defined as

$$e_{visual}(\mathbf{X}_i) = \sum_{j \in i} \rho_h \left(e_{ij}^T \cdot \Sigma_{ij}^{-1} \cdot e_{ij} \right) \tag{3.20}$$

where $\mathbf{X}_i$ is the pose at frame i, ρ_h is the robust Huber cost function, e_{ij} is the reprojection error of a map point j in the frame i, and Σ_{ij} is the covariance matrix. When running the pure visual configuration of ORB-SLAM, the error term e_{visual} is the only element of the cost function. Here, all map points are kept fixed during optimization.

We modify the cost function to include depth measurements, constraining the depth-aligned translation of the pose. The depth-aligned axis needs to be provided by the user before runtime, which we here denote as d_{align}. This parameter is a 3-element normalized unit vector that we can multiply to the translation, thus recovering the depth estimate. This parameterization allows the user to specify how the camera and depth sensor are oriented relative to each other without having to modify the code directly. This also reduces the ambiguity for the system when correcting the trajectory. We can describe the depth estimate d_{est} as:

$$d_{est} = t_{wc}^{T} \cdot d_{align} \tag{3.21}$$

where t_{wc} is the estimated translation vector of the pose $\mathbf{X}_i$.

We also need to take into account the potential misalignment between the estimated trajectory and the global reference frame. Misalignment occurs when the estimated trajectory does not align itself with the gravity vector of the world frame. This can happen for any camera-based setup since they have no way of relating their measurements with regards to the gravity vector. Depth measurements are always taken with regards to the gravity vector, thus a shift in the alignment of the estimated trajectory can create wrong correlations between the translation and measured depth. We can minimize this by aligning the trajectory with the world frame for example during the initialization procedure, but we cannot guarantee that the system will always converge towards the correct alignment. Figure 3.5 showcases the misalignment effect.

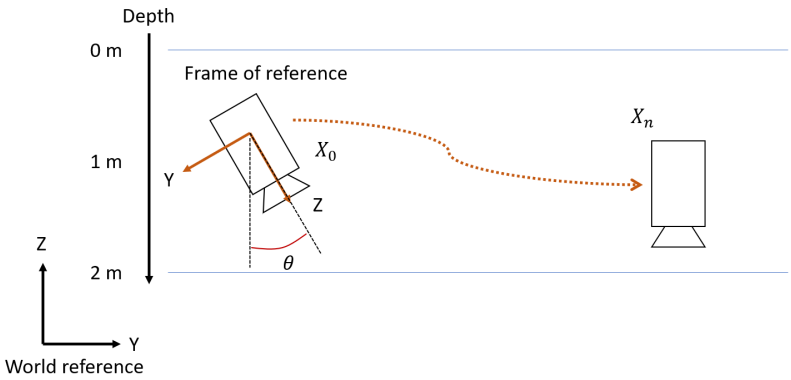


Figure 3.5: Misalignment effect. θ is the angle of error between the initial pose estimate and the world reference frame.

To increase the robustness towards misalignment, we have chosen to adopt the same strategy implemented in UW-VO (Ferrera, 2019). The strategy uses relative depth measurements between two consecutive frames, which we denote using d_{rel}. We then denote absolute depth measurements as just d. The relative depth can be expressed mathematically as:

$$d_{rel} = d_i - d_{i-1} \tag{3.22}$$

where i is the index of the current frame. Finally, we can write an expression for our error term e_{depth} for depth measurements as the squared Mahalanobis distance:

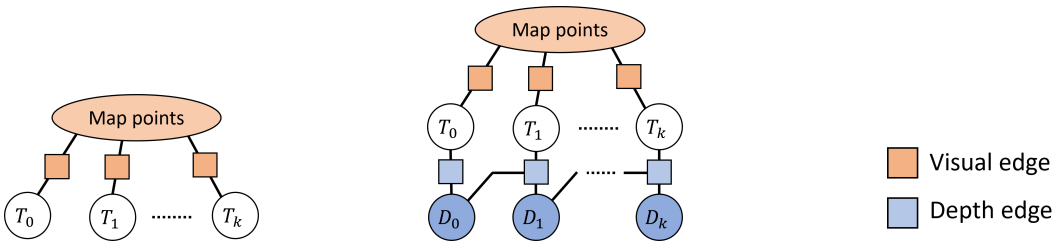
$$e_{depth-rel}(\mathbf{X}_i) = \|d_{rel-meas} - d_{rel-est}\|^2_{2\sigma^2} \tag{3.23}$$

Additionally, we increase the weight of this error term during optimization. This is to compensate for the fact that the visual residuals are relatively high compared to the depth.

The new cost function can therefore be written as:

$$\mathbf{X}_i^* = \underset{\mathbf{X}_i}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}_i) + e_{depth-rel}(\mathbf{X}_i) \right) \tag{3.24}$$

The graphical representation can be seen in Figure 3.6. Note that we only add our error term if the difference between consecutive depth measurements is less than 0.5 meters. We noticed occasionally in some datasets that the depth measurement spikes in its readings. Although this is more an issue with regards to the dataset, we have nonetheless included this check as part of the system permanently. The check is also applied to any other optimization procedure that uses the depth error term.



(a) Visual (b) Visual-pressure

Figure 3.6: Graph representation of the pose optimization function

3.6.2 Local mapping

The main changes to local mapping involve the local bundle adjustment algorithm. Visual-pressure initialization also occurs within this thread, but we cover this in a dedicated section in 3.6.4.

The original local bundle adjustment algorithm uses a similar error term e_{visual} as described in Equation3.20, but optimization occurs over a set of keyframes. The map points are also optimized, and some of the keyframes are kept fixed as described in section 3.5.2. The cost function is therefore in the form:

$$\mathbf{X}^* = \underset{\mathbf{X}_k}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}) \right) \tag{3.25}$$

where $e_{visual}(\mathbf{X})$ is defined as:

$$e_{visual}(\mathbf{X}) = \sum_{i \in k} \left(e_{visual}(\mathbf{X}_i) \right) \tag{3.26}$$

where k is the set of keyframes involved in the optimization. We have to update our error term for depth in the same manner, taking the sum of errors for the set of all keyframes k that are included.

$$e_{depth-rel}(\mathbf{X}) = \sum_{i \in k} \left(e_{depth-rel}(\mathbf{X}_i) \right) \tag{3.27}$$

The new local bundle adjustment equation can then be written as:

$$\mathbf{X}^* = \underset{\mathbf{X}_k}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}) + e_{depth-rel}(\mathbf{X}) \right) \tag{3.28}$$

3.6.3 Loop closing

Bag of Words

We primarily implement our changes to the loop closure algorithm by building a new ORB vocabulary tree for the system. ORB-SLAM3 uses DBoW2 for maintaining the place-recognition database. The vocabulary for this must be built offline.

Vocabulary training is a fairly straightforward process. Training requires a dataset of images and the same feature descriptor that will be used in SLAM, in this case, the ORB descriptor. ORB features are extracted from the image, and the image gets

converted to a bag-of-words vector. This bag-of-words vector uses the vocabulary to describe the contents of the image. The larger the vocabulary is, the more specific the place recognition algorithm can be at describing images, but this also increases the computational demand. The size of the vocabulary is dictated by the depth level (L) and k-branch factor (K), for which the total size of the vocabulary is equal to K^L. We refer interested readers to (Gálvez-López & Tardos, 2012) for more details regarding this algorithm.

We trained four different vocabularies to compare them with the base vocabulary. We use the default values for the depth and k-branch factor, as these have been found to be a good baseline for real-time place recognition algorithms (Gálvez-López & Tardos, 2012). These values are shown in Table 3.3. Table 3.4 summarizes the information regarding the different vocabularies.

Parameter	**Value**
Vocabulary depth level (L)	6
Vocabulary branch factor (K)	10

Table 3.3: Universal vocabulary parameters

Vocabulary name	**Total Images**	**Training data**	**Comments**
AH-Voc	6688	AQUALOC harbor	Highly fitted
AH-Voc-C	6711	AQUALOC harbor	CLAHE version
UW-Voc	11663	AQUALOC AFRL-VI	Universal
UW-Voc-C	11571	AQUALOC AFRL-VI	CLAHE version

Table 3.4: Summary of different vocabularies

Aqualoc Harbor Vocabulary (AH-Voc) This vocabulary has been trained on the AQUALOC harbor dataset (Ferrera et al., 2019). A majority of testing and development has been done using this dataset. We wanted to see if a well-fitted vocabulary would help establish loop closures more effectively. The training data is composed of each sequence having images taken at a rate of 10 Hz, resulting in a total of 6688 images.

Aqualoc Harbor Vocabulary CLAHE (AH-Voc-C) This dataset is the CLAHE equivalent of the Aqualoc Harbor Vocabulary. We performed a separate run for collecting images from each sequence with CLAHE applied to them. This vocabulary is trained on a total of 6711 images.

Underwater Vocabulary (UW-Voc) This vocabulary has been trained on the harbor and archaeological sites of the AQUALOC dataset and the bus and cave sequences of AFRL-VI (Rahman, Karapetyan et al., 2018). Images are taken at an acquisition rate of 5 Hz, resulting in a total of 11663 images. This results in a more general vocabulary, and research has shown that general vocabularies tend to yield good results (Mur-Artal & Tardós, 2014).

Underwater Vocabulary CLAHE (UW-Voc-C) This is the CLAHE equivalent of the underwater vocabulary. The images are collected in a separate run with CLAHE applied, with a total of 11571 images.

(a) AQUALOC - Harbor

(b) AQUALOC - Archaeological site

(c) AFRL - Bus

(d) AFRL - Cave

3.6.4 Initialization

The sensor fusion described shows how the system operates normally, but we must first calculate the scale and alignment of the trajectory. The error would otherwise be too high for the system to function nominally and thus produce an inaccurate trajectory. SLAM initialization is therefore one of the most important aspects for producing accurate trajectories.

We present 2 main categories of initialization, each with 2 variations, totaling 4 types of initialization strategies.

Strategy 1A: Iterative depth average

The first strategy is similar to the one presented in UW-VO and involves the following steps: A scale factor is calculated every time the system creates a new keyframe. The scale factor is taken as the average ratio between the absolute depth measurement and depth-aligned translation of the pose over the first N number of valid keyframes. By default, we set N to 10 keyframes. A keyframe is considered valid if it has depth value greater than the desired threshold, which we here set to 0.01 meters. The scale factor s can be expressed in the form:

$$s = \frac{1}{N} \sum_{i \in k}^{N} \frac{d_{meas}}{d_{est}} \quad , \quad N \geq 10 \tag{3.29}$$

where N is the number of valid keyframes in the current map. The factor s is then applied on the map. This processed is repeated until the scale factor is within 2% of its estimation and remains within the margin 10 times consecutively. Once the scale has been solved, we attempt to calculate the rotation matrix R needed to align the trajectory. We pose this is an optimization problem in the form:

$$R^* = \underset{R}{\operatorname{argmin}} \left(e_{init-r}(\mathbf{X}) \right) \tag{3.30}$$

$$e_{init-r}(\mathbf{X}) = \sum_{i \in k}^{N} \| d_{meas} - (R_{wc} \cdot t_{wc})^T \cdot d_{align} \|^2_{\sigma^2} \tag{3.31}$$

where R_{wc} is the estimated rotation matrix to align the trajectory, d_{meas} is the depth absolute measurement and t_{wc} is the estimated pose translation. All terms except for

R_{wc} is kept fixed during optimization. R_{wc} is part of the $SO(3)$ group and optimization is performed on the manifold. The rotation is then applied on the map.

Solving for the rotation matrix is a later addition to this strategy, hence why it is solved separate from the scale. We believe that this is sub-optimal as the scale and rotation are correlated to each other when minimizing the error in the trajectory.

We finally run a global bundle adjustment procedure to correct the trajectory and allow the fusion of depth measurements. It is similar to the bundle adjustment equation (Equation 3.28), with a few key differences. We use absolute depth measurements and we optimize over the entire trajectory instead.

$$\mathbf{X}^* = \underset{\mathbf{X}_{0:k}}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}) + e_{depth}(\mathbf{X}) \right) \tag{3.32}$$

$$e_{depth}(\mathbf{X}) = \sum_{i \in k} \left(e_{depth}(\mathbf{X}_i) \right) \tag{3.33}$$

$$e_{depth}(\mathbf{X}_i) = \| d_{meas} - d_{est} \|^2_{\sigma^2} \tag{3.34}$$

After running this procedure once, we allow the system to add depth error terms in pose estimation and local bundle adjustment as described in Section 3.6.1 and 3.6.2 respectively.

Strategy 1B: Set depth average

The second variation is very similar to the first one. We still calculate the average in the same manner. The difference is that we no longer calculate the scale at every iteration but instead at set intervals. This has the benefit of reducing computational load, even if the strategy is already fairly efficient, as well as prevent a situation where the calculated scale fluctuates too heavily, resulting in less reliable initialization.

The procedure here runs twice given a set of criteria. The first procedure happens after the system acquires 15 keyframes with depth measurements greater than 0.01 m. The second procedure occurs once we have a minimum of 30 keyframes with depth measurements greater than 0.02 m. Table 3.5 summarizes these criteria.

Strategy 1 advantages and disadvantages The greatest advantage to this strategy is its simple implementation as well as its computational efficiency. Its major

Procedure	Minimum keyframes	Threshold [m]
1	15	0.01
2	30	0.02

Table 3.5: Strategy 1B criteria

disadvantage is that it is not easy to expand. Notably, we would prefer to solve for scale and rotation at the same time. The implementation here also does not take into account sensor noise, and is highly sensitive to any outliers in the calculation.

Strategy 2A: Iterative depth optimization

This strategy addresses some of the shortcomings of the previous strategy. The main advantage is that here, we pose the initialization problem entirely as a graph optimization problem. This makes it easier to tightly integrate different representations of the parameters, in this case the scale and rotation matrices. This also makes it easier to expand in case we want to solve for additional parameters or integrate new measurements. Finally, the sensor's noise parameters can also be taken into account.

The cost function is very similar to Equation 3.30 but this time the scale s is also added as an optimization variable. Only s and R are optimized, the rest are kept fixed.

$$s^*, R^* = \underset{s,R}{\operatorname{argmin}} \left(e_{init-vp}(\mathbf{X}) \right) \tag{3.35}$$

$$e_{init-vp}(\mathbf{X}) = \sum_{i \in k}^{N} \| d_{meas} - s \cdot d_{est} \|^2_{\sigma^2} \tag{3.36}$$

$$d_{est} = (R_{wc} \cdot t_{wc})^T \cdot d_{align} \tag{3.37}$$

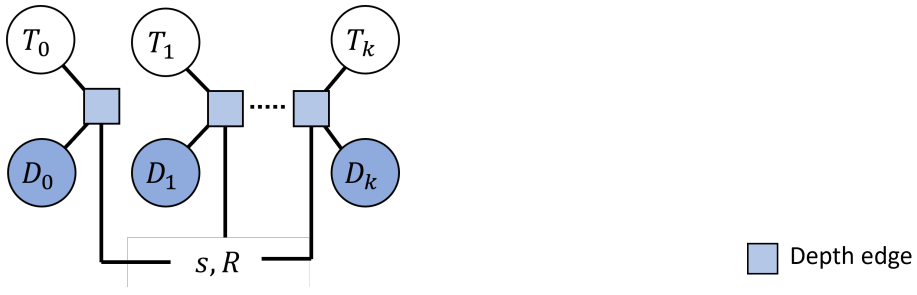


Figure 3.8: Initialization procedure. The poses are kept fixed.

Note that we do not actually apply the estimated rotation at every step. Instead, it is only applied after we obtain a consistent scale value. For some reason, ORB-SLAM3 does not keep track of any rotation that has been applied previously to the map, and we can therefore only apply it once. Nonetheless, we still solve for the rotation matrix jointly with the scale to improve accuracy.

After running the procedure once, we allow for depth to be fused in pose optimization and local bundle adjustment. The procedure is repeated until the scale factor is within 2% of its estimation and remains within this margin 10 times consecutively, after which we run global bundle adjustment.

This strategy has one major concern, namely that due to its use of non-linear least squares optimization, it becomes relatively expensive to compute at every iteration. Similar to Strategy 1A, it may also be susceptible to unreliable initialization if the scale fluctuates too heavily.

Strategy 2B: Set depth optimization

The final strategy combines the ideas of Strategy 1B and 2A. The procedure instead only takes place twice given a set criteria, and the procedure itself is posed as an optimization problem. This should significantly reduce the computational load of the system.

To reiterate, the procedure only runs when the given criteria has been met, namely with regards to the number of valid keyframes. Table 3.6 summarizes the criteria. We note that the values are identical to strategy 1B.

Procedure	Minimum keyframes	Threshold [m]
1	15	0.01
2	30	0.02

Table 3.6: Strategy 2B criteria

Strategy 2 advantages and disadvantages Since the initialization procedure here is posed as an optimization problem, it retains the advantages and disadvantages of using a factor graph. Setting it up is intuitive and can easily be expanded with new measurements. The disadvantage is that it can become computationally expensive.

Ideal motion for initialization

Typically, the proposed initialization strategies are able to correctly estimate scale and rotation close to the true values as long as there is some motion in the depth-aligned axis. Even minimal motion here is enough for successful initialization. Mainly, the issue arises when there is considerable misalignment between the trajectory and world reference frame, or when the pressure sensor acquisition rate is very low. We can maximize the reliability and accuracy of these initialization procedures given a few guidelines on how the ROV should move.

It is optimal for the system to have some translational motion across all axes and preferably avoid motion where the measured depth values are close to zero. This holds especially true when there is considerable misalignment between the trajectory and the world reference frame. This allows for the system to initialize much faster and more reliably. Translation across multiple axes reduces the ambiguity especially for estimating rotation, where several configurations oftentimes satisfy the constraints set during graph optimization.

3.7 Visual-Inertial-Pressure SLAM

In this section, we address the challenge of integrating visual, inertial, and pressure sensor modalities to perform SLAM. Although depth measurements have proven effective in recovering the correct scale, they face limitations when there is little to no motion along the depth axis. Moreover, the misalignment between the estimated trajectory and the world reference scale can further compound these limitations. IMUs offer a solution to these issues, as they can easily recover the correct alignment by aligning the gravity vector measured by the accelerometer, even in scenarios with limited motion. When initialized successfully, IMUs can accurately estimate the up-to-scale trajectory and motions of an underwater vehicle, allowing it to operate for limited periods with low visual information.

3.7.1 Challenges of visual-inertial SLAM underwater

The primary challenge in utilizing the visual-inertial configuration of ORB-SLAM underwater is the unreliable initialization procedure. Proper initialization requires determining the initial bias of the IMU, which is crucial for accurately estimating the ROV's motions. Typically, the initial bias can be reliably recovered by exciting the IMU across all axes through translational and rotational motion. However, this becomes difficult underwater, where ROVs are constrained by slower motions. The challenge is further exacerbated in monocular configurations, where the system must also estimate the scale.

To overcome these challenges, depth measurements are incorporated into the initialization procedure. The additional depth constraints assist in determining the correct scale and alignment, indirectly constraining the bias during optimization. As a result, the system can successfully initialize even when there is limited motion, provided there is sufficient variation along the depth axis.

We use the visual-inertial configuration of ORB-SLAM3 as our baseline for integrating these sensor modalities. Since the alignment can be reliably corrected, we also employ absolute depth measurements when optimizing the cost functions, which should lead to a more accurate overall trajectory compared to the relative depth measurements used in the visual-pressure configuration.

The following sections discuss key components of the visual-inertial-pressure (VIP) SLAM configuration, including tracking, local mapping, and loop closure, as well as the initialization procedure.

3.7.2 Tracking

For the first pose estimation, the base visual-inertial configuration uses the estimated motions of the IMU to predict the current pose and velocity in the robot. Since the system does not need to perform graph optimization, it is considerably less computationally expensive. This does come at a cost of some accuracy. However, in the underwater domain, it was much more prone to tracking loss, likely due to the slow motions of the robot. We have therefore opted to use the same pose optimization as in the previous section, reusing Equation 3.24 which only uses visual and depth error terms.

$$\mathbf{X}_i^* = \underset{\mathbf{X}_i}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}_i) + e_{depth-rel}(\mathbf{X}_i) \right) \tag{3.24 revisited}$$

In the second pose estimation step, we fuse all three components: visual, inertial, and depth measurements. The addition of inertial and depth measurements allow us to lower the number of visually matched inliers for tracking to be considered a success. The equation to be optimized is shown below:

The original visual-inertial pose estimation function is written as:

$$\mathbf{X}_i^* = \underset{\mathbf{X}_i}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}_i) + e_{inertial}(\mathbf{X}_i) \right) \tag{3.38}$$

where

$$e_{visual}(\mathbf{X}_i) = \sum_{j \in i} \rho_h \left(e_{ij}^T \cdot \Sigma_{ij}^{-1} \cdot e_{ij} \right) \tag{3.20 revisited}$$

$$e_{inertial}(\mathbf{X}_i) = \left[e_{\Delta R_{i-1,i}}, e_{\Delta v_{i-1,i}}, e_{\Delta p_{i-1,i}} \right] \tag{3.39}$$

The terms in the vector $e_{inertial}$ are the error terms for the preintegrated rotation, velocity, and position measurements denoted as $e_{\Delta R_{i-1,i}}$, $e_{\Delta v_{i-1,i}}$ and $e_{\Delta p_{i-1,i}}$ respectively. $e_{inertial}$ also includes a covariance matrix $\Sigma_{i-1,i}$. Their full definitions have not been

included here. Instead, we refer the reader to the paper on ORB-SLAM3 (Campos et al., 2021).

Adding our depth term to the cost function, we get:

$$\mathbf{X}_i^* = \underset{\mathbf{X}_i}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}_i) + e_{inertial}(\mathbf{X}) + e_{depth}(\mathbf{X}_i) \right) \tag{3.40}$$

$$e_{depth}(\mathbf{X}_i) = \|d_{meas} - d_{est}\|^2_{\sigma^2} \tag{3.34 revisited}$$

Note the use of absolute depth measurements. We can use this as initialization is able to reliably recover scale and alignment. Absolute measurements are therefore able to constrain the pose estimation globally, resulting in an overall more accurate trajectory.

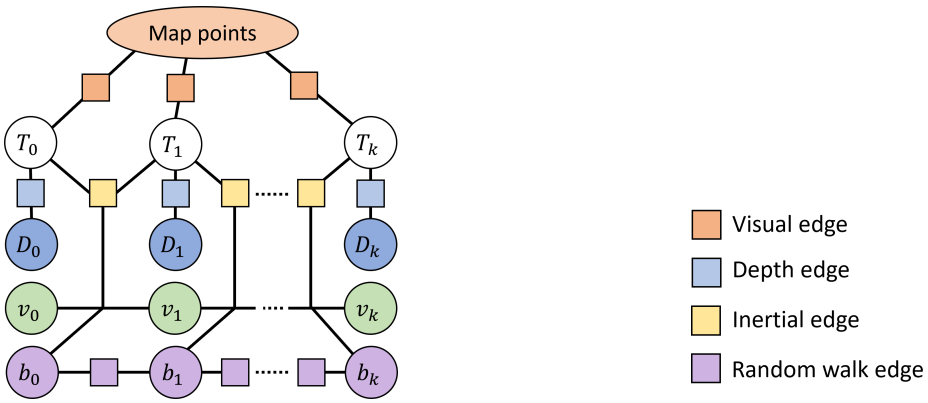


Figure 3.9: VIP optimization function

When it comes to the second pose estimation step, it creates a local map using map points found from previous frames that are used for evaluating the current frame. These map points are projected into the frame for comparison. The visual-inertial configuration for ORB-SLAM searches points over a wider projection. This results in a coarser search and thus more map points. For reference, the base monocular configuration has a search threshold of 1 while the visual-inertial configuration has a search threshold of 10 when the IMU is uninitialized, 6 when partially initialized, and 2 during normal operation. While the value for normal operation is reasonable, we found that the values of 10 and 6 before proper initialization greatly diminished the accuracy of the pose prediction for our configuration. We have therefore set our values to 2, 2, and 1 respectively.

We also implement a modified motion estimation procedure using the IMU and depth measurements for situations with little visual information. Usually, the IMU motion prediction calculates the pose and velocity according to the IMU readings and accumulated bias. We scale the delta between the previous pose and current pose according to the ratio between the current and previous depth measurement versus the current and previous depth-aligned translation.

$$t_{wb-\Delta-corrected} = s \cdot t_{wb-\Delta} \tag{3.41}$$

$$s = d_{meas} / d_{est} \tag{3.42}$$

Due to the rapid accumulation of bias, the system is only able to operate in this state for a short time (about 1-2 seconds).

3.7.3 Local mapping

We modify the implementation of the inertial local bundle adjustment process of ORB-SLAM by adding absolute depth measurements to the cost function. Equations 3.43 and 3.45 show the original and new cost functions respectively.

$$\mathbf{X}^* = \underset{\mathbf{X}}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}) + e_{inertial}(\mathbf{X}) \right) \tag{3.43}$$

where

$$e_{visual}(\mathbf{X}) = \sum_{i \in k} \left(e_{visual}(\mathbf{X}_i) \right) \tag{3.26 revisited}$$

$$e_{inertial}(\mathbf{X}) = \sum_{i \in k} \left(e_{inertial}(\mathbf{X}_i) \right) \tag{3.44}$$

Adding in the absolute depth error term, we get:

$$\mathbf{X}^* = \underset{\mathbf{X}}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}) + e_{inertial}(\mathbf{X}) + e_{depth}(\mathbf{X}) \right) \tag{3.45}$$

$$e_{depth}(\mathbf{X}) = \sum_{i \in k} \left(e_{depth}(\mathbf{X}_i) \right) \tag{3.33 revisited}$$

3.7.4 Loop closure

We implement the same changes to loop closure as described in Section 3.6.3. The default ORB vocabulary is simply replaced with the custom-trained vocabularies.

3.7.5 Initialization

The initialization procedure described here is inspired by the visual-inertial initialization procedure shown in ORB-SLAM3. To summarize how it works, it initializes the visual and inertial systems separately before doing joint optimization once each system attains good initial estimates.

The steps can be enumerated as follows:

1. Vision-only estimation: The system initializes running only pure monocular SLAM. We continue this until we have 15 keyframes in the map meeting the minimum measured depth threshold. The poses are then transformed and observed with respect to its body frame instead of the camera frame.

2. Inertial-depth estimation: Here, we estimate the optimal inertial variables (biases and velocities) as well as the scale and alignment. We use depth measurements to aid in this process. The variables can be stacked into the state vector Y_k such that:

$$Y_k = \{s, R_{\mathrm{w}g}, b, \bar{v}_{0:k}\} \tag{3.46}$$

   where s is the scale factor, R_{wg} is the rotation matrix needed to align the trajectory with the world reference frame, b is the bias for the accelerometer and gyroscope, assumed to be constant during initialization, and $\bar{v}_{0:k}$ is the up-to-scale body velocities starting from the first keyframe, initially estimated from the Vision-only estimation step.

3. Visual-inertial-pressure estimation: With good initial estimates for the inertial and visual parameters, we perform a joint visual-inertial-pressure optimization procedure to refine the solution. This is achieved by running a global bundle adjustment procedure.

The state vector Y_k is originally solved in ORB-SLAM3 with the following cost function:

$$\mathcal{Y}_k^* = \underset{\mathbf{Y}_k}{\operatorname{argmin}} \left(||b||^2_{\Sigma_b^{-1}} + \sum_{i=1}^{k} \left(e_{inertial}(\mathbf{X}_i) \right) \right) \tag{3.47}$$

We add a similar error term as in section 3.6.4. The difference is that we compare it with the z-translational component of the body pose t_{wb}. Since the depth is taken relative to the camera pose, we need to take into account the translational offset between the camera and the body frame, which we can retrieve from the transformation matrix T_{cb}. We will refer to its translational component as t_{cb}.

$$e_{init-depth}(\mathbf{X}_i) = \| d_{meas} - s \cdot d_{est} - t_{cb}^T \cdot d_{align} \|^2_{\sigma^2} \tag{3.48}$$

$$d_{est} = (R_{wc} \cdot t_{wb})^T \cdot d_{align} \tag{3.49}$$

Adding it to 3.47, the new cost function can be written in the form:

$$\mathcal{Y}_k^* = \underset{\mathbf{Y}_k}{\operatorname{argmin}} \left(||b||^2_{\Sigma_b^{-1}} + \sum_{i=1}^{k} \left(e_{inertial}(\mathbf{X}_i) \right) + \sum_{i=1}^{k} \left(e_{init-depth}(\mathbf{X}_i) \right) \right) \tag{3.50}$$

The factor graph representation of this is found in Figure 3.10.

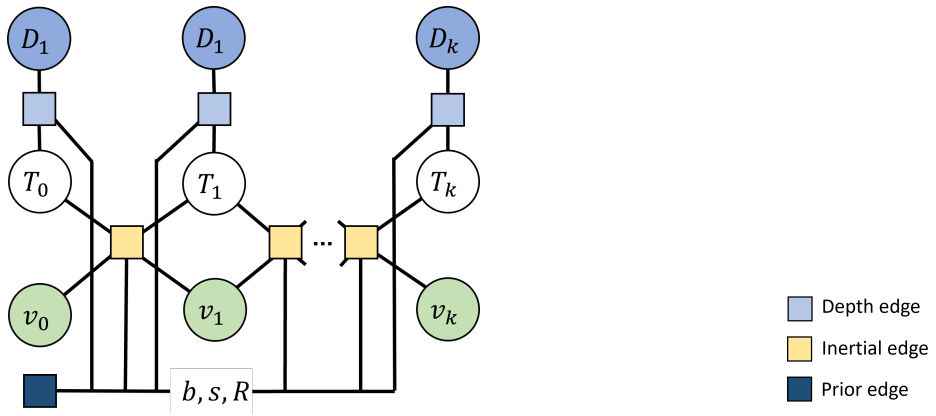


Figure 3.10: VIP initialization procedure

For the joint optimization in step 3, we simply add the absolute depth error term to the bundle adjustment procedure. The global bundle adjustment is very similar to Equation 3.45, but uses common biases for all keyframes, includes the same prior information for biases, and optimizes all keyframes currently stored in the map.

$$\mathbf{X}^* = \underset{\mathbf{X}_{0:k}}{\operatorname{argmin}} \left(e_{visual}(\mathbf{X}) + e_{inertial}(\mathbf{X}) + e_{depth}(\mathbf{X}) \right) \tag{3.51}$$

After running this procedure once, we allow the system to run its normal VIP configuration. We repeat steps 2 and 3 twice to improve the initial estimates further. This occurs after the system contains a certain number of keyframes that meet the minimum measured depth threshold similar to the step 1. Table 3.7 provides a quick summary of the steps performed, the number of keyframes required, and the depth threshold.

Procedure	**Minimum keyframes**	**Threshold [m]**	**Time since last proc. [s]**	**Step 1**	**Step 2**	**Step 3**
1	15	0.01	5	✓	✓	✓
2	20	0.03	5		✓	✓
3	20	0.05	5		✓	✓

Table 3.7: VIP Initialization procedure

This initialization procedure is more robust in the underwater environment compared to the visual-inertial initialization of ORB-SLAM3, which would in most cases fail to initialize at all or was unable to converge towards an accurate solution. Initialization for this configuration is also more consistent compared to the visual-pressure configuration since there is less ambiguity when solving for the scale and alignment.

3.8 Results

In this section, we analyse the performance of the proposed SLAM configurations. First, we assess the impact of image preprocessing on tracking performance. Next, we evaluate how well the system can estimate the given trajectory. This is the most comprehensive section, covering the accuracy of the trajectory, how much the system drifts, and how well the initialization procedures work. Finally, we discuss the changes to loop closure.

The majority of the tests were conducted on the AQUALOC Harbor dataset (Ferrera et al., 2019). AQUALOC is an underwater dataset for visual-inertial-pressure localization. It contains three primary environments; a harbor environment, and two archaeological sites. The ground truth for these datasets have been reconstructed using Colmap, a state-of-the-art SfM reconstruction algorithm (Schonberger & Frahm, 2016). We will simply refer to AQUALOC Harbor as the Harbor dataset. For some of the tests, we include Sequence 1 from the simulation presented in Chapter 2. we will refer to this dataset as UwUE. All results are gathered over the course of 10 runs for each sequence included in the tests.

We also attempted testing the system on other datasets, such as the archaelogical sites in AQUALOC. However, we found that our system was unable to properly operate in these environments as they exhibit a large amount of dynamic disturbances. We elaborate on this further in the discussion section (3.9).

Unless otherwise specified, we use the parameters for the visual component detailed in Table 3.8 in all our tests.

Parameter	Value
Number of tracked points	1500
CLAHE grid size	6 x 6
CLAHE clip limit	10

Table 3.8: Default visual parameters

3.8.1 Image preprocessing

In this section, we review the results of preprocessing the image using CLAHE.

We evaluate our implementation by measuring how consistently the system is able to extract features from images and how many inliers it can find. The number of extracted features decreases the likelihood of losing track due to low texture and low contrast present underwater. However, this metric by itself is not sufficient to describe an improvement in tracking. Ultimately, the number of inliers provide a better indication to the performance of the system; the more inliers there are, the more map points that can be added to the map and the more accurate the trajectory estimation can be. This is still not a perfect metric as it can in certain cases cause noise to be registered as a feature in the environment.

We test a total of four CLAHE configurations to evaluate the effectiveness of different CLAHE values. The details regarding each configuration is shown in Table 3.9.

Parameter	**No-CLAHE**	**CLAHE-1**	**CLAHE-3**	**CLAHE-10**
Number of tracked points	3000	3000	3000	3000
CLAHE grid size	-	6 x 6	6 x 6	6 x 6
CLAHE clip limit	-	1	3	10

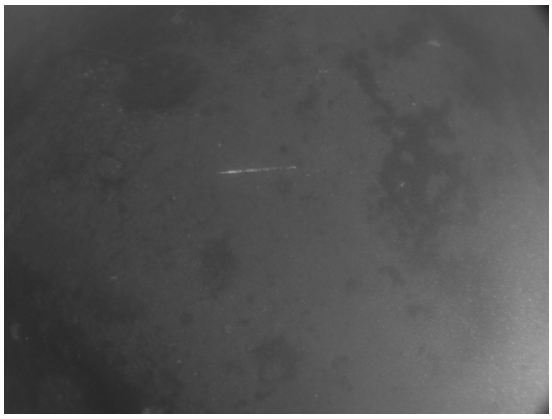
Table 3.9: CLAHE configurations

The number of features have been set higher than usual. In normal operations, the system usually manages to maintain a stable number of tracked features. We have increased this value to better highlight the results of each configuration.

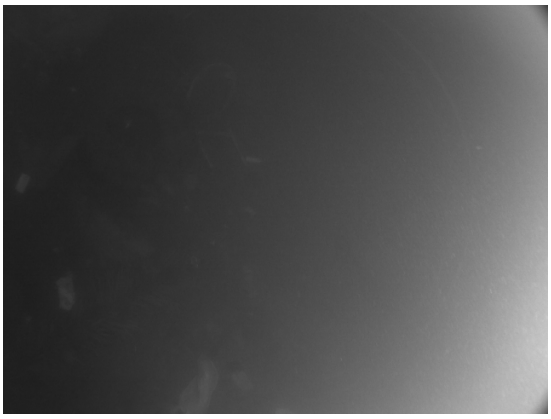
We have chosen Harbor sequences 1 and 4, as well as UwUE sequence 1 to collect our results. Sample images of the environments are given in Figure 3.11. The first harbor sequence contains a section with a very uniform seabed, making it difficult to identify distinct features. The fourth sequence contains a section where the ROV travels high above the seabed, obscuring the view below due to the haziness of the water. In normal operations, the system will usually fail to relocalize in this section. The UwUE sequence contains very low illumination and contrast. Unlike in the harbor datasets, the modelled ROV here uses a forward-facing camera.

CLAHE-3 AND CLAHE-10 are used for the harbor sequences while CLAHE-1 AND CLAHE-3 are used for the UwUE sequence. The UwUE uses more conservative values due to some inherent limitations of the simulation. Namely, when the CLAHE

clip limit is too high, an unwanted halo is projected onto the center of the camera view. Using too high of a clip limit causes the system to falsely assume that it is a feature in the environment (Figure 3.11d).

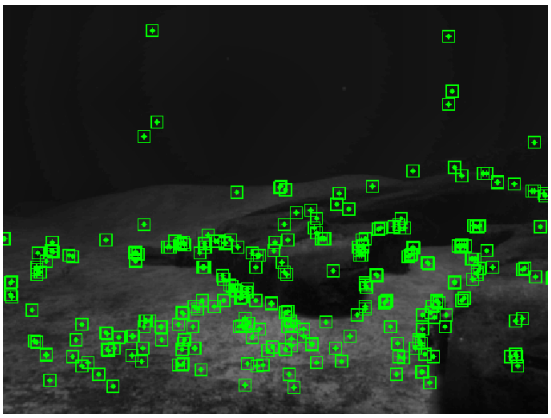


(a) Harbor 1: low-texture area



(b) Harbor 4: ROV high above the seabed

(c) UwUE 1: Dark environment



(d) UwUE 1: Halo effect. The halo is mistaken for a landmark in the environment.

Figure 3.11: Sample images from the chosen sequences

The results shown in Figures 3.12, 3.13 and 3.14 plot the mean number of tracked points and inliers over the course of the trajectory. The data has been resampled at a rate of 1 Hz, down from the camera framerate of 20 Hz for all sequences.

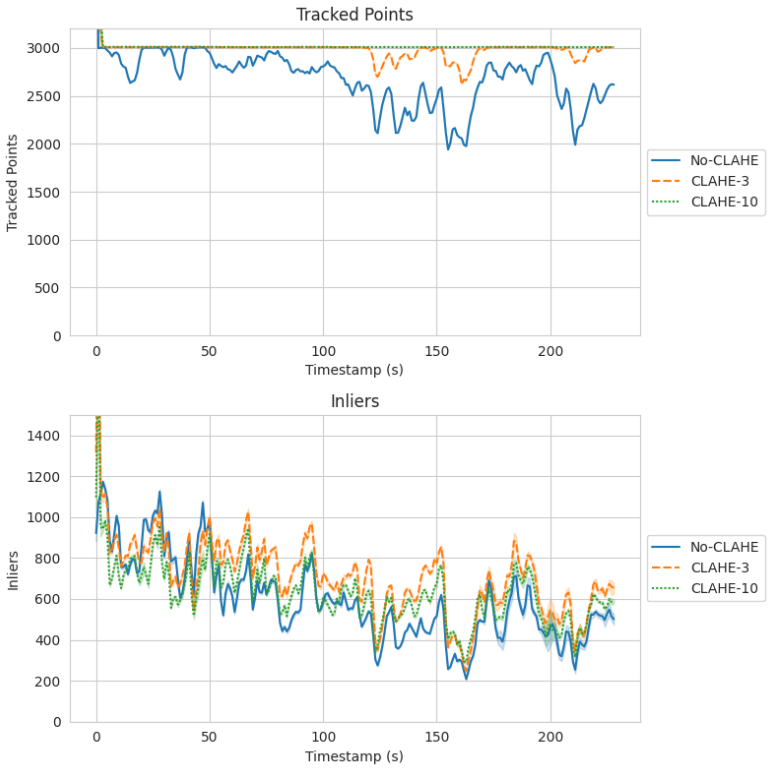


Figure 3.12: Harbor 1

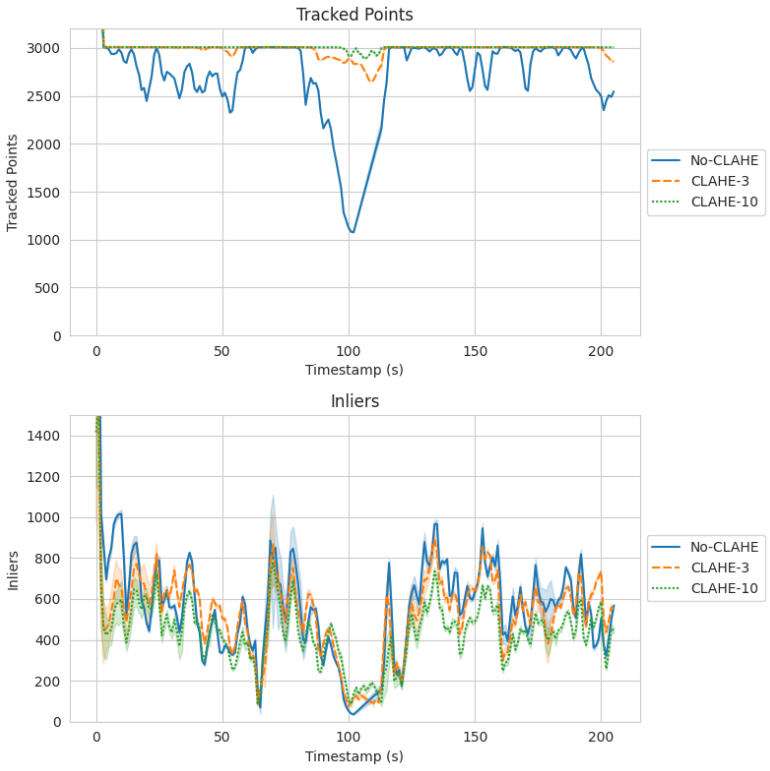


Figure 3.13: Harbor 4

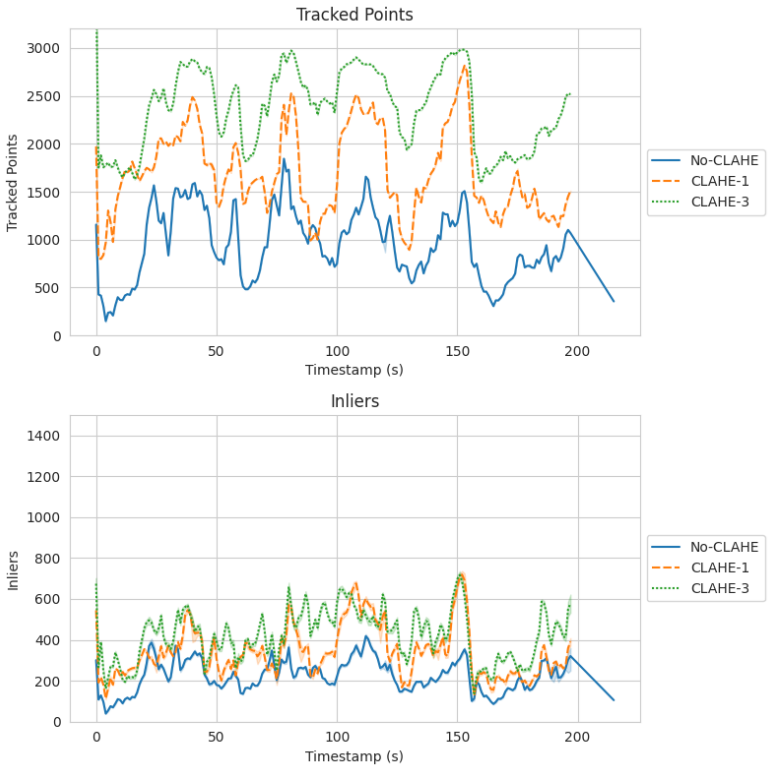


Figure 3.14: UwUE 1

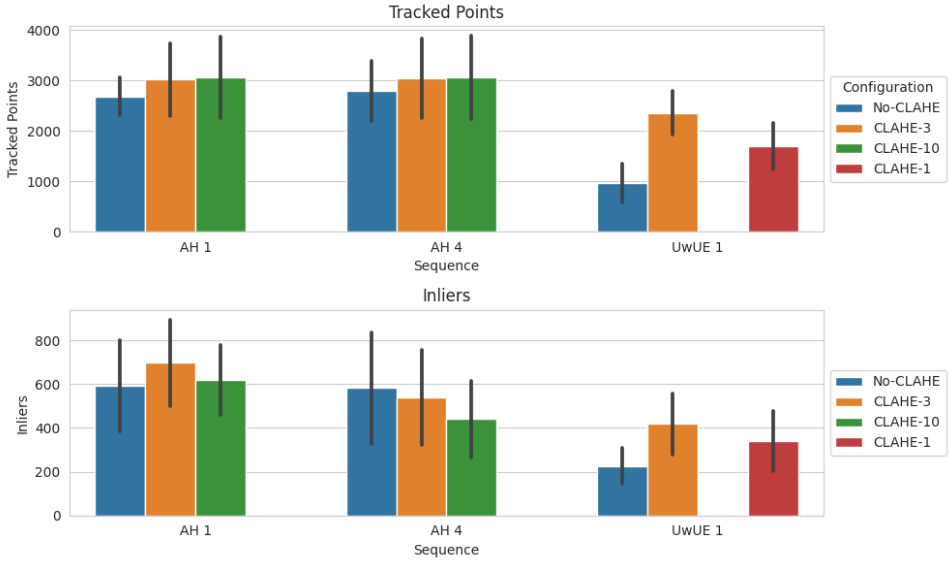


Figure 3.15: Mean tracked points and inliers per dataset with standard deviation (black lines)

We can see based on Figures 3.12 to 3.14 that increased values of CLAHE are correlated with a greater number of tracked points. Harbor 1 and 4 consistently maintain the required number of tracked points. The spikes at the very start of the

trajectories indicate the initialization of the map and is part of the usual initialization procedure.

Despite the increase in tracked points, higher CLAHE values do not directly correlate to a higher number of inliers. In fact, we can see on Figure 3.12 that CLAHE-10 caused a decrease in the number of inliers. Figure 3.13 also shows that both CLAHE configurations actually performed slightly worse than the base configuration. The only period where they performed better was in the 100-second mark, but not by much.

On the other hand, we can see a significant improvement in the UwUE sequence the higher the CLAHE value was, even though the number of inliers are still relatively low.

Figure 3.15 summarizes the results for each dataset, showcasing the mean number of tracked points and inliers as well as their standard deviation. We can easily see here that the number of tracked points greatly benefit from CLAHE. Performance varies depending on the dataset if we look at the number of inliers.

3.8.2 Trajectory analysis

This section presents the analysis of the trajectories for the proposed SLAM configurations. First, we analyze the estimated global trajectories of each configuration by comparing them to the relevant ground truth values. We also test the configurations for drift, and analyze their initialization procedures.

The majority of the data gathered here is from the Harbor dataset. We do however include UwUE sequence 1 in Table 3.10, which shows the absolute trajectory error for each configuration. It is not included in any further analysis, the reason for which is given in Section 3.8.2.

Absolute trajectory error

The absolute trajectory error (ATE) provides a quantitative metric for evaluating the overall accuracy of the trajectory.

Each trajectory is compared with its corresponding ground truth and aligned using Umeyama alignment (Umeyama, 1991), but without scaling the trajectory. The practical process of analysing these trajectories is achieved with the help of the Evo Python package (Grupp, 2017). Table 3.10 shows the median root mean squared error

(RMSE) calculated for each configuration. Values marked with *X* indicate that the configuration was unable to successfully complete the sequence.

We have also included the results from UW-VO (Ferrera, 2019) in the AQUALOC dataset. However, note that their values are based on an older version of the ground truth. It has since been adjusted to be more accurate. As such, we cannot make a proper one-to-one comparison.

The trajectories have also been plotted for each sequence, which can be seen from Figures 3.16 to 3.22 for the Harbor dataset. Figure 3.23 shows the trajectory for UwUE sequence 1. These figures also include ORB-SLAM's base monocular configuration.

		UW-VO		ORB-SLAM				
Seq	Length [m]	VP	VIP	VP-1A	VP-1B	VP-2A	VP-2B	VIP
H1	36.725	0.490	0.420	0.371	**0.166**	0.801	0.723	0.234
H2	68.920	**0.360**	0.370	0.583	0.560	0.444	0.526	0.489
H3	23.025	0.250	0.260	0.138	0.148	0.235	0.222	**0.079**
H4	47.645	X	**1.560**	X	X	X	X	X
H5	27.045	0.130	**0.090**	0.515	0.592	1.279	1.107	0.369
H6	11.711	**0.040**	0.060	0.645	0.224	0.395	0.115	0.069
H7	23.868	X	**1.160**	X	X	X	X	1.612
U1	155.022	-	-	5.081	4.595	4.956	**4.570**	5.207

Table 3.10: Absolute trajectory error (RMSE) [m] on AQUALOC harbor (H) dataset as well as sequence 1 of UwUE (U)

We can see based on Table 3.10 that the proposed configurations are comparable, if not slightly underperforming compared to UW-VO. Again, their values reflect an older version of the ground truth and therefore cannot be compared directly.

None of the visual-pressure configurations were able to successfully complete Harbor sequences 4 and 7. These sequences are the most challenging in the Harbor dataset, as they contain sections with very little visual information. Both sequences also contain collisions, which frequently cause the systems to fail.

The visual-inertial-pressure configuration shows better overall results compared to the visual-pressure configuration. Unfortunately, it was still unable to consistently complete sequence 4. Figure 3.19 shows the best trajectory it managed to complete,

which as we can see still diverges significantly from the ground truth. The configuration also diverges by a considerable margin in Harbor 7, but showed more consistent performance here than in Harbor 4.

The results for UwUE sequence 1 show very high values for the ATE. However, the issues here are more than likely due to the sensor measurements not being synchronized. The values are therefore not indicative of the system's performance for any configuration. We have decided to keep the result for the ATE here, but further analysis in this section no longer uses UwUE sequence 1.

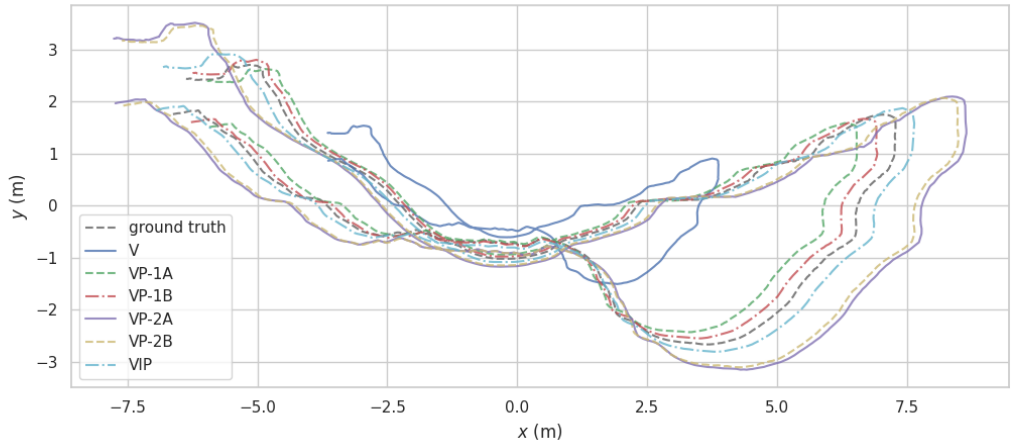


Figure 3.16: Harbor 1

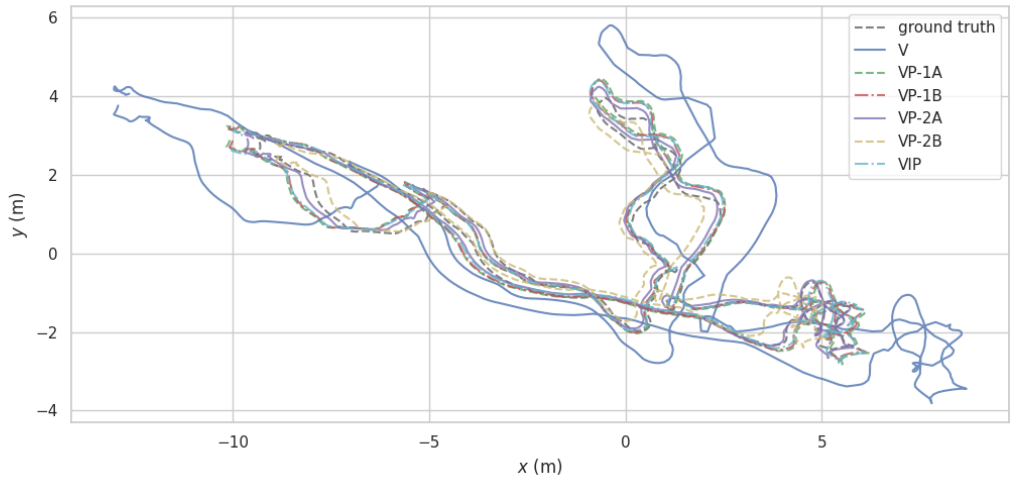


Figure 3.17: Harbor 2

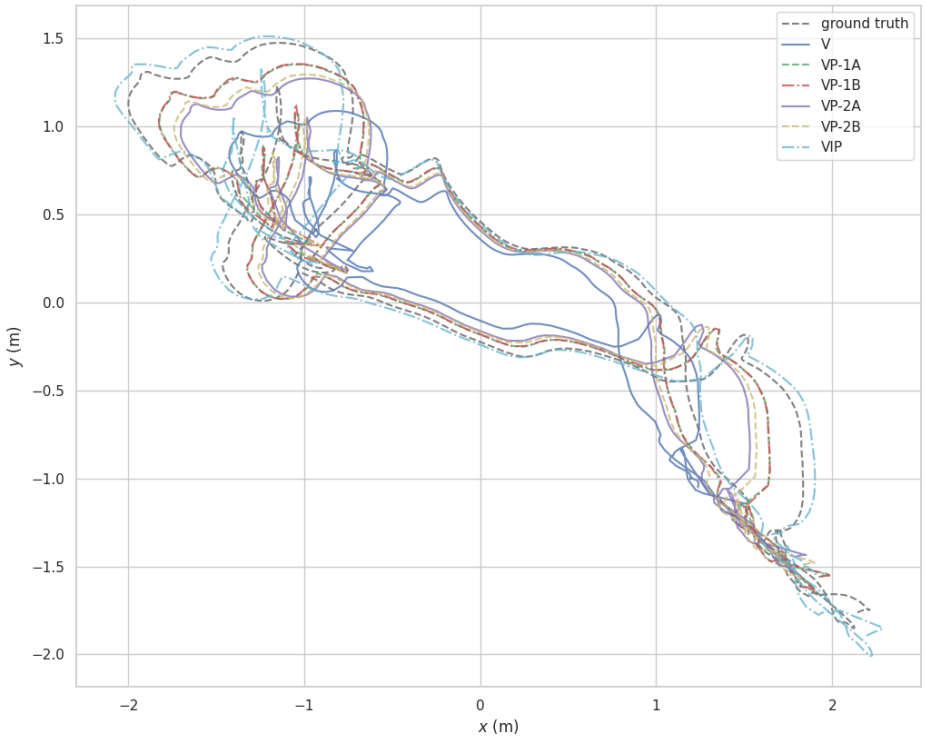


Figure 3.18: Harbor 3

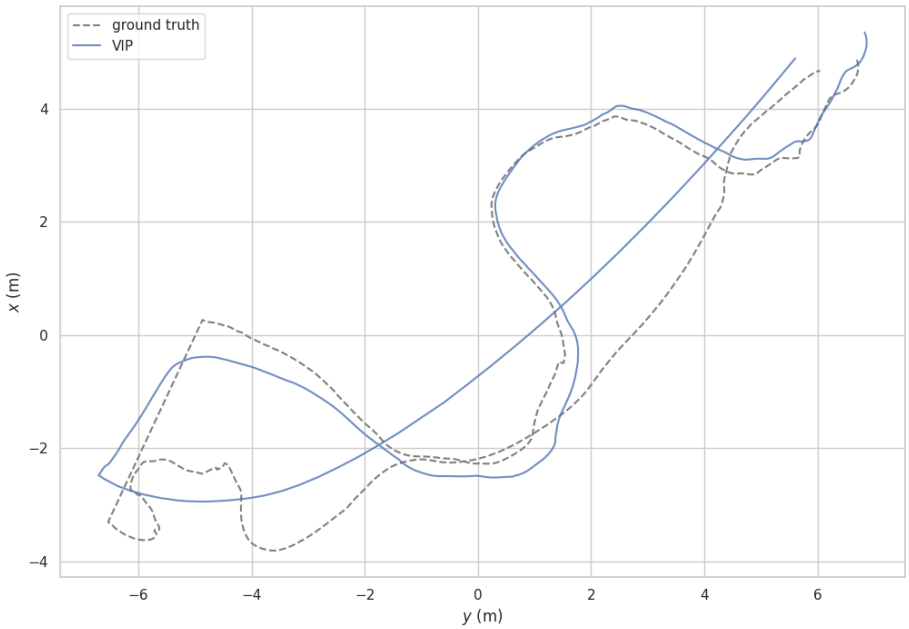


Figure 3.19: Harbor 4. Note that the trajectory shown here is the best trajectory out of 10 runs.

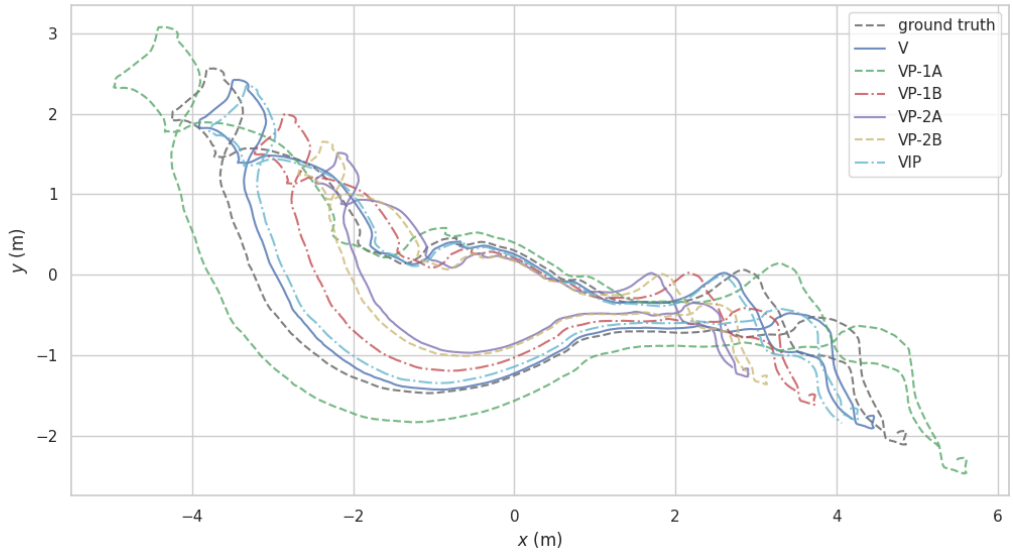


Figure 3.20: Harbor 5

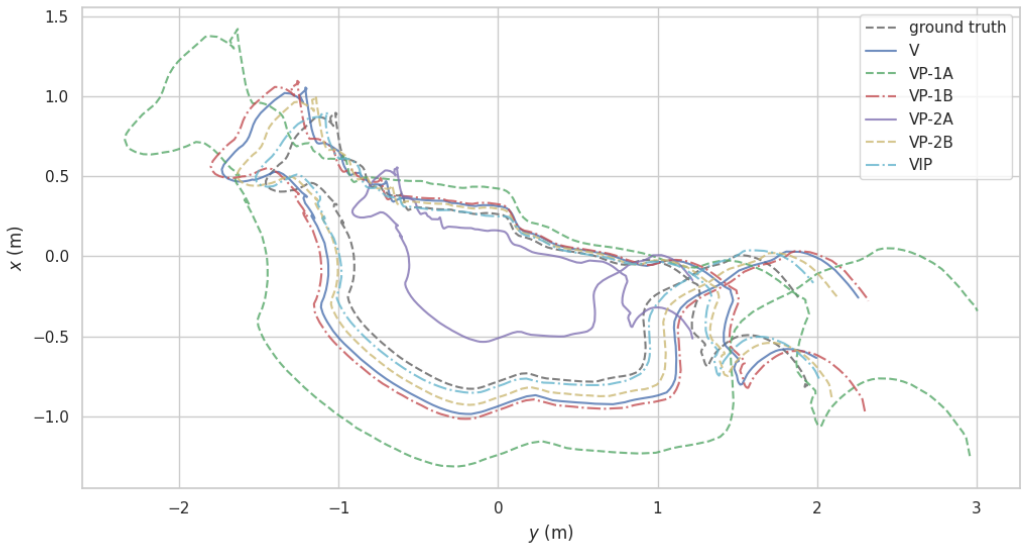


Figure 3.21: Harbor 6

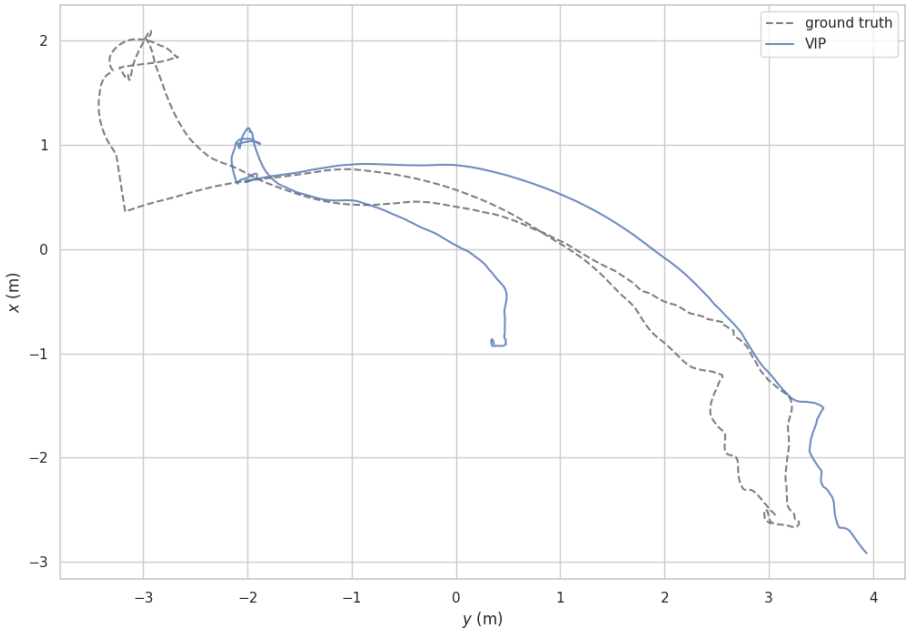


Figure 3.22: Harbor 7

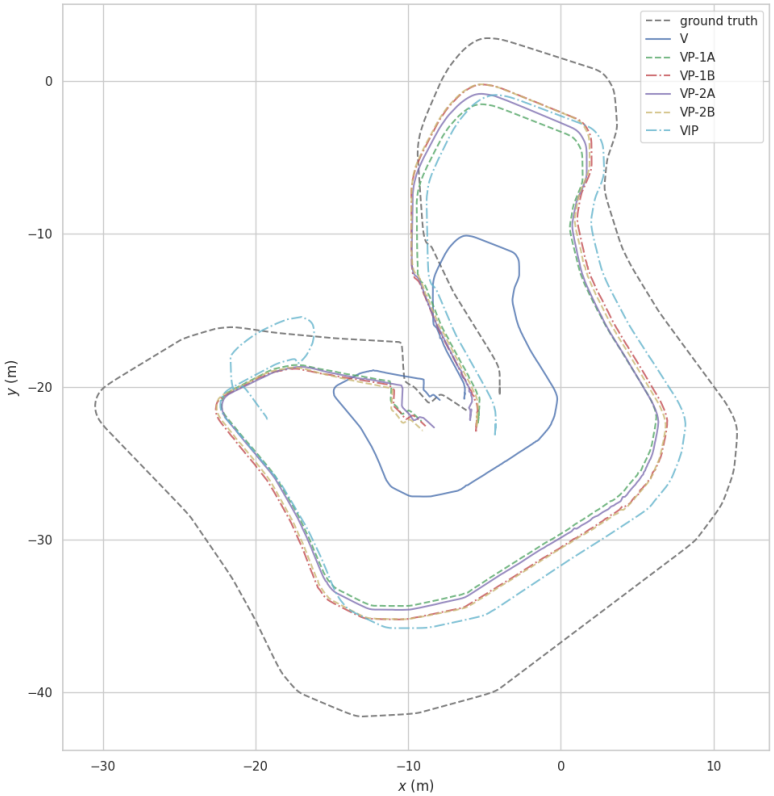


Figure 3.23: UwUE 1

Since the values presented in Table 3.10 usually increases according to the length of the trajectory, it can be difficult to compare the error of each configuration directly across all sequences. Figure 3.24 shows the mean APE for all configurations as a percentage relative to the trajectory length. We do not include the values for Harbor 4 and 7, as their errors are too high. From this, we can better compare how each system performs relative to each sequence. For instance, while it seemed at first that all configurations had a high ATE for Harbor 2, we can actually see here that they all achieved relatively good results compared to the other sequences.

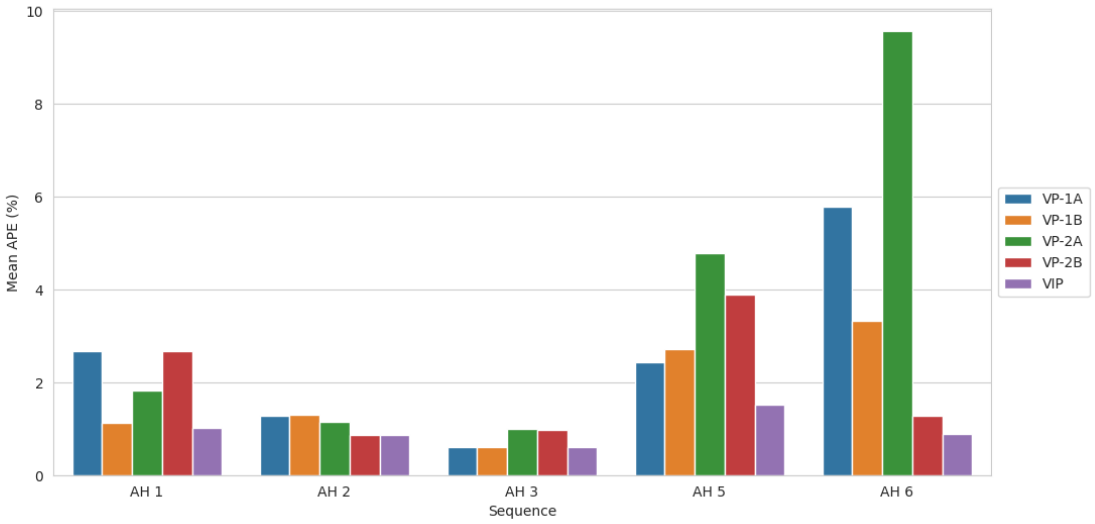


Figure 3.24: Mean APE as a percentage of trajectory length [%]

Relative pose error

The relative pose error (RPE) provides information on how the trajectory drifts over time by comparing the motion between poses. It is also a useful metric to measure the system when it is operating normally. Unlike the ATE, it is more useful to evaluate the RPE by scaling up the trajectory. This allows for a more direct comparison between configurations. This also means that the initialization procedure is not relevant in this examination. Since the different visual-pressure configurations only differ in their initialization procedure, their normal operations will produce the same result. Therefore, we choose only one of them for comparison. Specifically, we choose the values for configuration VP-1B, which, for this section, we will simply refer to as VP.

We include the base visual configuration in our comparison as well. Table 3.11 shows the mean RPE for each sequence, including its standard deviation. We use centimeters due to the small values.

	V		VP		VIP	
Seq	Mean	Std.	Mean	Std.	Mean	Std.
H1	2.4	1.7	1.9	0.1	17.1	0.6
H2	9.5	0.3	9.5	0.2	19.3	0.3
H3	1.3	0.04	1.3	0.04	11.5	0.5
H4	-	-	-	-	18.7	2.8
H5	1.9	0.04	2.0	0.1	9.4	2.7
H6	1.1	0.2	1.0	0.03	6.8	0.1
H7	-	-	-	-	10.5	1.4
Avg	3.24	0.45	**3.14**	**0.094**	13.33	1.2

Table 3.11: RPE for all sequences. All units are measured in centimeters [cm]

We can see that both the base monocular and visual-pressure configurations exhibit very little drift. On the other hand, the visual-inertial-pressure configuration shows much higher drift than the others, which is a somewhat surprising result. We discuss this further in the discussion section.

Initialization

Here, we analyze the results of the different initialization procedures. We look at the accuracy in scale, alignment, and initialization time. The scale is the most important factor among these, as it has the greatest impact on the accuracy of the trajectory. However, since the project uses a sensor that relies on accurate alignment, the rotation of the trajectory relative to the world reference frame plays a bigger role than in most other cases. Initialization time is the least important in this context and is most useful in real-time operations, where faster initialization leads to useful localization and mapping information more quickly. In the following sections, each metric is presented in a table containing the mean and standard deviation for each configuration, as measured for each sequence. We calculate each metric by scaling and aligning the trajectory with the ground truth using Umeyama alignment.

Scale The scale error is measured as a percentage of the actual scale, the results of which can be found in Table 3.12. Looking at the visual-pressure configurations, we can see that using depth measurements alone is generally good enough to recover the scale, although their performance varies considerably.

For the visual-inertial-pressure configuration, we see that the initialization procedure is extremely effective at recovering scale, with an average error below 3%.

	VP-1A		VP-1B		VP-2A		VP-2B		VIP	
Seq	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
1	20.80	26.75	8.66	10.27	14.62	9.93	9.40	25.22	1.74	11.49
2	6.40	18.94	6.43	20.47	6.98	17.43	7.52	6.18	0.23	10.68
3	8.82	1.73	8.98	1.88	14.62	3.04	14.21	1.61	3.40	12.28
5	4.57	23.52	6.16	25.83	41.43	7.78	33.85	4.46	3.66	34.94
6	54.67	51.35	34.18	37.45	70.42	170.85	8.89	13.55	4.31	10.67
Ave.	19.06	24.46	**12.88**	**19.18**	29.62	41.81	14.78	10.21	**2.67**	**16.01**

Table 3.12: Scale error as percentage of actual scale [%]

Alignment The rotation error measures the difference in alignment between the estimated trajectory and the ground truth. The values shown in Table 3.13 show the normalized pitch and roll angles as a single value. We do not take yaw into account, as it is always arbitrarily set during initialization. The only reliable way of recovering the yaw is if the system has some way to measure the global heading, such as with a compass. We show the values in degrees as it is easier to interpret.

The values do not follow quite the same trend as the scale. The overall results are decent, with all configurations able to recover the rotation within 10 degrees of the true alignment.

Unsurprisingly, the visual-inertial-pressure configuration shows extremely good results for recovering the alignment. The procedure is very accurate and reliable, highlighting how useful the IMU is for calculating rotation.

Initialization time The initialization time is measured from when the system first creates a map until the initialization procedure finishes. The time is measured in seconds, and the results can be seen in Table 3.14.

	VP-1A		VP-1B		VP-2A		VP-2B		VIP	
Seq	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
1	20.82	13.31	14.25	2.10	10.48	2.51	20.59	20.25	2.05	0.49
2	7.50	11.59	6.10	8.48	8.56	10.70	4.53	1.66	0.57	0.54
3	1.89	1.17	5.56	1.89	5.43	0.95	6.27	1.56	1.02	0.10
5	13.47	2.28	12.36	3.70	11.48	3.97	13.00	2.82	1.00	0.62
6	3.27	1.02	2.45	0.98	6.97	6.62	2.91	1.80	1.05	0.16
Ave.	9.40	5.88	**8.14**	**3.43**	8.58	4.95	9.46	5.62	**1.14**	**0.38**

Table 3.13: Rotation error in degrees [°]

For the visual-pressure configurations, we see that the average initialization times are very similar across the board, with all of them managing to initialize in under 20 seconds. The A configurations have higher standard deviations, but this is expected as their criteria for a successful initialization are more dynamic compared to B.

The visual-inertial-pressure configuration has on average longer initialization times, which is expected given the increased number of steps in the procedure.

	VP-1A		VP-1B		VP-2A		VP-2B		VIP	
Seq	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
1	18.85	5.85	16.31	0.79	16.06	4.86	16.37	1.06	30.05	2.44
2	18.87	1.56	15.41	0.34	19.95	4.33	15.64	0.53	17.78	0.76
3	13.50	0.31	15.77	0.21	17.65	0.49	16.01	0.42	25.91	0.52
5	14.58	0.85	15.84	0.66	17.66	1.11	15.48	0.37	20.69	4.22
6	14.08	0.90	17.56	1.43	18.61	9.46	17.93	1.46	42.42	5.74
Ave.	**16.00**	**1.89**	16.18	0.69	17.99	4.05	16.29	0.77	27.37	2.73

Table 3.14: Initialization time in seconds [s]

3.8.3 Loop closure

In this section, we analyze the results of the custom ORB vocabularies. The ORB vocabulary is mostly relevant when querying the database for matching keyframes. There are several other geometric verification steps that are unaffected by the change of vocabulary, which have not been included here. Our analysis simply looks for a

correlation between the number of BoW matches and loop closures found.

We conduct the tests on Harbor sequences 1 and 5. These two sequences triggered loop closures most often and have therefore been chosen as candidates for testing, although loop closure across all datasets we have tested on has generally been rare.

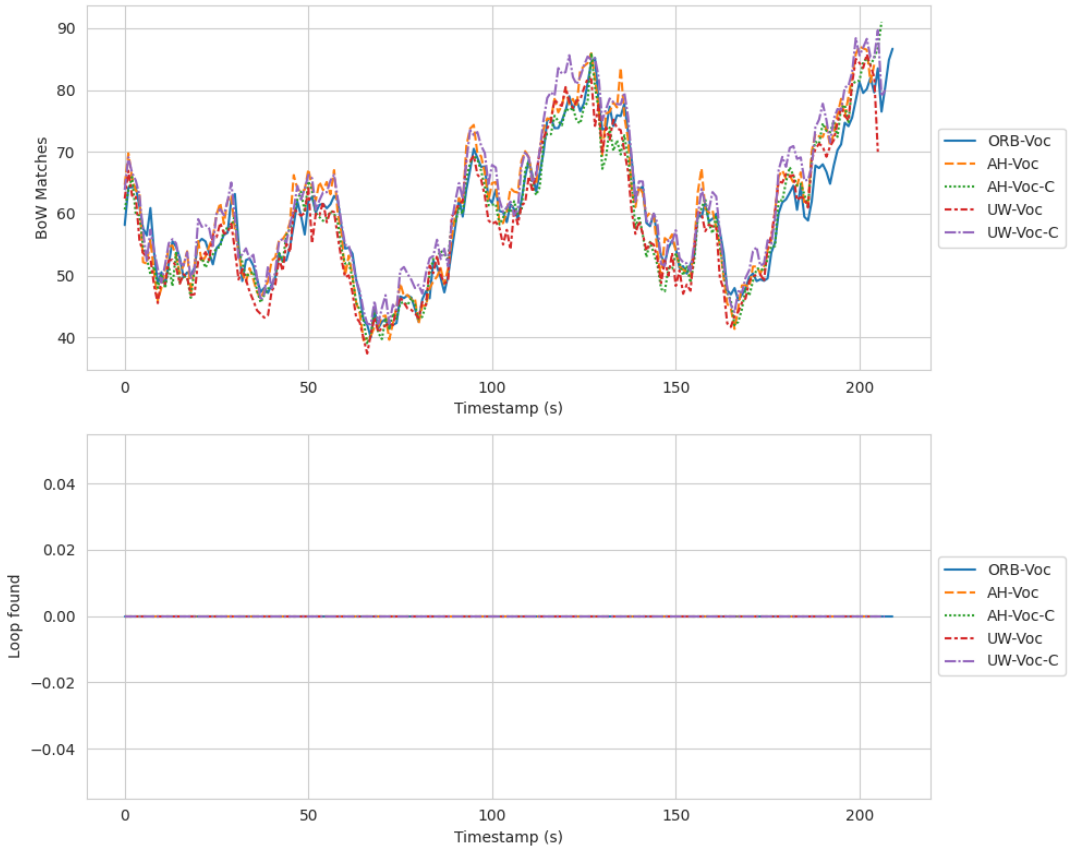


Figure 3.25: Harbor 1

Figures 3.25 and 3.26 plot the values of the number of BoW matches and loop closures found over the course of the trajectory. Looking at the number of BoW matches, it is difficult to see any noticeable improvement in the number of matches for each configuration. UW-Voc is the only configuration that shows a mild increase in BoW matches, although this is only for Harbor 1.

The system did not manage to find a loop closure for Harbor 1 over the course of 10 runs. We can see that the number of BoW matches spikes when it returns to its starting position but does not trigger a loop closure. In Harbor 5, the system is able to recognize the loop when returning. Here, we see a noticeable improvement in the number of loop closures found for the AH-Voc configuration.

Looking at Figure 3.27a, we can more easily see that UW-Voc-C does detect more

BoW matches, but only for Harbor 1. In Harbor 5, all other configurations actually show fewer matches. Despite this, AH-Voc managed to find the most loop closures.

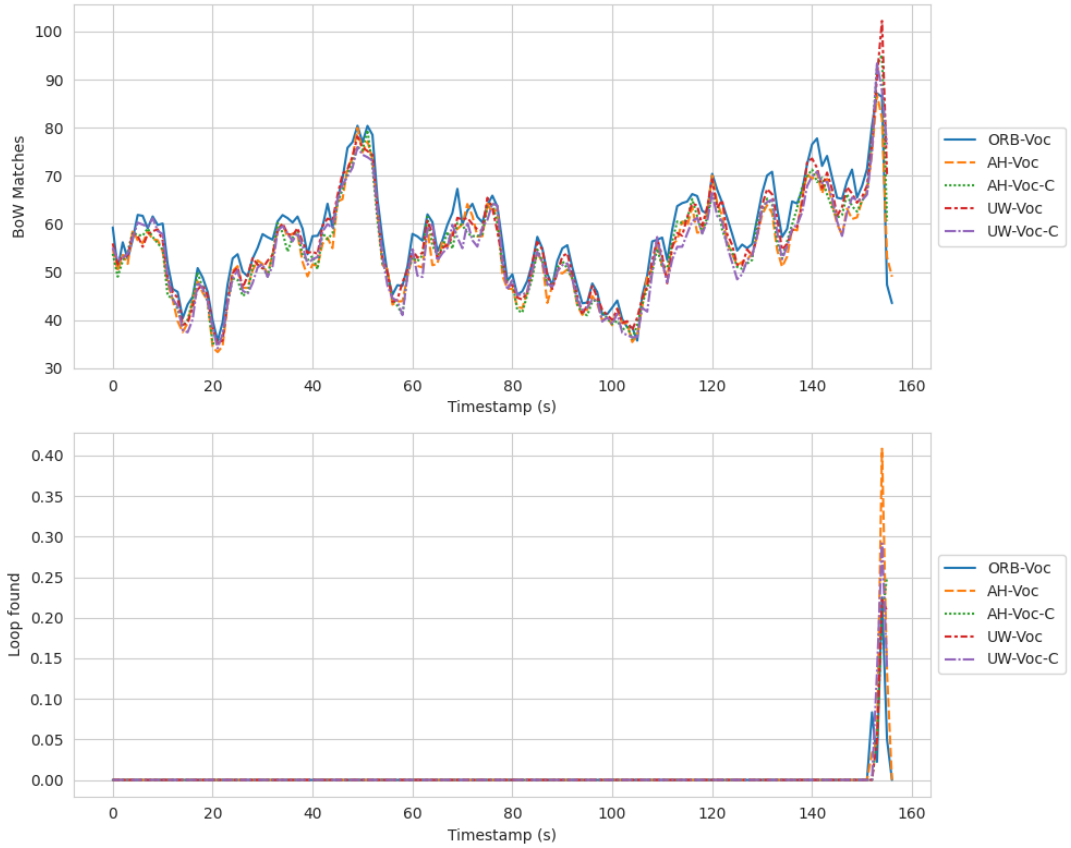
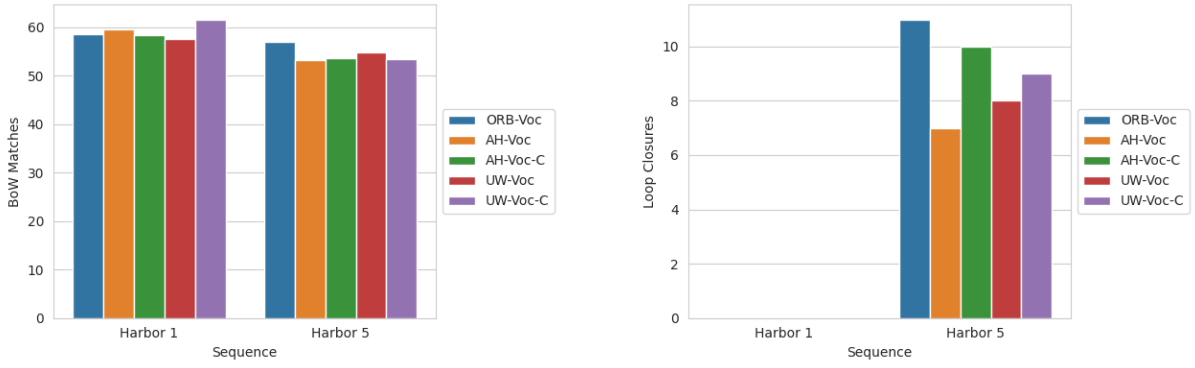


Figure 3.26: Harbor 5



(a) Mean number of BoW matches

(b) Number of loops found

Figure 3.27: summary of results for each trajectory

3.9 Discussion

In this section, we discuss the results of the previous section. We also provide here a brief section on the limitations of the study as well as suggestions for further work.

3.9.1 Image preprocessing

The results show that applying CLAHE greatly correlates to a higher and more stable number of tracked points. This is greatly beneficial in areas with low texture. However, the number of inliers did not benefit as much from higher CLAHE values. In fact, it was sometimes the opposite. One hypothesis regarding this is that the adaptive nature of the histogram equalization can make the same feature in consecutive images appear as being different due to a change of exposure. Another thing to point out is that applying CLAHE reduced the standard deviation for the inliers, which suggests that CLAHE is beneficial in that it provides a more consistent performance for feature recognition.

Based on the results, it is difficult to decide which configuration had the best performance. Rather, the data suggests that to what degree CLAHE should be applied is highly dependent on the environment where it is used. We can see that even for the same dataset, its performance does not clearly indicate that one should be used over another. The rule of thumb here seems to be to use as minimal amount of CLAHE as is feasible to increase the contrast of the image. It may also be interesting to dynamically change the clip limit of CLAHE in the system during operation.

It is also important to note that CLAHE will not help against highly dynamic environments. We briefly mentioned at the start of the results that we wanted to test the system on other datasets, but we found that it could not handle the visual disturbances.

3.9.2 Trajectory analysis

Based on the ATE results in Section 3.8.2, we can note that fusing depth measurements does aid the system in reducing the error by correctly scaling the trajectory. In our discussion, we will look at the visual-pressure and visual-inertial-pressure configurations separately.

Visual-pressure configuration

We can see how much influence the initialization procedures have on the accuracy of the trajectory by looking at the results of the different visual-pressure configurations. Surprisingly, the depth average-based strategies (VP-1X) outperforms the optimization-based strategies (VP-2X) most of the time. The B-configurations also exhibit better overall results, implying that scale and rotation calculations are better solved at set intervals instead of an iterative process. The configurations using iteration-based initialization techniques struggle particularly with Harbor 6. This sequence contains very little motion along the z-axis, which we believe is the reason for the poorer performance here. VP-1B showcases the best overall results in trajectory estimation for the visual-pressure configuration, although VP-2B does not fall far behind either.

The results of the ATE are further supported when we look at the RPE in Section 3.8.2 as well as the initialization results in Section 3.8.2. First, we analyze more deeply the results for RPE.

The changes to the pose estimation and local bundle adjustment algorithms did not change the RPE by a noticeable amount, except for the decrease in the standard deviation. This indicates that the depth measurements help constrain the trajectory even if the contribution is small. However, this is because the base monocular configuration already shows excellent performance with very little drift. We believe that the values are low due to the slow movement of the ROV. Because of this, the results of the RPE analysis might not properly represent the performance of the system if we were to generalize it. Monocular configurations usually tend to suffer from scale drift over time, especially during rotational motions. Testing for datasets with faster motions should therefore be done to better verify if the depth measurements can properly constrain the drift in scale.

When looking at the initialization procedures, we can clearly see that the error in scale directly correlates to the error in the trajectory. Set interval strategies (VP-XB) are better than the iterative strategies in almost all cases, and VP-1B shows the most accurate scale estimates in the visual-pressure category. VP-2B however does not fall far behind, and shows a more consistent initialization procedure with less overall standard deviation. In fact, it was the only strategy that managed to recover the scale

within 10% of the true value for Harbor 6. We note that this sequence has very little initial motion along the depth-aligned axis, which implies that this strategy is better suited for these types of situations compared to the others. VP-2A has by the worst results and does not generalize well to all sequences, although we are unsure as to the specific reason for its inconsistent behavior.

All visual-pressure configurations were generally able to calculate the alignment reliably. We can however notice that they all struggle with Harbor 1. This is likely due to the fact that in this sequence, the ROV has limited motions along one of the translational axes until much later on. The initialization procedure is therefore unable to correct for this. Harbor 5 also has similar issues. VP-1B again shows the lowest error, but not by a considerable amount compared to the other configurations. Since the trend does not quite follow the same pattern as the scale or ATE, we can see that recovering the alignment does not play as significant a factor in the accuracy of the trajectory.

For the initialization times, all configurations have very similar values. VP-1A has the shortest initialization period, although the other configurations are not too far off. This indicates that all configurations are able to reliably finish their initialization within the span of the trajectory. The consistent initialization times are also a good sign if the user were to conduct live testing, as they are in almost all cases guaranteed to get nominal localization and mapping information within an expected amount of time.

Visual-inertial-pressure configuration

The visual-inertial-pressure configuration is noticeably better than the visual-pressure configurations in that its results are more consistent. However, this configuration was still unable to complete Harbor 4 and 7, meaning that the system is still not sufficiently robust towards short-term visual loss. This suggests that the modified pose estimation step described in Section 3.7.2 is not sufficient to counteract the accumulation of bias in the system. To a certain extent, this is not surprising given that the modification simply scales the entire pose delta based on the ratio between the depth and z-axis translation. Alternative methods must therefore be explored to counteract this, although this may be difficult without adding new sensor modalities.

By examining its RPE and initialization results, we see that its main weakness

is with regards to drift in trajectory. Surprisingly, the visual-inertial-pressure configuration exhibits much higher drift than the others. There are two likely causes for this. The first is that the movements are simply too slow, which makes it difficult for the system to accurately calculate the inertial parameters. The second possible cause might be the local bundle adjustment algorithm used. Although it has been proven to be more efficient and demonstrated comparable accuracy in indoor environments, it might not be as well suited for the underwater domain.

Looking at initialization, the configuration is extremely reliable at recovering scale and alignment. The initialization time is longer than the visual-pressure configuration, but this is expected due to its extensive procedure. However, the long initialization time of 42 seconds for Harbor 6 is still unusual. The trajectory only has a duration of 113 seconds, which means that it uses over a third of the trajectory to fully initialize. The sequence contains very slow motions overall, even compared to the rest of the dataset. We can reasonably assume that this is the main issue behind the long initialization period.

Map quality

We provide a brief qualitative analysis of the map quality, which is heavily tied to the accuracy of the trajectory.

For the AQUALOC Harbor dataset, the map is surprisingly detailed despite the fact that ORB-SLAM creates a sparse map reconstruction. We believe that this is likely due to the slow motions of the ROV and the downward-facing camera, which allows for many of the same features to be re-observed (given that there are no visual occlusions). We can also notice that there are many stray map points, likely due to particles in the water that are mislabeled as features.

The maps of the base configuration and visual-pressure configuration are more consistent. This is also supported by the fact that they drift very slowly from the actual trajectory (section 3.8.2). Likewise, we can see that the drift in trajectory for the visual-inertial-pressure configuration is noticeable on the map. Figure 3.29 shows where the ROV revisits a previous location, but the map does not align with itself.

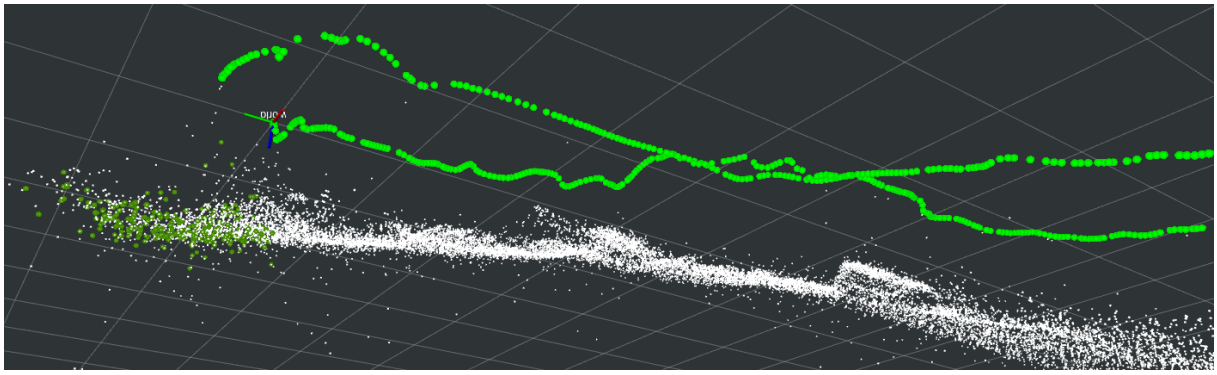


Figure 3.28: Side-view of the map produced by the base monocular configuration.

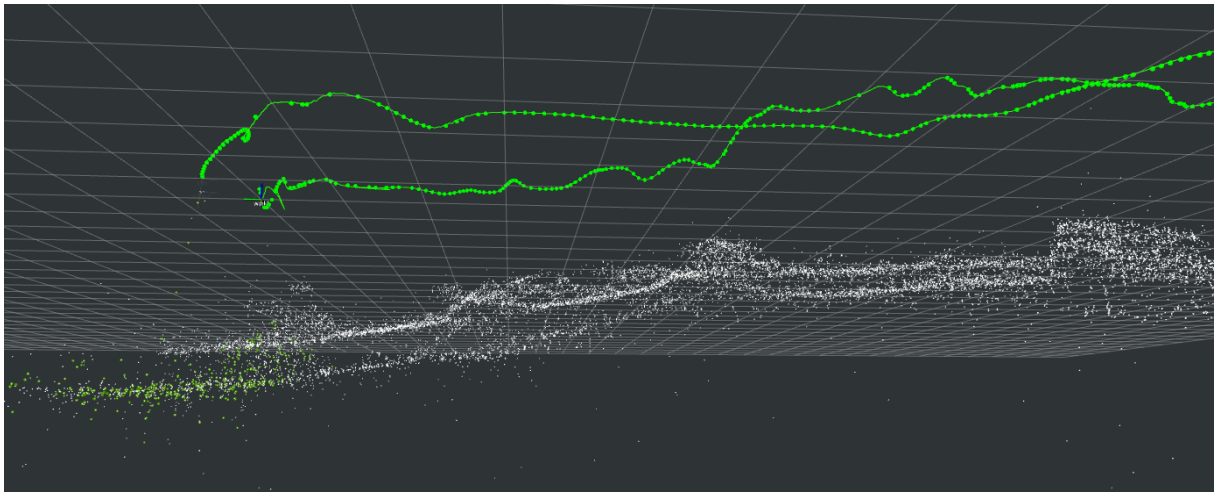


Figure 3.29: Side-view of the map produced by the visual-inertial-pressure configuration. Notice how the map does not overlap like in Figure 3.28.

Reliability

The system is generally reliable in performing localization and mapping. Occasionally, the system will lose track for no apparent reason, but we are unsure whether this is an issue with the system or the data that is being fed to it. Usually, it can recover from these anomalies. However, the visual-pressure configuration is less reliable in recovering from tracking loss midway through the session. Usually, it is unable to initialize successfully afterward, likely because it no longer has ideal motions for successful initialization. The visual-inertial-pressure configuration is better in that regard.

One thing to keep in mind with the visual-inertial-pressure configuration is that it is highly reliant on accurate timestamps between the visual and inertial data. The performance degrades considerably even if the offset is only in the tens of milliseconds. Practically, it makes it much more difficult to use this configuration for live testing without proper software or hardware synchronization.

Computation time

One metric that has not been quantitatively analyzed is the computation time for the different parts of the system. We remark that there is no noticeable decrease in the perceived performance of the system. However, a quantitative analysis should still be conducted on this at some point. The specific areas of interest include the impact of image preprocessing and the various optimization algorithms changed in the system.

Summary of trajectory analysis

The visual-pressure configuration shows good overall performance specifically due to the low amount of drift. The initialization procedures, while sufficient, still contain room for improvement. Scale is clearly the most important factor for initialization, and the different procedures have on average an error of 20%. We believe that this can be improved if the dataset contain the ideal motions for initialization as we described in Section 3.6.4. However, the system should still be sufficiently robust that it can handle these non-ideal situations.

The visual-inertial-pressure configuration show the opposite results, exhibiting highly accurate initialization estimates but suffering from high drift. This means that

the pose estimations and local bundle adjustment procedures need to be revisited and improved upon. As it stands, the ideal configuration seems to be using the inertial measurements only during initialization and letting the system operate in its base configuration afterwards.

3.9.3 Loop closure analysis

The results on our loop closure analysis show that there is no inherent improvement to the number of BoW matches between the different custom configurations and the base configuration. Although the number of loops might have improved slightly with one of the configurations, namely AH-Voc, this improvement seems to fall within the margin of error. We can see that all other configurations only differ by 1 or 2 loops. Therefore, it might not be advantageous to invest time and effort into creating a new vocabulary based on these results alone.

To a certain extent, these results are perhaps not that surprising. We can clearly see that the BoW matches spike whenever the system recognizes a previous location, which if we were to compare with the trajectory, lines up reasonably well on the timeline. However, BoW matches are prone to false positives, hence the reason for the geometric verification procedure. The geometric verification steps should therefore be further investigated to see if it can be better tailored to the underater domain.

3.9.4 Comparison to other SLAM systems

UW-VO is the best SLAM system to compare to ours and is the only one with data associated with a public dataset. In our comparisons, we see that UW-vo generally performed better, again keeping in mind that its results are based on an older ground truth configuration. However, we still believe that our project presents some interesting insights on other factors such as the different initialization procedures explored and the analysis on different vocabularies for place recognition.

Given that Svin2 supports the visual-inertial-pressure configuration, it would have been interesting to have added it to our comparison as well. However, setting it up took longer than expected and we had to drop it in favor of other developments in the project.

3.9.5 Limitations

One of the main limitations of this study is the datasets used for testing and experimentation. The majority of tests were done for the AQUALOC Harbor dataset which only model a limited area, and do not encompass the full range of possible environments and conditions of the underwater domain. We tested to a very limited extent the archaeological dataset but our system was unable to handle the challenges present in these sequences.

There are plenty other combinations of the different parts of our system that have not been explored exhaustively, such as testing CLAHE parameters over a much broader range of values, or testing and changing all the different initialization parameters. The DBoW2 vocabularies were also not tested much with other configurations of the system.

Further analysis of the results could also be done, such as presenting any possible correlations between the variables in image preprocessing and initialization. As we explained earlier, it would also be of interest to perform a quantitative analysis of the system's computational performance.

3.9.6 Suggestions for future work

The biggest challenge of the configurations presented still lies within the visual pipeline. Further work should be done to improve the performance of the visual component, either through image preprocessing techniques or a complete overhaul of the system. Deep learning techniques seem like a highly interesting avenue to explore to accomplish this.

More work should be done to improve the visual-pressure and visual-inertial-pressure configurations to address the main issues described in Section 3.9.2. Another alternative is to explore other sensor modalities that could be integrated to the system, although they must be chosen carefully if the goal is to support low-cost configurations.

The loop closure capabilities are perhaps the weakest link in this configuration. Further work should be done to improve the loop closure capabilities. Exploring other alternative methods such as the cluster-based loop closure approach presented

by (Negre et al., 2016) earlier could be interesting, although this would also require overhauling other parts of the system.

3.10 Conclusion

In this chapter, we presented the development of a visual-pressure and visual-inertial-pressure configuration using ORB-SLAM3 as our basis. We implemented changes to the visual component by adding image preprocessing. We modified the pose estimation and local bundle adjustment algorithms to include depth measurements in the tracking and local mapping threads respectively. We also presented several initialization procedures for the visual-pressure configuration. Finally, for the visual-pressure configuration, we created several ORB vocabularies trained in the underwater domain.

We modified the visual-inertial configuration of ORB-SLAM to create the visual-inertial-pressure configuration. Here, we also modified the visual component in the same manner, and also made changes to the tracking and local mapping threads respectively. We created a new initialization procedure for the visual-inertial-pressure configuration.

The results showed that these configurations were able to successfully adapt ORB-SLAM to work better in the underwater environment. Applying CLAHE helped increase the number of tracked points, although care must be taken as setting the value too high negatively impacts the number of inliers the system is able to find.

The depth measurements were sufficient to recover the scale, although more work is needed to refine the scale estimation for the visual-pressure configuration. On the other hand, the visual-inertial-pressure configuration showed remarkable results when recovering scale and aligning the trajectory, but suffered from a drift in the trajectory during normal operation.

The new underwater vocabularies did not show a noticeable improvement in the system's ability to form loop closures. Further work should be done to improve this part of the system.

There are several paths that can be taken for future work. Firstly, more tests should be conducted to verify the performance of the system in different environments, although we note that it currently cannot handle environments with too many dynamic disturbances. The visual component, as is the case with any visual SLAM algorithm, can be further improved to better handle the underwater domain, either through new image preprocessing techniques or a complete overhaul of the visual pipeline.

Deep learning techniques could be especially interesting here. New sensor modalities should also be considered. For this project, we wanted to make sure that the system is applicable to a wide variety of low-cost configurations. Those who wish to continue with the same line of reasoning should therefore be mindful on what other sensor modalities could be included. Finally, more work on the place recognition algorithm would greatly improve the performance of the system. Alternative place recognition systems should also be explored that may be better suited for the underwater domain.

Chapter 4

Project Conclusion

In this thesis, we presented two main contributions to the field of underwater robotics. The first contribution was the design, implementation, and evaluation of an underwater simulation environment for visual SLAM and other robotic applications. This environment was developed using Unreal Engine and Gazebo, focusing on generating visually realistic environments that closely resemble real-world underwater conditions. The second contribution involved the adaptation of ORB-SLAM3 for underwater environments using visual-pressure and visual-inertial-pressure configurations. This was achieved by modifying the visual component with image preprocessing, implementing changes to the pose estimation and local bundle adjustment algorithms, creating new vocabularies, and designing several initialization procedures for the visual-pressure configuration.

The results of our work show that both the simulation environment and the modified ORB-SLAM configurations are capable of providing valuable tools for research and development in the field of underwater robotics. The simulation environment offers realistic visuals and sensor data, while the adapted ORB-SLAM configurations demonstrate improved performance in the underwater domain.

However, there remain several challenges and opportunities for future work. For the simulation environment, usability remains a significant challenge, and further development should focus on streamlining the development pipeline and reducing computational demand. Perhaps the most important aspect that should be focused on first is synchronizing the sensor data properly, which would greatly increase the usefulness of the simulation. Moreover, expanding the evaluation to cover a wider

range of scenarios and features, implementing new environments, additional sensors, and a GUI would increase the value of the simulation for researchers and developers.

For the adapted ORB-SLAM configurations, further work is needed to refine the scale estimation for the visual-pressure configuration, address drift in the trajectory for the visual-inertial-pressure configuration, and improve the system's ability to form loop closures. Additionally, more tests should be conducted in different environments, and further improvements to the visual component should be explored, including the potential application of deep learning techniques.

We hope to make the realm of underwater robotics more accessible to the public. To that end, the implementations of the simulation and SLAM system have been made readily available online in the following github repositories (Tomter, 2023a, 2023b). Those who wish to continue the development of these systems are also welcome to do so.

Bibliography

Ahamed, J., Abas, P. E., & De Silva, L. (2019). Review of underwater image restoration algorithms. *IET Signal Processing*, *13*. https://doi.org/10.1049/iet-ipr.2019.0117

Anwar, S., Li, C., & Porikli, F. (2018). Deep underwater image enhancement. *arXiv preprint arXiv:1807.03528*.

Bailey, T., & Durrant-Whyte, H. (2006). Simultaneous localization and mapping (slam): Part i the essential algorithms. *IEEE Robotics and Automation Magazine*, *13*(2), 99–110.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: Speeded up robust features. In A. Leonardis, H. Bischof & A. Pinz (Eds.), *Computer vision – eccv 2006* (pp. 404–417). Springer Berlin Heidelberg.

Bellingmo, P. R. (2020). *Dp control system for blueye pioneer* (Master's thesis). NTNU.

Blanco-Claraco, J. L. (2022). A tutorial on $\mathbf{SE}(3)$ transformation parameterizations and on-manifold optimization.

BlueRobotics. (2023). Bluerov2 [Accessed: 08.05.2023]. https://bluerobotics.com/store/rov/bluerov2/

Calonder, M., Lepetit, V., Ozuysal, M., Trzcinski, T., Strecha, C., & Fua, P. (2012). Brief: Computing a local binary descriptor very fast. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *34*(7), 1281–1298. https://doi.org/10.1109/TPAMI.2011.222

Campos, C., Elvira, R., Rodríguez, J. J. G., Montiel, J. M. M., & Tardós, J. D. (2021). Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, *37*(6), 1874–1890. https://doi.org/10.1109/TRO.2021.3075644

Chaudhary, A., Mishra, R., Kalyan, B., & Chitre, M. (2021). Development of an underwater simulator using unity3d and robot operating system. *OCEANS*

2021: San Diego – Porto, 1–7. https://doi.org/10.23919/OCEANS44145.2021.9706012

Chen, E., & Guo, J. (2014). Real time map generation using sidescan sonar scanlines for unmanned underwater vehicles. *Ocean Engineering*, *91*, 252–262. https://doi.org/https://doi.org/10.1016/j.oceaneng.2014.09.017

Chen, L., Yang, A., Hu, H., & Naeem, W. (2020). Rbpf-msis: Toward rao-blackwellized particle filter slam for autonomous underwater vehicle with slow mechanical scanning imaging sonar. *IEEE Systems Journal*, *14*(3), 3301–3312. https://doi.org/10.1109/JSYST.2019.2938599

code-iai. (2023). Rosintegration plugin for unreal engine 4 [Accessed: 10.12.2022]. https://github.com/code-iai/ROSIntegration

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). CARLA: An open urban driving simulator. *Proceedings of the 1st Annual Conference on Robot Learning*, 1–16.

Engel, J., Koltun, V., & Cremers, D. (2018). Direct sparse odometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *40*(3), 611–625. https://doi.org/10.1109/TPAMI.2017.2658577

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *kdd*, *96*(34), 226–231.

Ferrera, M. (2019). *Monocular visual-inertial-pressure fusion for underwater localization and 3d mapping.* (Doctoral dissertation). Université Montpellier.

Ferrera, M., Creuze, V., Moras, J., & Trouvé-Peloux, P. (2019). Aqualoc: An underwater dataset for visual–inertial–pressure localization. *The International Journal of Robotics Research*, *38*(14), 1549–1559.

Field Robotics Lab. (2022). Project dave repository [Accessed: 15.10.2022]. https://github.com/Field-Robotics-Lab/dave

Forster, C., Zhang, Z., Gassner, M., Werlberger, M., & Scaramuzza, D. (2017). Svo: Semidirect visual odometry for monocular and multicamera systems. *IEEE Transactions on Robotics*, *33*(2), 249–265. https://doi.org/10.1109/TRO.2016.2623335

Fossen, T. I. (2011). *Handbook of marine craft hydrodynamics and motion control*. John Wiley & Sons.

Furrer, F., Burri, M., Achtelik, M., & Siegwart, R. (2016). Rotors—a modular gazebo mav simulator framework. In A. Koubaa (Ed.), *Robot operating system (ros): The complete reference (volume 1)* (pp. 595–625). Springer International Publishing. https://doi.org/10.1007/978-3-319-26054-9_23

Gálvez-López, D., & Tardos, J. D. (2012). Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, *28*(5), 1188–1197.

Georgiou, T., Liu, Y., Chen, W., & Lew, M. (2020). A survey of traditional and deep learning-based feature descriptors for high dimensional data in computer vision. *International Journal of Multimedia Information Retrieval*, *9*(3), 135–170.

Grupp, M. (2017). Evo: Python package for the evaluation of odometry and slam.

Harris, C., Stephens, M., et al. (1988). A combined corner and edge detector. *Alvey vision conference*, *15*(50), 10–5244.

He, B., Liang, Y., Feng, X., Nian, R., Yan, T., Li, M., & Zhang, S. (2012). Auv slam and experiments using a mechanical scanning forward-looking sonar. *Sensors*, *12*(7), 9386–9410. https://doi.org/10.3390/s120709386

Hidalgo, F., & Bräunl, T. (2015). Review of underwater slam techniques. *2015 6th International Conference on Automation, Robotics and Applications (ICARA)*, 306–311. https://doi.org/10.1109/ICARA.2015.7081165

Hidalgo Herencia, F. (2019). *Simultaneous localization and mapping in underwater robots* (Doctoral dissertation). The University of Western Australia. https://doi.org/10.26182/5c944ec20af50

Katara, P., Khanna, M., Nagar, H., & Panaiyappan, A. (2019). Open source simulator for unmanned underwater vehicles using ros and unity3d. *2019 IEEE Underwater Technology (UT)*, 1–7. https://doi.org/10.1109/UT.2019.8734309

Klein, G., & Murray, D. (2007). Parallel tracking and mapping for small ar workspaces. *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, 225–234. https://doi.org/10.1109/ISMAR.2007.4538852

Koenig, N., & Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, *3*, 2149–2154 vol.3. https://doi.org/10.1109/IROS.2004.1389727

Kümmerle, R., Grisetti, G., Strasdat, H., Konolige, K., & Burgard, W. (2011). G 2 o: A general framework for graph optimization. *2011 IEEE International Conference on Robotics and Automation*, 3607–3613.

Leutenegger, S., Chli, M., & Siegwart, R. Y. (2011). Brisk: Binary robust invariant scalable keypoints. *2011 International Conference on Computer Vision*, 2548–2555. https://doi.org/10.1109/ICCV.2011.6126542

Leutenegger, S., Forster, A., Furgale, P., Gohl, P., & Lynen, S. (2016). Okvis: Open keyframe-based visual-inertial slam (ros version).

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, *60*(2), 91–110.

Manhães, M. M. M., Scherer, S. A., Voss, M., Douat, L. R., & Rauschenbach, T. (2016). UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation. *OCEANS 2016 MTS/IEEE Monterey*. https://doi.org/10.1109/oceans.2016.7761080

Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., & von Stryk, O. (2012). Comprehensive simulation of quadrotor uavs using ros and gazebo. In I. Noda, N. Ando, D. Brugali & J. J. Kuffner (Eds.), *Simulation, modeling, and programming for autonomous robots* (pp. 400–411). Springer Berlin Heidelberg.

Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., et al. (2002). Fastslam: A factored solution to the simultaneous localization and mapping problem. *Aaai/iaai*, *593598*.

Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., et al. (2003). Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *IJCAI*, *3*, 1151–1156.

Mur-Artal, R., Montiel, J. M. M., & Tardós, J. D. (2015). Orb-slam: A versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, *31*(5), 1147–1163. https://doi.org/10.1109/TRO.2015.2463671

Mur-Artal, R., & Tardós, J. D. (2014). Fast relocalisation and loop closing in keyframe-based slam. *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 846–853.

Mur-Artal, R., & Tardós, J. D. (2017a). Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, *33*(5), 1255–1262. https://doi.org/10.1109/TRO.2017.2705103

Mur-Artal, R., & Tardós, J. D. (2017b). Visual-inertial monocular slam with map reuse. *IEEE Robotics and Automation Letters*, *2*(2), 796–803. https://doi.org/10.1109/LRA.2017.2653359

Negre, P. L., Bonin-Font, F., & Oliver, G. (2016). Cluster-based loop closing detection for underwater slam in feature-poor regions. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2589–2595. https://doi.org/10.1109/ICRA.2016.7487416

Negre Carrasco, P. L., Bonin-Font, F., & Oliver-Codina, G. (2016). Global image signature for visual loop-closure detection. *Autonomous Robots*, *40*, 1403–1417.

Newcombe, R. A., Lovegrove, S. J., & Davison, A. J. (2011). Dtam: Dense tracking and mapping in real-time. *2011 International Conference on Computer Vision*, 2320–2327. https://doi.org/10.1109/ICCV.2011.6126513

Noh, H., Araujo, A., Sim, J., Weyand, T., & Han, B. (2017). Large-scale image retrieval with attentive deep local features. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

NVIDIA Corporation. (2023a). Isaac sdk [Accessed: 30.04.2023]. https://developer.nvidia.com/isaac-sdk

NVIDIA Corporation. (2023b). Isaac sim [Accessed: 30.04.2023]. https://developer.nvidia.com/isaac-sim

Oceaneering. (2023). Enovus rov [Accessed: 08.05.2023]. https://www.oceaneering.com/rov-services/rov-systems/

Open Robotics. (2021). Gazebo [Accessed: 07.05.2023]. https://gazebosim.org/home

Open Robotics. (2023). Turtlebot [Accessed: 16.04.2023]. https://www.turtlebot.com/

OpenCV. (2023). Opencv color conversions [Accessed: 09.04.2023]. https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html

Paull, L., Saeedi, S., Seto, M., & Li, H. (2014). Auv navigation and localization: A review. *IEEE Journal of Oceanic Engineering*, *39*(1), 131–149. https://doi.org/10.1109/JOE.2013.2278891

Pizer, S. M., Amburn, E. P., Austin, J. D., Cromartie, R., Geselowitz, A., Greer, T., ter Haar Romeny, B., Zimmerman, J. B., & Zuiderveld, K. (1987). Adaptive histogram equalization and its variations. *Computer vision, graphics, and image processing*, *39*(3), 355–368.

Prats, M., Pérez, J., Fernández, J. J., & Sanz, P. J. (2012). An open source tool for simulation and supervision of underwater intervention missions. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2577–2582. https://doi.org/10.1109/IROS.2012.6385788

Qin, T., Li, P., & Shen, S. (2018). Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, *34*(4), 1004–1020. https://doi.org/10.1109/TRO.2018.2853729

Rahman, S., Karapetyan, N., Li, A. Q., & Rekleitis, I. (2018). A modular sensor suite for underwater reconstruction. *OCEANS 2018 MTS/IEEE Charleston*, 1–6. https://doi.org/10.1109/OCEANS.2018.8604819

Rahman, S., Li, A. Q., & Rekleitis, I. (2018). Sonar visual inertial slam of underwater structures. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 5190–5196. https://doi.org/10.1109/ICRA.2018.8460545

Rahman, S., Li, A. Q., & Rekleitis, I. (2019). Svin2: An underwater slam system using sonar, visual, inertial, and depth sensor. *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1861–1868. https://doi.org/10.1109/IROS40897.2019.8967703

Rehder, J., Nikolic, J., Schneider, T., Hinzmann, T., & Siegwart, R. (2016). Extending kalibr: Calibrating the extrinsics of multiple imus and of individual axes. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 4304–4311. https://doi.org/10.1109/ICRA.2016.7487628

Ribas, D., Ridao, P., Neira, J., & Tardos, J. D. (2006). Slam using an imaging sonar for partially structured underwater environments. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5040–5045. https://doi.org/10.1109/IROS.2006.282532

Rosten, E., & Drummond, T. (2006). Machine learning for high-speed corner detection. In A. Leonardis, H. Bischof & A. Pinz (Eds.), *Computer vision – eccv 2006* (pp. 430–443). Springer Berlin Heidelberg.

ROVOP. (2023). Schilling uhd generation iii [Accessed: 08.05.2023]. https://www.rovop.com/schilling-uhd-generation-iii/

Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. (2011). Orb: An efficient alternative to sift or surf. *2011 International Conference on Computer Vision*, 2564–2571. https://doi.org/10.1109/ICCV.2011.6126544

Schonberger, J. L., & Frahm, J.-M. (2016). Structure-from-motion revisited. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4104–4113.

Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. *Field and Service Robotics: Results of the 11th International Conference*, 621–635.

Shi, J., & Tomasi. (1994). Good features to track. *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 593–600. https://doi.org/10.1109/CVPR.1994.323794

Teixeira, B., Silva, H., Matos, A., & Silva, E. (2020). Deep learning for underwater visual odometry estimation. *IEEE Access*, *8*, 44687–44701. https://doi.org/10.1109/ACCESS.2020.2978406

Teledyne Marine. (2023). Slocum g3 glider [Accessed: 08.05.2023]. http://www.teledynemarine.com/slocum-glider?ProductLineID=14

thien94. (2023). Orb-slam3-ros [Accessed: 10.10.2022]. https://github.com/thien94/orb_slam3_ros

Tian, Y., Fan, B., & Wu, F. (2017). L2-net: Deep learning of discriminative patch descriptor in euclidean space. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Tomter, E. (2023a). Orb-slam vip [Accessed: 15.05.2023]. https://github.com/einatomter/orb_u

Tomter, E. (2023b). Underwater unreal simulation [Accessed: 15.05.2023]. https://github.com/einatomter/UwUESim

Umeyama, S. (1991). Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, *13*(04), 376–380.

Unity Technologies. (2022). Unity robotics hub [Accessed: 11.12.2022]. https://github.com/Unity-Technologies/Unity-Robotics-Hub

UZ-SLAMLab. (2022). Orb-slam3 github [Accessed: 01.04.2022]. https://github.com/UZ-SLAMLab/ORB_SLAM3

Wang, Y., Zhang, J., Cao, Y., & Wang, Z. (2017). A deep cnn method for underwater image enhancement. *2017 IEEE international conference on image processing (ICIP)*, 1382–1386.

Williams, B., Cummins, M., Neira, J., Newman, P., Reid, I., & Tardós, J. (2009). A comparison of loop closing techniques in monocular slam [Inside Data Association]. *Robotics and Autonomous Systems*, *57*(12), 1188–1197. https://doi.org/https://doi.org/10.1016/j.robot.2009.06.010

Appendix A

Underwater simulation

A.1 Code

We have included here the code that was used to create the simulation. These are located in the "Simulation" folder.

"1_frontend" contains the code that is needed for the front-end, namely Unreal Engine, including the fixed ROSIntegration plugin. All assets used for the simulation is also included here. We recommend following the installation guide from ROSIntegration to get this working (code-iai, 2023).

"2_backend" contains the code for the back-end, which mainly consists of the Project Dave package and the custom X3 ROV. These need to be built using ROS "catkin build". The packages are already located in a source folder, and can be simply be added to an existing ROS workspace. Project Dave does however require further prerequisites to run. We recommend checking the authors' repository for how to install it (Field Robotics Lab, 2022).

"3_datasets" contains the recorded sequences from the simulation. These are the processed rosbags to save on space. The data included here still contains synchronization issues, which the user should be mindful of.

A.2 ROV implementation

This section provides detailed information on the ROV implemented in Gazebo.

Table A.1 provides a brief overview of the physical properties of the X3 ROV. Table

A.2 shows the thruster parameters.

Physical properties	**Value**
Mass	9 kg
Length (x)	0.485 m
Width (y)	0.257 m
Height (z)	0.354 m

Inertia	**Value**
I_{xx}	0.1257 $kg\ m^2$
I_{yy}	0.2704 $kg\ m^2$
I_{zz}	0.2081 $kg\ m^2$

Table A.1: Simulated X3 properties

The following parameters are set for the thrusters of the X3.

Thruster property	**Value**
Dynamic time constant	0.1
Rotor constant	5×10^{-5}

Table A.2: Thruster properties

The sensors are provided in Table A.3. Table A.4 shows the camera parameters as implemented in Unreal Engine.

Sensor	**Noise**	**Update rate**	**Bias**
Pressure sensor	$1 \times 10^{-3}\ m$	10 Hz	
Accelerometer	$3.08 \times 10^{-2}\ m/s^2$	200 Hz	$6.8 \times 10^{-6}\ m/s^2$
Gyroscope	$3.08 \times 10^{-5}\ rad/s$	200 Hz	$1.7 \times 10^{-2}\ rad/s$
Magnetometer	$1 \times 10^{-3}\ m$	200 Hz	

Table A.3: Sensor specifications

Intrinsics	
f_x	480 px
f_y	480 px
c_x	480 px
c_y	270 px
Distortion	
k_1	0.0
k_2	0.0
p_1	0.0
p_2	0.0

Table A.4: Camera parameters

The hydrodynamic parameters of the ROV are given as shown below.

$$M_A = -\begin{bmatrix} 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 20 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix} \tag{A.1}$$

Next, we have the linear damping matrix of the system.

$$D(v) = -\begin{bmatrix} -25 & 0 & 0 & 0 & 0 & 0 \\ 0 & -30 & 0 & 0 & 0 & 0 \\ 0 & 0 & -28 & 0 & 0 & 0 \\ 0 & 0 & 0 & -10 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 & 0 \\ 0 & 0 & 0 & 0 & 0 & -10 \end{bmatrix} \tag{A.2}$$

Finally, we have the thruster allocation matrix (TAM). The TAM translates the desired output of the controller to the output of the thrusters. Project DAVE provides a helper function to calculate the TAM for you which has been used here. Some of the values have also been tweaked manually based on the drone's behavior in simulation.

$$TAM = \begin{bmatrix} -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.045 & -0.045 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.08 & -0.08 & 0.085 & 0 \end{bmatrix} \tag{A.3}$$

Appendix B

Underwater Visual SLAM

The appendix here includes the implemented SLAM system. The contents are in the "Underwater_slam" folder, which contains "1_orb_u" and "2_results".

"1_orb_u" is the implementation of the SLAM system shown in this paper. The system contains configurations to run the AQUALOC dataset, UwUE, and a few others as well. The visual-pressure configuration here uses initialization procedure "1-B".

"2_results" contains all the results gathered and used for analysis. This folder is mostly unstructured, but most of the data are stored either as csv files or in the TUM trajectory format, which is commonly used for recording the trajectory of a SLAM system.

We also wanted to include the data used for training the new ORB vocabularies, but the total dataset was too large (>60GB). We note however that the images we acquired are from publicly available datasets online, which we described in our project.