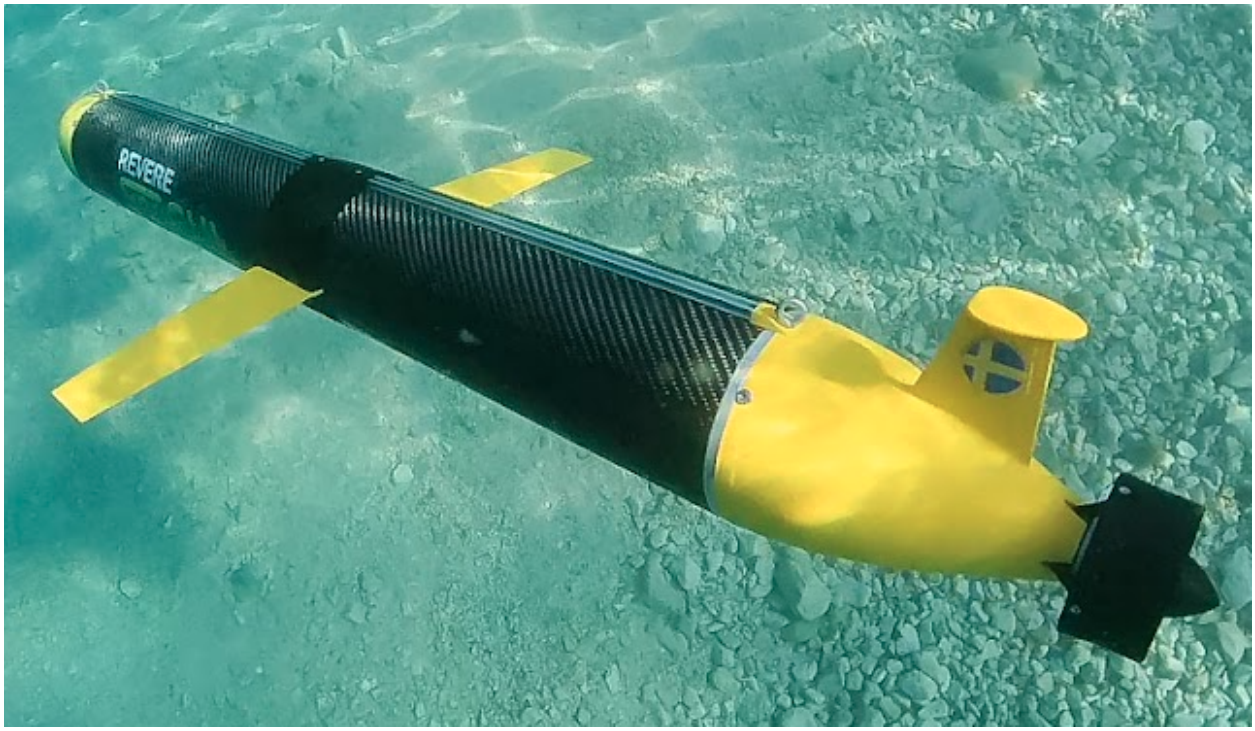
CHALMERS
UNIVERSITY OF TECHNOLOGY




# Development of a hardware in the loop rig for an autonomous underwater glider from a product development standpoint

Master's thesis in Product Development, MSc

Oskar Persson

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis in Product Development, MSc

# Development of a hardware in the loop rig for an autonomous underwater glider from a product development standpoint

Oskar Persson


CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanics and Maritime Sciences
Chalmers University of Technology
Gothenburg, Sweden 2026

Development of a hardware in the loop rig for an autonomous underwater glider from a product development standpoint
Oskar Persson



Supervisor: Tarun Kadri Sathiyan, Department of Mechanics and Maritime Sciences
Examiner: Ola Benderius, Department of Mechanics and Maritime Sciences

Master's Thesis 2026
Department of Mechanics and Maritime Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Gothenburg, Sweden 2026

Development of a hardware in the loop rig for an autonomous
underwater glider from a product development standpoint
Oskar Persson
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

# Abstract

This thesis presents the design and implementation of hardware in the loop (HIL) rig developed for the autonomous underwater glider (AUG) SeaGul. The rig simulates real dives by reading and responding to the actual hardware output of SeaGul with the use of load cells. During these simulated dives load cell data are sent to the simulation that calculates depth, distance, amount of water displaced, centre of gravity (CoG), pitch, and roll. Data calculated during the simulated dives are compared with results from real world dives. By analysing pitch and depth readings along with estimated timing between samples, the system approximates the glider's speed and distance covered in each dive phase. Providing a simplified simulation that can also get hardware responses in specific circumstances and edge cases. In addition, this work investigates the potential of HIL rigs as effective tools for the development and testing of AUG.

The results show that the HIL rig can find the CoG which is useful when ballasting SeaGul. It can also replicate core dive behaviours, such as response at depth, surfacing, and orientation changes. Furthermore, can the HIL rig consistently provide approximate estimations of dive speed, acceleration, pitch, roll, depth, and distance travelled but with limited precision. This highlights both the utility of the system and the need for improved simulation.

Despite sensitivity to disturbances and limitations due to incomplete real dive data, the HIL rig functions as intended, and all hardware components operate reliably. This demonstrates that the HIL rig is a valuable development and testing platform for SeaGul and potentially other AUG's. Supporting key behaviour insights, CoG calibration, and the ability to evaluate hardware responses under controlled edge case scenarios.

Keywords: Hardware in the loop (HIL), Autonomous underwater glider (AUG), System validation, Product development, SeaGul

# Preface

This report presents the outcome of the master's thesis project carried out in the Department of Mechanics and Maritime Sciences of Chalmers University of Technology during the spring of 2025.

# Acknowledgements

I express my sincere gratitude to my examiner, Ola Benderius, for his valuable feedback and insightful suggestions throughout this project and for giving me the opportunity to continue my work with SeaGul. Special thanks to my supervisor, Tarun Kadri Sathiyan, for his continuous support and encouragement during the thesis process. I would also like to acknowledge the contributions of the previous group whose work, help and input helped me move forward with this project.

Oskar Persson, Göteborg, August 2025

# List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

| | |
|---|---|
| AUG | Autonomous Underwater Glider |
| AUV | Autonomous Underwater Vehicle |
| BLE | Bluetooth Low Energy |
| CAV | Connected and Automated Vehicle |
| CoG | Center of Gravity |
| HIL | Hardware in the Loop |
| IMU | Inertial Measurement Unit |
| IoT | Internet of Things |
| LQR | Linear Quadratic Regulator |
| MQTT | Message Queuing Telemetry Transport |
| PD | Product Development |
| RAM | Random Access Memory |
| ROV | Remotely Operated Vehicle |
| UAV | Unmanned Aerial Vehicle |
| UNO | Arduino Uno |
| UUID | Universally Unique Identifier |
| UUV | Unmanned Underwater Vehicle |

# Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

## Constants

| | |
|---|---|
| $g$ | Gravitational acceleration, $9.82\,\mathrm{m/s^2}$ |
| $\rho_w$ | Freshwater density, $997\,\mathrm{kg/m^3}$ |
| $m_{SG}$ | Mass of SeaGul, $17.9\,\mathrm{kg}$ |
| $P_{atm}$ | Atmospheric pressure, $100000\,\mathrm{Pa}$ |
| $P_{max}$ | Maximum sensor pressure, $100\,\mathrm{bar}$ |
| $P_{min}$ | Minimum sensor pressure, $0\,\mathrm{bar}$ |
| $P_{mode}$ | Pressure offset (zero at 1 bar), 1 |
| $step_v$ | Incremental step, 0.00005 |
| $B$ | byte, 8 bits |

## Variables

| | |
|---|---|
| $\mathrm{W}_x$ | Weight distribution along x-axis (load cells) |
| $\mathrm{W}_y$ | Weight distribution along y-axis (load cells) |
| $\theta$ | Pitch angle of SeaGul (°) |
| $\phi$ | Roll angle of SeaGul (°) |
| $m_{lce1}$ | Reading from load cell 1 (g) |
| $m_{lce2}$ | Reading from load cell 2 (g) |
| $m_{lce3}$ | Reading from load cell 3 (g) |
| $L_x$ | Load cell distance from weight in X-axis (M) |
| $L_y$ | Load cell distance from weight in Y-axis (M) |
| $F_b$ | Buoyant force (N) |

| | |
|---|---|
| $F_g$ | Gravitational force (N) |
| $F_{tot}$ | Total vertical force (N) |
| $V_D$ | Volume of water displaced ($m^3$) |
| $V_{D0}$ | Volume of water displaced when neutral buoyant($m^3$) |
| $V_{D+}$ | Volume of water displaced when positive buoyant($m^3$) |
| $V_{D-}$ | Volume of water displaced when negative buoyant($m^3$) |
| $a_{Zb}$ | Vertical acceleration due to buoyancy ($m/s^2$) |
| $a_{Zp}$ | Vertical acceleration component due to pitch angle ($m/s^2$) |
| $a_{Ztot}$ | Total vertical acceleration in z-axis ($m/s^2$) |
| $a_{Xtot}$ | Horizontal acceleration component ($m/s^2$) |
| $v_Z$ | Vertical velocity (m/s) |
| $v_X$ | Horizontal velocity (m/s) |
| $depth_S$ | Depth during simulation reached by SeaGul (m) |
| $distance_S$ | Horizontal distance travelled by SeaGul (m) |
| $DR_{AN}$ | Data rate of Arduino NANO (B/s) |
| $DR_{RPI}$ | Data rate of Arduino NANO (B/s) |
| $P$ | Payload (B) |
| $T$ | Transfer time (s) |
| $\triangle t$ | Time interval between updates(s) |

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

Autonomous underwater vehicles (AUVs) and, more specifically, autonomous underwater gliders (AUG) collect data in our oceans over an extended period that spans several months for observation and research purposes [Liblik et al., 2016]. An AUG can be operational during this period due to its energy efficient way of operating in a sawtooth pattern using its buoyancy motor, taking advantage of Archimede's principle by changing its total volume to dive and resurface [Busquets-Mataix et al., 2015]. With onboard sensors and subsystems for adjustments of pitch, roll, and yaw, navigation and communication allow AUG to collect data and navigate our oceans autonomously. During 2023, Chalmers Revere, with the Department of Mechanic and Maritime Sciences, took an interest in developing their own AUG called SeaGul. From 2023 to the present, SeaGul is continuously being developed, and more rigorous tests will be performed.

During the testing of SeaGul at Bornö Institute for Ocean and Climate Studies at the end of 2024, the importance of testing SeaGul's control systems, communication, data collection, and navigation without needing a body of water became clear. Due to the amount of time, preparation and planning needed unrelated to the scope of the tests. Therefore, it was decided to investigate the possibility of developing hardware in the loop (HIL) rig as a product development (PD) tool for SeaGul.

HIL rigs are systems developed to simulate products in their intended operational domains as closely to reality as possible by simulating real life obstructions that occur underwater. HIL rigs become a powerful tool for developing, evaluating, and validating complex systems like SeaGul when simulated inputs can be sent freely within the sensor's capabilities. Furthermore, it is a flexible and time efficient way of conducting tests when no body of water is needed. Finally, the HIL rig will provide the opportunity to perform two types of tests. Full scale testing of the complete system and subsystem testing, all within a controlled environment.

Being able to test more frequently, make both small and significant changes, be better prepared for a physical test, and save resources such as money, workforce, and time makes this an interesting opportunity to investigate whether the commitment to developing this system not only saves resources, but also allows for a faster and more flexible development process when easing the testing of SeaGul by not always requiring a body of water.

## 1.1 Purpose

This project aims to design and build a HIL rig that simulates SeaGul's deployment cycle in realistic operational conditions, enabling the development, testing, and validation of SeaGul's capabilities within a controlled environment.

## 1.2 Research questions

- Can SeaGul autonomously complete a simulated dive from realistic depth data?
- Can SeaGul autonomously collect data during a simulated dive?
- Are HIL rigs usable to aid in product development regarding gliders?
- Is SeaGul a feasible platform for data collection from a product development perspective?

## 1.3 Limitations / Demarcations

This work investigates SeaGul and no other AUV platforms. This work focuses on the technical aspects of SeaGul, the usage, and the simulation; it will not include areas such as commercial strategies. Furthermore, because the tests are performed on land, the water salinity and temperature sensors on SeaGul will give incorrect values. They, therefore, will not be used for analysis or further investigation to conclude the results. The sensor needed for SeaGul to operate autonomously is the depth sensor, which enables it to dive and resurface. For corrections and adjustment of orientation, there is an inertial measurement unit (IMU). Therefore, these sensors are the focus when developing the HIL rig.

## 1.4 Related work

Liu et al. [2024] designed and verified an HIL control system for remotely operated work class vehicles (ROVs). Their framework enables realistic environmental modelling and motion control validation in real time. Together with the hardware controller, it runs the control algorithm and software, which solve the control tasks from the control console and distribute control instructions to ROV actuators. Significantly reduce risk and operational costs before deployment.

Xu et al. [2025] introduced a buoyancy driven glider simulator capable of modelling non-linear dynamics, linear quadratic regulator (LQR) control, and recursive guidance in a software only environment. However, this does not interface directly with physical hardware.

The AUV control HIL case where robust controllers were deployed on a low-cost Raspberry Pi platform with virtual system modelling has shown that realistic hardware-based validation can be achieved without expensive facilities [Bao et al., 2018].

HIL approaches in other domains underscore the versatility of the method. Prochazka et al. [2021] wanted to increase the operational safety of unmanned aerial vehicles (UAV) by allowing redundant actuators to take over when actuators fail, thus addressing the control problem. To evaluate this, they wanted to use HIL simulations to verify that the implementation of the flight controller and the capability of algorithms were correct before real flights. Kim et al. [2019] performed repeatable tests for dynamic real-life scenarios in connected automated vehicles (CAV). Where their goal was to assess control and plan algorithms using HIL. Another interesting use of HIL is from Otte et al. [2018], who attempted to use HIL to validate power system control applications by simulating large scale power networks and control solutions while including real life components.

While these studies underscore the value of HIL systems with real time hardware integration, they are primarily focused on general underwater vehicles, work class ROVs or used in other areas. AUG, with their unique propulsion methods, low power operation, and mission endurance requirements, presents distinct integration and testing challenges. The current literature lacks real time, hardware integrated HIL platform tailored to the operational and control characteristics of gliders, and no existing study offers a modular, hardware connected rig designed specifically for both subsystem and full system testing of a buoyancy driven AUG like SeaGul in a PD context. Addressing this gap, the present work integrates physical sensors, response interfaces, and realistic glider dynamics into a single, flexible test bed that enables performance assessment, risk reduction, and iterative design improvements without the cost and complexity of extensive sea trials.

### 1.4.1 Development of SeaGul

Like other AUG, SeaGul uses the same principles by having a buoyancy motor module that pumps oil back and forth to change its volume. The rotational module in the middle can change pitch, roll, and yaw. An electrical module on the back includes microcontrollers and internal sensors, such as an IMU. In the end cap, outside of SeaGul in contact with water, external sensors are located, such as the depth sensor. These three modules allow SeaGul to be autonomous by using sensor data over its real time pitch, roll, yaw, and depth. SeaGul can adjust its orientation and perform dives at specified depths from these inputs [Jonsson et al., 2023]. Later, SeaGul development further increased its capability to collect data by including a salinity sensor and a temperature sensor. Safety functionality increased with the development of a drop weight system, allowing SeaGul to float to the surface by dropping the weight, resulting in a change of volume used in an emergency [Falkhage et al., 2024].

### 1.4.2 HIL rig simulation

The HIL rig consists of hardware and software and looks vastly different depending on the product for which it is designed. The concept of HIL rigs remains the same,

evaluating the performance of products under realistic conditions. Different tests and validations of systems can be conducted by integrating these systems into a product's hardware and software within realistic simulated environments. Due to their flexibility, HIL rigs are used in many areas.

### 1.4.3 Data collection capabilities of AUG's

AUG's collect data by moving slowly in a sawtooth pattern, using its buoyancy motor to dive and resurface and its wings to drive itself forward. AUG's are usually deployed for months, and because of their endurance and energy efficiency, they move slowly. Despite this, when collecting data, low speed is essential for better accuracy and correlation between the data points gathered. Looking at five different gliders, the Slocum Thermal, Deepglider, Seaglider, Spray, and Slocum Battery, typical data samples include temperature, salinity, currents, chlorophyll fluorometer readings, depth, and suspended material such as microscopic animals and other particles. The depth at which these gliders collect data is typically between 4 and 1500 meters, while the deep gliders have a depth range of up to 6000 meters. The distance these gliders can cover is usually between 1500 and 4700 km. Again, the Deepglider achieved 8500 km, while the Slocum Thermal covered 40000 km. The glider's speeds are typically between 0.25 and 0.4 meters/second [Wood and Mierzwa, 2013]. Depending on the mission and the type of data collection, these gliders can be used to meet the needs because of the wide range of depths, distances, and sensors among them.

### 1.4.4 Product Development Tools for Underwater vehicles

The benefit of using the HIL rig as a PD tool is the reduced risk of ocean tests, fewer resources needed, and more concurrent testing can be performed. The challenge is to imitate the reality and development of the system itself, which can prove to be difficult. One significant detail to highlight is that realistic outcomes depend on the product in question not being able to detect when the HIL rig is connected. The product needs to behave as in real life without affecting its behaviour physically or through software.

Alternative testing methodologies for underwater vehicle applications are also being developed. From National Institute of Standards and Technology [2017], rigorous testing for remotely operated maritime systems is demonstrated using the *Test Method Suite* presented. This suite includes multiple objects, bodies of water, and different approaches to test control schemes, such as inspecting floating objects above and below water, grasping floating objects, and hooking floating objects. From [Higuera et al., 2022], they present an autonomous unmanned underwater vehicle that is used as a platform to evaluate different control, communication, or vision algorithms as a support tool for research tasks in underwater exploration. Another powerful tool used is MATLAB Simulink to develop and simulate control schemes. In [Herman, 2021], they wanted to design the control to achieve the desired water

trajectories. They concluded that they could examine the system's dynamics and compare different models in the simulation stage. Thus, it is vital to be able to decide whether to proceed with the experiments or not.

#### 1.4.4.1 HIL as a product development tool

As Zhang et al. [2022] shows, an HIL rig can integrate 3D environments and be further developed with time. It is a PD tool that develops side by side with the product. Shao et al. [2014] concluded that not only are HIL rig systems needed for UUV's research and development process, but they could also satisfy both deep- and shallow water simulations, and the results are close to tests in water, which further strengthens the use of HIL rig systems.

## 1.5 Outline of thesis

Chapter 2 provides an overview of the different parts of building an HIL rig to guide readers and provide a better understanding of the process. Chapter 3 presents the HIL rig developed for SeaGul. Chapter 4 discusses SeaGul's HIL rig capabilities and its use as a product development tool. Lastly, Chapter 5 presents the conclusion.

# 2
# Methods

HIL rig systems like the one developed for SeaGul consist of multiple components and integrated subsystems that work together to form a complete functional system. To guarantee the reliability and integrity of the resulting data, robust and deliberate methods are implemented. This chapter will outline the methods, starting with a system overview before detailing each specific section.

## 2.1 Overview of setup

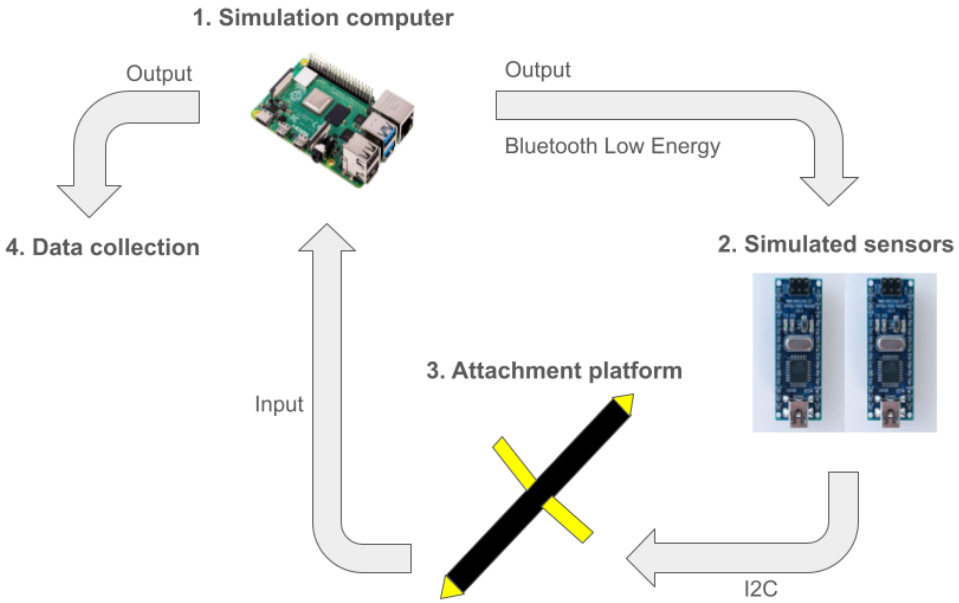



**Figure 2.1:** High level of the systems inputs and outputs

Figure 2.1 presents the complete system. Showing on a high level the different outputs and inputs inside the HIL rig used for simulations.

1. **Simulation computer (Raspberry Pi)**
   Raspberry Pi is the main computer in charge of simulating dives at different depths and updating the simulated model depending on the output from the attachment platform. Furthermore, the Raspberry Pi is responsible for

distributing the correct data from the updated simulation via Bluetooth Low Energy (BLE) to the Arduino NANO boards.

2. **Simulated sensors (Arduino NANO BLE 33)**
   Each Arduino NANO simulates one sensor. In this case, two Arduino NANO are used, simulating the Bar100 sensor (depth sensor) and the ICM 20948 (orientation sensor). These two simulated sensors are connected via $I^2C$ to the glider and replace the two real sensors.
3. **Attachment platform**
   The glider is prepared for the simulation in the same way as an underwater dive. It is sealed, placed under vacuum pressure, and adjusted for a neutrally buoyant state. SeaGul was then mounted onto an attachment platform with four external load cells to determine its orientation. These sensors provide real time input to the simulation, allowing the virtual model to update accordingly, completing the HIL cycle.
4. **Data collection**
   All data from the simulation are collected and saved in an external file after each simulated dive.

## 2.2 Raspberry Pi

In the HIL rig, Raspberry Pi is responsible for processing data from the attachment platform, updating the simulation, and generating new depth and orientation values, which are then sent to the corresponding Arduino NANO. Using a Raspberry Pi for this task, rather than relying entirely on a desktop computer, simplifies the setup. Once the Raspberry Pi is powered on, the system is ready to be used without requiring the user to run multiple programs or manage various software dependencies.

The Raspberry Pi is suited for this role for several reasons. As concluded in Patil [2016], it offers an excellent balance between cost and performance. Its versatility is further enhanced by a large, active community and access to a wide range of libraries supported by the Linux-based operating system. This allows for efficient performance at a low cost. Additionally, Raspberry Pi's ability to interface with various peripherals provides flexibility for future expansion or customisation of the HIL rig.

## 2.3 Arduino Nano BLE 33

The purpose of choosing these microcontrollers is due to their built-in Bluetooth low energy (BLE) capabilities, allowing them to communicate with the Raspberry Pi and simultaneously have the $I^2C$ communication protocol built in. This makes building easier since the actual sensors use $I^2C$ as their communication protocol. The NANO was favoured because of its small size (45 mm x 18 mm) and affordability as an inexpensive microcontroller. Making it possible and affordable to add more simulated sensors in the future.

### 2.3.1 Front end

The front end refers to what SeaGul is allowed to perceive related to requesting data. SeaGul requests data from the simulated sensors, and the simulated sensors output the data they received from the Raspberry Pi.

The Bar100 depth sensor is important to simulate hardware responses from SeaGul and to initiate dives. From its input changes SeaGul's states from idle to diving and resurfacing, as shown in 2.2. SeaGul is either neutral buoyant (idle) or waits for the command to start. The valve is opened, allowing oil to return to the reservoir, making it negatively buoyant to start diving (diving). When Bar100 reads the specified depth, SeaGul pumps oil out to the bladder, making it positively buoyant (resurfacing).

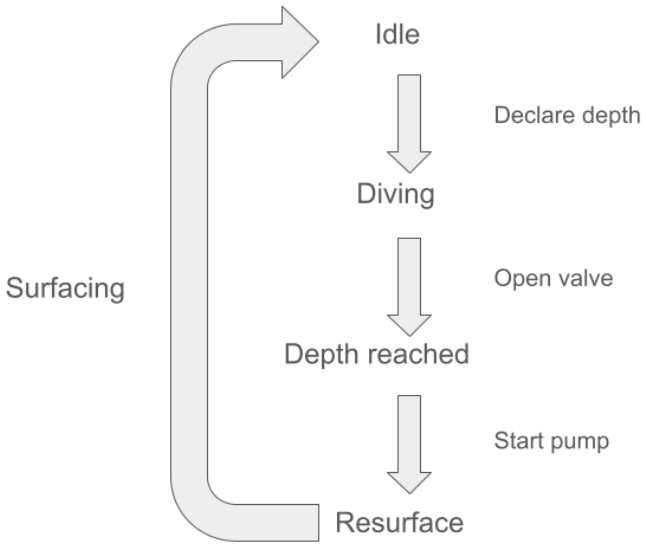



**Figure 2.2:** SeaGul's states during a dive

SeaGul corrects orientation during dives by adjusting the battery position. By taking orientation data from the orientation sensor IMU-20948's gyroscope, SeaGul can correct itself, making this an important sensor for simulating and getting hardware responses.

The two simulated sensors connect to SeaGul, as shown in Figure 2.3. Here, the SeaGul control circuit is shown, with the depth sensor connected via a 4-wire pin connector and the IMU connected through four wires to the $I^2C$ pin out.

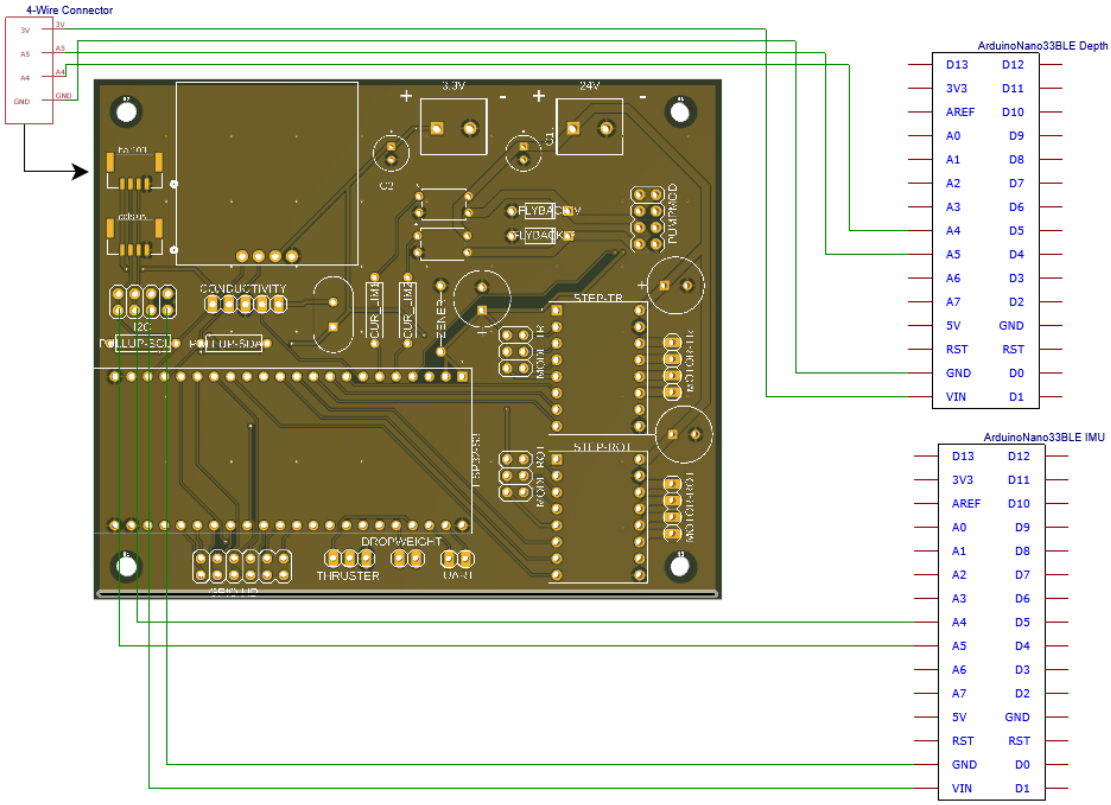



**Figure 2.3:** Simulated sensor circuit

Both sensors will follow the logic shown below in 2.1 to simulate them. Reverse engineering the sensor libraries [BlueRobotics, 2023] and [SparkFun Electronics, 2025] to determine the data to send.

**Listing 2.1:** Simulated sensor code logic

```
#define REGISTER_ADDRESSES // Addresses to registers the sensor uses

void setup() {
//Initialise sensor as a peripheral
Wire.begin(SENSOR_ADDRESS);
// requestEvent is called when the controller asks for data
Wire.onRequest(requestEvent);
}
void requestEvent() {
 requestID=Wire.read(); // Which register the controller want data from
 if(requestID==ADDRESSES){ //Checks data to send depending on register
    uint8_t data[];
    data[0]=BYTE0;        //Send relevant data
    data[1]=BYTE1;        //depending on which
    data[2]=BYTE2;        //register is accessed
    Wire.write(data,3);
}
 else {
    Wire.write(0x00);   //Send some arbitrary number
}
```

SeaGul interacts with the web interface to receive information. Figure 2.4 shows the settings for the dive depth and the potentiometer value that determines how much oil is pumped back and forth. Figure 2.5 shows the positions of the stepper motors' and allows manual movement forward, backward, left, and right. In Figure 2.6 the user can initiate dives and place SeaGul in different states.

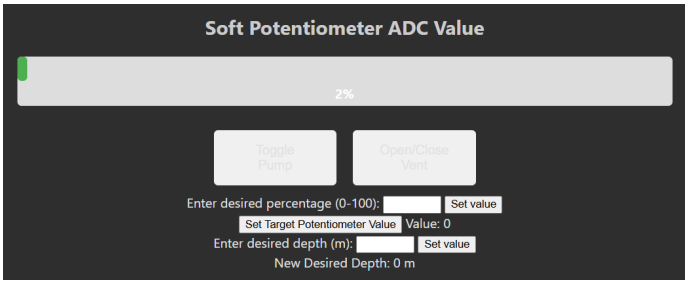



**Figure 2.4:** Webpage set depth, potentiometer

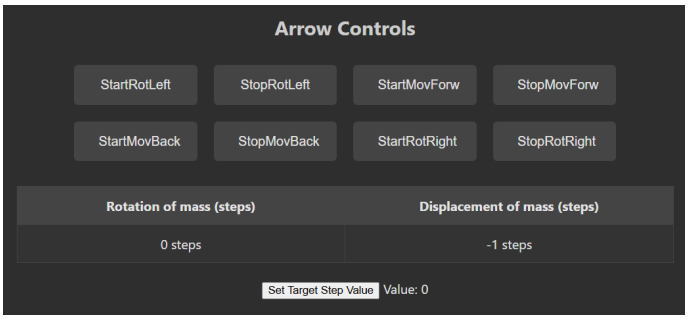



**Figure 2.5:** Webpage arrow control


**Figure 2.6:** Webpage buttons

When the user initiates the dive, SeaGul takes the information from the target position of the stepper motors, the potentiometer target value, and the desired depth. Thereafter, SeaGul will go into the diving state and begin to open its valve to reach the desired oil volume. During the dive, SeaGul read values from the depth sensor and IMU. If IMU reports a high rotation or pitch value, SeaGul adjusts the adjustment. When the desired depth is reached, SeaGul goes into a diving up state and shifts the motors backward to the target for the stepper motors and starts pumping oil to fill the bladder. When the surface is reached, SeaGul goes into the surface state and shifts the motors all the way forward. For this to work, the entire frontend loop needs to work. In Figure 2.7 the frontend loop is shown. First, the webpage is populated. During the dive, the simulated depth and IMU sensors will send SeaGul readings from the simulation, and SeaGul response according to the updated values.

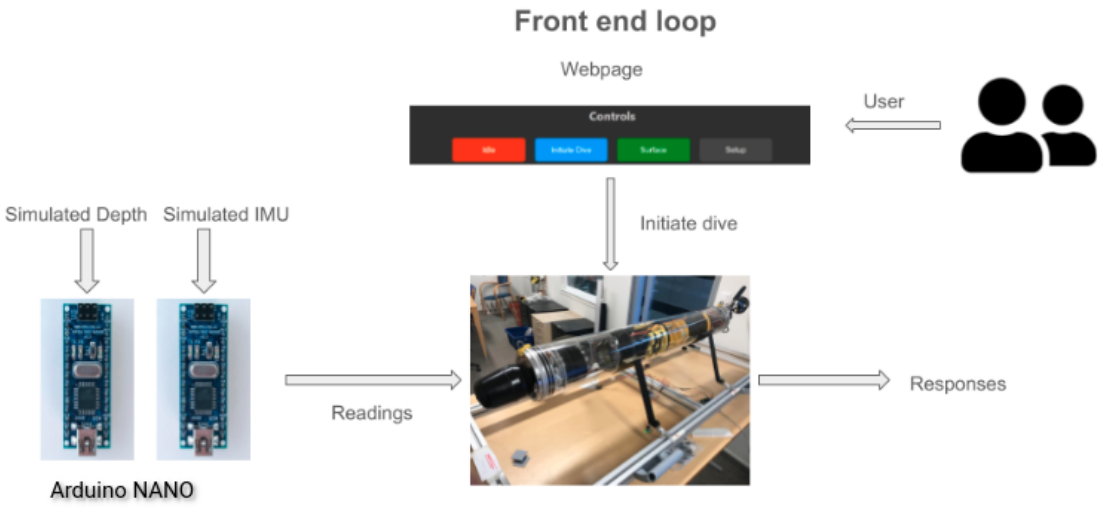



**Figure 2.7:** Front end loop

### 2.3.2 Backend

The back end refers to what is not visible to SeaGul related to receiving data. The aim is for SeaGul to not know whether it is connected to the HIL rig or is in the water. This is important, and having an intermediary device, in this case the Arduino NANO, makes it possible. To achieve this, the data are sent via BLE to the Arduino NANO to avoid interference with the $I^2C$ channels. The Raspberry Pi is equipped with Bluetooth 4.2, which supports BLE, making this a desirable choice when considering the amount of data sent. For this to work, the BLE must be able to send the amount of data requested from the sensors.

The models of Raspberry Pi 3 B+ and newer use BLE 4.2, which supports up to 251 bytes of payload. The rate at which it can transfer data is 7,5ms.

$$DR_{RPI} = \frac{P}{T} = \frac{251}{0.0075} \approx 33.5 \ KB/s \tag{2.1}$$

From our two combined sensors, the maximum total is 25 bytes (20 IMU, 5 Bar100). Here is not the speed at which the sensors gather information. A critical parameter is the frequency with which the glider requests data. In our case, the program running on the glider asks for data around every 105ms (3 data points and a 35ms delay between).

$$DR_{AN} = \frac{P}{T} = \frac{25}{0.105} \approx 238 \ B/s \tag{2.2}$$

The loading of the simulated sensors with data before SeaGul asks for it to avoid data underflow is important. This data collection interval is remarkably high compared to typical glider behaviour, 4-20 times a day [NOAA Atlantic Oceanographic and Meteorological Laboratory (AOML), 2022], and can be alternated with a much longer delay and still maintain realistic integrity.

From Equation 2.2, SeaGul asks for 238 bytes/second, and with the setup, it is theoretically possible from Equation 2.1 to send 33.5 KB/second, allowing further development and expansion of the rig. Even if the payload and effectiveness of the transfer time are less.

The back-end data transfer chain follows Figure 2.8. Here, load cells read the responses from SeaGul and send them to Arduino UNO, which recalculates them into grammes and sends them to the Raspberry Pi that does the simulation. The readings are calculated for depth, pitch, and roll and sent to a Python program that sends the values to the correct simulated sensor.

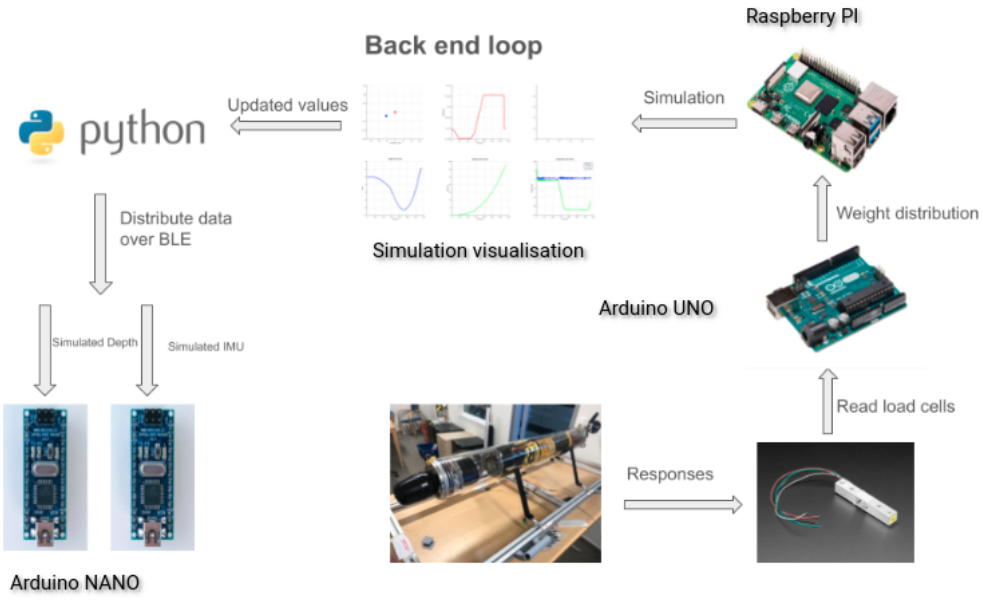



**Figure 2.8:** Back-end chain

### 2.3.3 Attachment platform

The attachment platform needs to know what SeaGul's hardware responses translate for its orientation now that SeaGul will not move.

#### 2.3.3.1 How it works

When SeaGul is placed on this platform, all load cells read 0 (SeaGul is neutral buoyant). When SeaGul translates or rotates its batteries, it will change its centre of gravity, which the load cells pick up. From their readings, an orientation is calculated by how much weight is distributed between them, and these responses from the load cells act like a feedback loop to the simulation to update orientation.

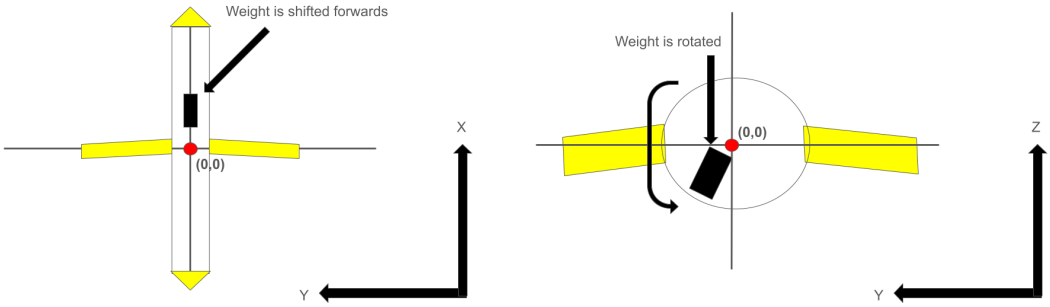



**Figure 2.9:** How the weight is shifted

Figure 2.10 shows the attachment platform. Four load cells are mounted on the metal frame. A bracket on each load cell is fastened and then secured to one of the two holds, to which SeaGul is lowered and secured.

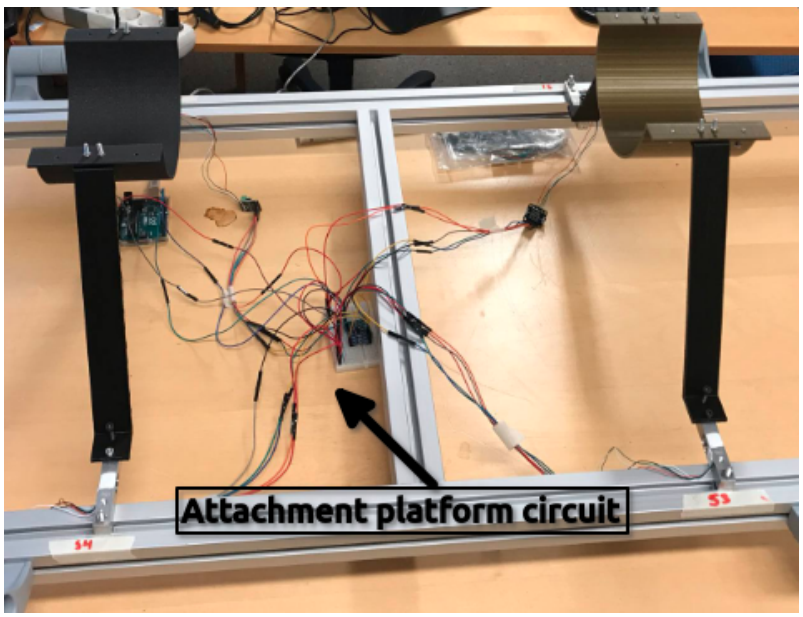



**Figure 2.10:** Attachment platform

For each load cell, an Adafruit NAU7802 breakout board is connected. These cards convert the analogue input from the load cell to the digital input with gain to boost the signal able to be measured. The cards also have inbuilt calibration, together with settings to change readings per second. These extra functions simplify the process. The load cells are connected to each of these four cards, which is connected to a TCA9548 breakout board to use multiplexing to an Arduino UNO. A multiplexing board is used for SDA and SCL inputs due to the NAU7802 having the same addresses, avoiding interference with each other. The multiplexer card oversees this and sends the information to the Arduino UNO. See Figure 2.11 for the electrical circuit.

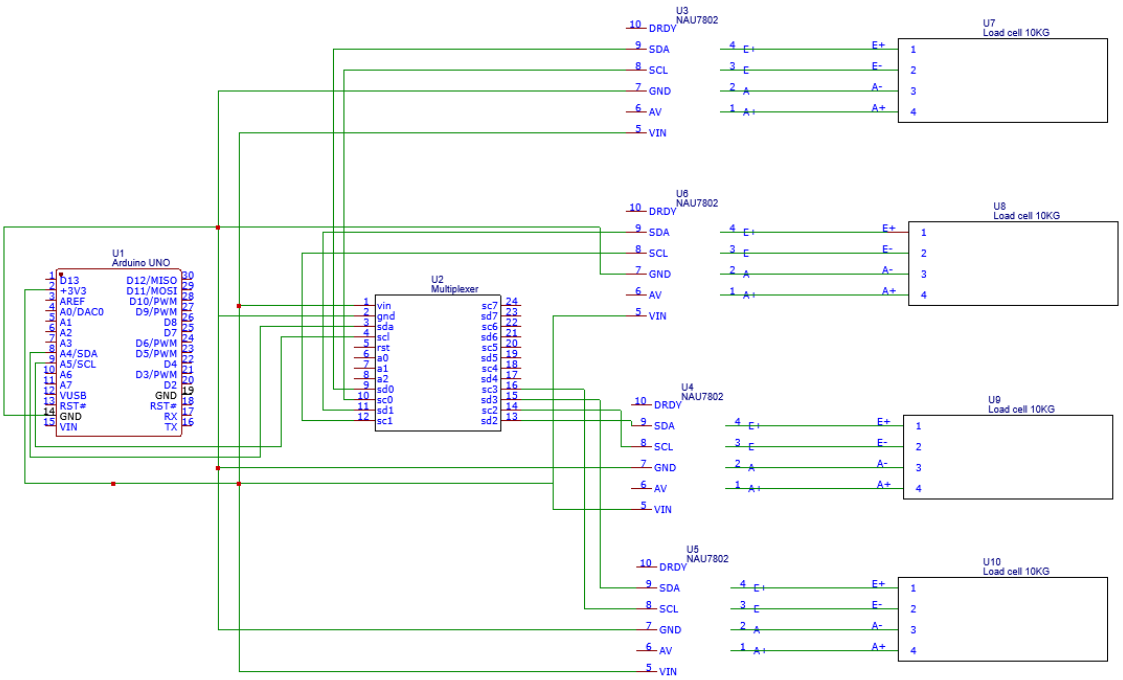



**Figure 2.11:** Attachment platform circuit

The load cells hold up SeaGul and need to manage a weight of around 20 kg simultaneously, giving accurate readings for an extended period. The load cells used can manage 10 kg each, and accuracy was evaluated with known weights of 100g, 750g, and 2095g that were placed on top of them for 10 minutes to monitor if the readings change over time.

## 2.4 Computer simulation

The simulation is done in Octave and starts by giving SeaGul a target depth through the website interface, which puts it into the diving state. From this point forward, the output from the attachment rig gives the simulation speeds at which SeaGul dives and its angle. From these readings, depth, distance, and orientation are updated during the dive and sent to the simulated sensors. During the simulation, all values are updated in real time, giving real time visualisation and responses from SeaGul. The flow of data is presented in Figure 2.12.

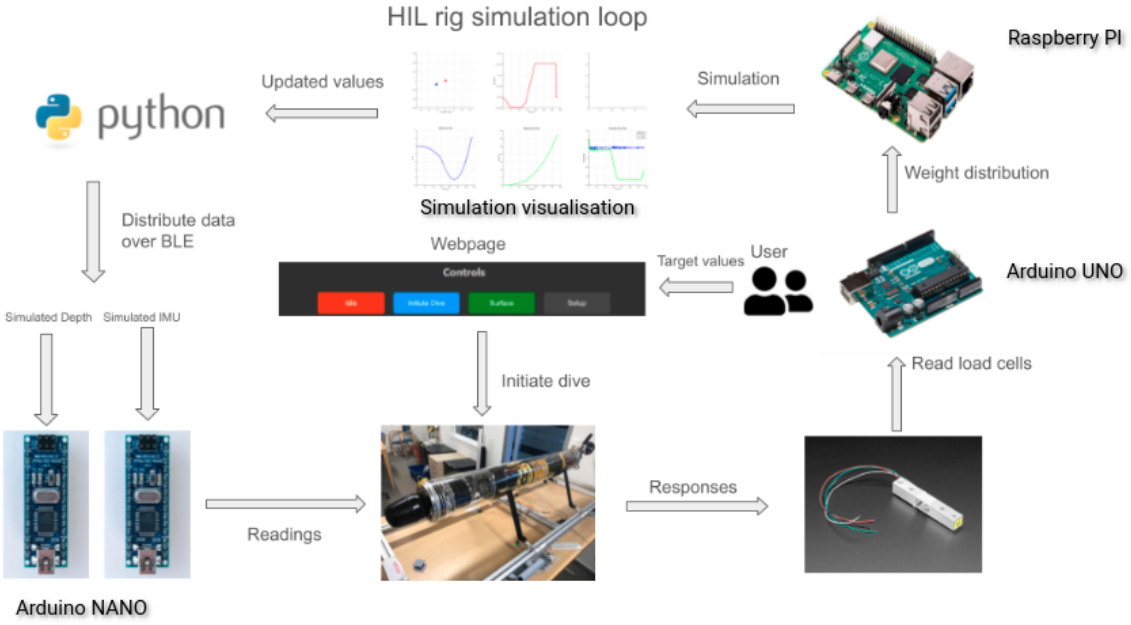



**Figure 2.12:** HIL rig simulation loop

To be able to follow the simulated dive more easily, a simulation visualisation interface was developed to show the user key data points in real time for a clearer view. In Figure 2.13 the whole interface is shown, which includes the centre of gravity, the displacement of the water (buoyancy), the depth, the distance, and the orientation with pitch and roll.

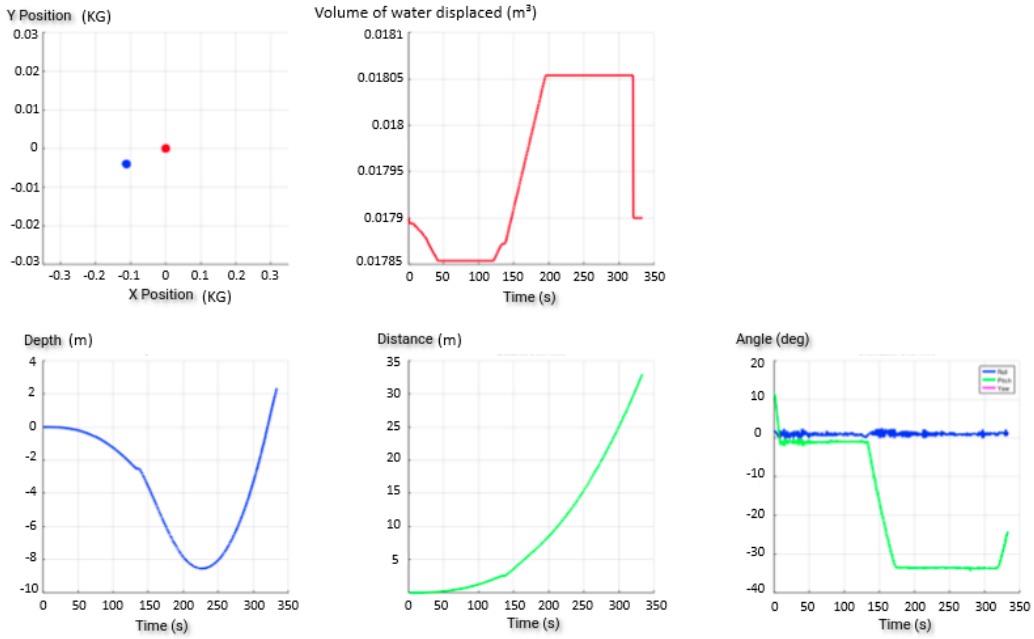



**Figure 2.13:** Simulation visualisation interface

In Figure 2.14 the real time centre of gravity is shown as a blue dot, and how it moves relative to its original buoyant state during the duration of the simulation, illustrated as the red dot in the middle between the four load cells in 2D. This helps to visualise how the weight is distributed in real time.

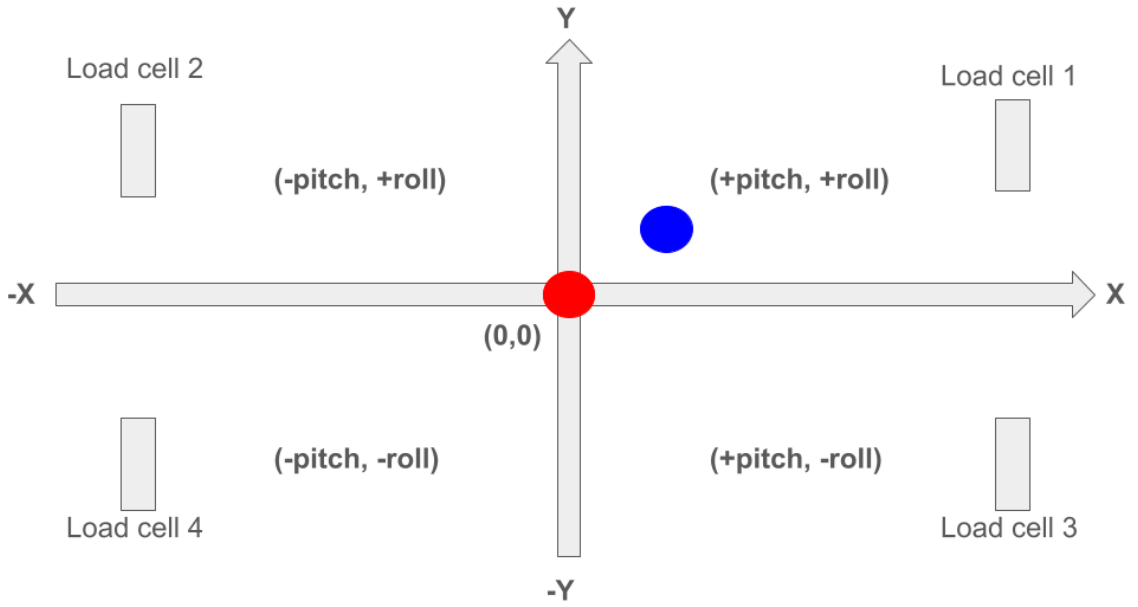



**Figure 2.14:** Visualisation of real time CoG

Four sectors are displayed, indicating how the weight is distributed and the corresponding pitch and roll values it produces. The position within the sector is also considered.

The equation 2.3 for the weight on the x-axis.

$$\mathrm{W}_x = (\frac{1}{1000}) \cdot (m_{lce1} + m_{lc3}) \tag{2.3}$$

Because the load cell gives the weight in grammes, it is first recalculated as KG. Furthermore, not all load cells are included in the equation. This is because the equilibrium in the beginning, they are all calibrated to 0, which leads to two becoming positive (+) and two negatives (-) with equal amounts, and therefore cancel each other out when the weight shifts. So, to negate that, the focus is on one side only.

Equation 2.4 is like 2.3 but for the y-axis.

$$\mathrm{W}_y = (\frac{1}{1000}) \cdot (m_{lc1} + m_{lc2}) \tag{2.4}$$

Again, because the four load cells cancel each other out, one side is used for the weight distribution in the y-axis.

From the position and amount of weight, the orientation calculations are done for both pitch and roll. Figure 2.15 illustrates how the pitch can be found, and in equation 2.5, it shows how the pitch is calculated.

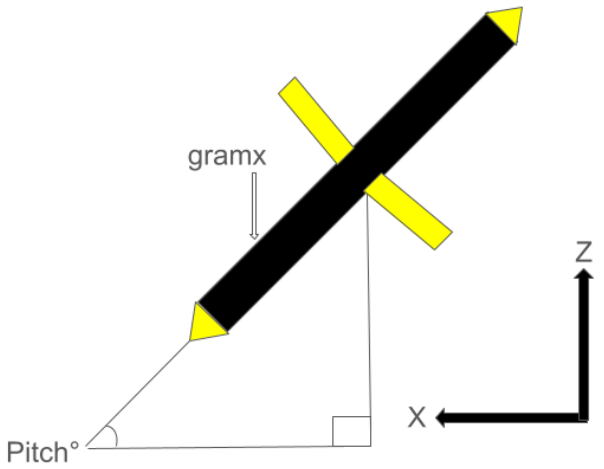



**Figure 2.15:** How translation of weight results in pitch

$$\theta = tan^{-1}\left(\frac{2 \cdot \mathrm{W}_x}{L_x}\right) \tag{2.5}$$

Here $L_x$ is how far from the load cells are from the weight, and because we start in the middle, the length is divided by two.

Figure 2.16 illustrates how the roll can be found, and in equation 2.6 it is how the pitch is calculated.

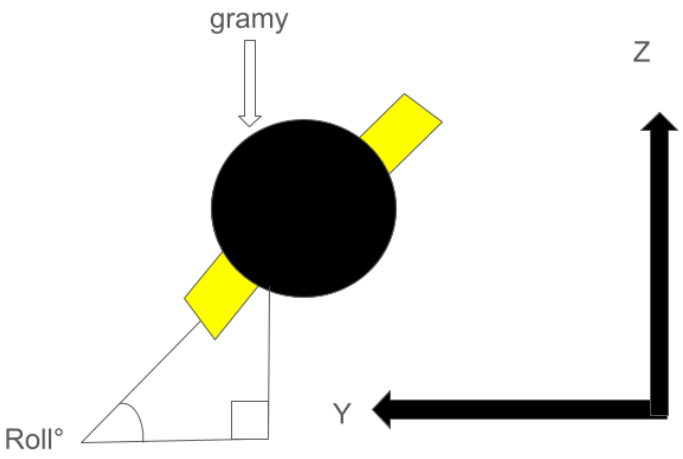



**Figure 2.16:** How rotation of weight results in roll

$$\phi = tan^{-1}\left(\frac{2 \cdot \mathrm{W}_y}{L_y}\right) \tag{2.6}$$

Here $L_y$ is how far from the load cells are from the weight, and because we start in the middle, the length is divided by two.

The buoyancy force is shown by simulating the volume of water displaced, making it positive and negative buoyant over time. This is done purely in the simulation and does not take readings from any external sensors. Instead, it is grounded in the fact that the buoyant force of AUG's is usually 1N Rudnick et al. [2004]. The buoyancy during the simulation starts as neutral buoyant, 0 Newtons. Then proceeds to change to negative buoyancy over time until -1 Newton. When SeaGul reaches its target depth, the buoyancy starts to change to a positive buoyancy of +1 Newton. The change in buoyancy over time is shown in Figure 2.17.

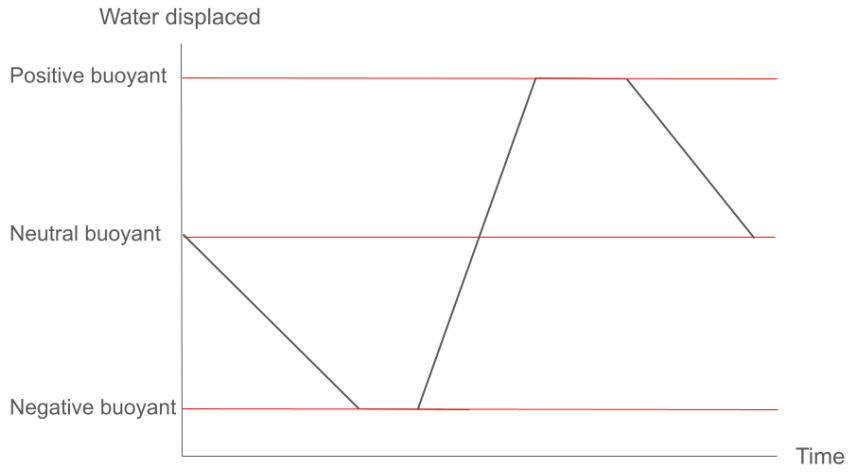



**Figure 2.17:** Buoyancy changes over time

Equations 2.7, 2.8, and 2.9 show how the three different amounts of volume displaced translate to 0, -1, and 1 Newton of force.

$$F_b = \rho_w \cdot V_D \cdot g \tag{2.7}$$

$$F_g = m_{SG} \cdot g \tag{2.8}$$

$$F_{tot} = F_b - F_g \tag{2.9}$$

With the insertion of

$$F_b = 0$$

$$F_b = -1$$

$$F_b = 1$$

The volume of water displaced for these values can be calculated.

The following values for the volume are

$$V_{D0} = 17953.86cm^3$$

$$V_{D+} = 18056cm^3$$

$$V_{D-} = 17851.72cm^3$$

The differences between neutral and negative are

$$V_{D0} - V_{D-} = 102.14cm^3$$

The differences between positive and neutral are

$$V_{D+} - V_{D0} = 102.139cm^3 \approx 102.14cm^3$$

This displacement of volume is done by the oil, so to displace 102.14 mL of water, the same amount of oil is pumped or let back to achieve 1 Newton of positive or negative buoyancy.

Furthermore, because buoyancy does not jump directly from neutral to positive or negative, but instead does it over some time, there is an equation that simulates the increases or decreases in water displacement with the amount of oil pumped or let back, as seen in Figure 2.17. Where first it starts at 17953.86 $cm^3$, then over time it reaches 17851.72 $cm^3$, increases when depth is reached to 18056 $cm^3$, and returns to 17953.86 $cm^3$ when the surface is reached. The incremental increases and decreases are made by the following equation 2.10. Where $step_v$, can be changed depending on which speed is preferred, which also translates to the change in buoyancy force.

$$V_{D,n} = V_{D,n-1} + step_v \tag{2.10}$$

At this point, a basic system to calculate pitch and roll from SeaGul's responses and simulate the change in buoyancy force over time is established. With this, a basic simulation can be implemented with the principles shown in Figure 2.18.

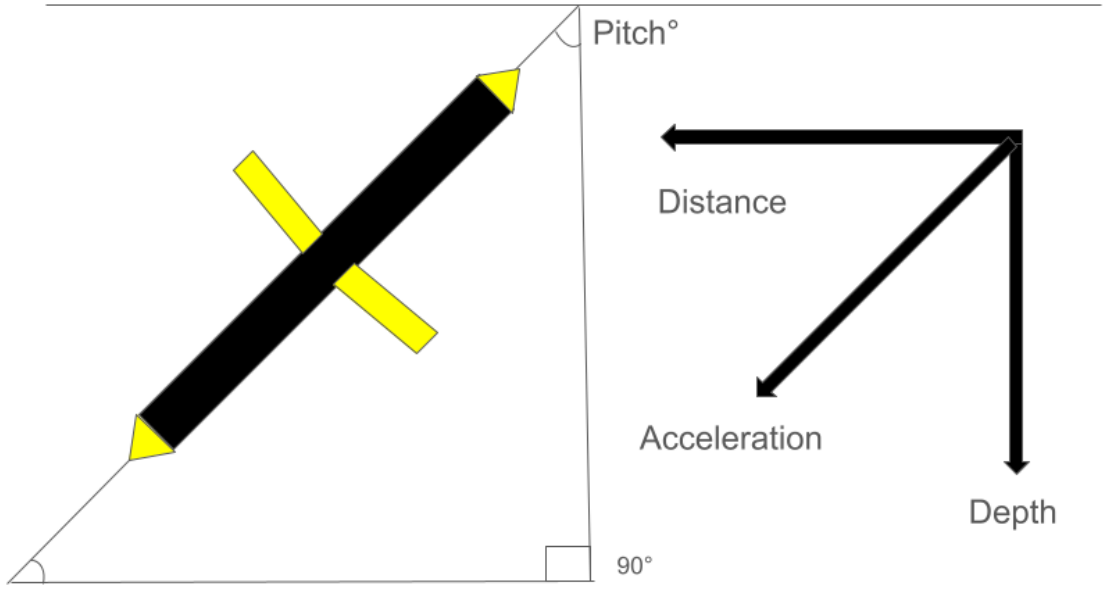



**Figure 2.18:** Basic system to calculate pitch and roll from SeaGul's responses during simulation

First, an acceleration is calculated. From this acceleration can the speed and distance be obtained to obtain both depth and length travelled. To get the distance travelled and depth, it is needed to divide the SeaGul's acceleration into the z-axis acceleration for the depth and the x-axis acceleration to get the distance. First, the amount of acceleration on the z-axis is calculated with equations 2.11 and 2.12

$$a_{Zb} = \frac{F_{tot}}{m_{SG}} \tag{2.11}$$

This is the acceleration achieved by the displacement of water in the z-axis. The simulation also needs to consider how the acceleration changes depending on the pitch. To simulate this, the amount of acceleration gathered from the pitch that is translated into acceleration on the z-axis is calculated by equation 2.12, and then the total z acceleration is calculated as shown in 2.13.

$$a_{Zp} = a_{Zb} \cdot sin\,(\theta) \tag{2.12}$$

$$a_{Ztot} = a_{Zp} + a_{Zb} \tag{2.13}$$

Because SeaGul can still descend if the pitch is 0, thanks to the buoyancy force, but the acceleration increases depending on the angle of attack. This is why the two accelerations are added together. The more pitch increases, the more $a_{Zp}$ increase and vice versa if pitch decreases.

The acceleration in the x-axis is calculated with equation 2.14.

$$a_{Xtot} = |a_{Zb} \cdot cos\,(\theta)| \tag{2.14}$$

The difference from z-axis acceleration is that the acceleration in the x-axis depends only on how much of $a_{Zb}$ is translated into x-axis acceleration, because there is no buoyancy force in the x direction. An important fact to consider is also that the acceleration is always forward, even if the pitch is negative, because SeaGul travels forward during its sawtooth pattern of up and down. Therefore, the absolute value is used.

With the two accelerations, $a_{Ztot}$ and $a_{Xtot}$, velocities can now be calculated in the two directions. Equations 2.15 and 2.16 show how $\triangle v_{Z,n}$ and $\triangle v_{X,n}$ are estimated.

$$\triangle v_{Z,n} = a_{Ztot} \cdot \triangle t \tag{2.15}$$

$$\triangle v_{X,n} = a_{Xtot} \cdot \triangle t \tag{2.16}$$

$\triangle t$ in the equations is the time between the updates from the attachment rig. If there is a delay in the simulation and $\triangle t$ becomes much larger, it defaults to a standard amount of 70ms to prevent large increases or decreases that ruin the simulation.

SeaGul's velocity increases during constant acceleration, decreases if acceleration decreases, and increases if acceleration increases. To simulate this, it is therefore needed to add or subtract the new velocities together to get the updated velocity on the z and x axes. This is done following equations 2.17 and 2.18.

$$v_{Z,n} = v_{Z,n-1} + \triangle v_{Z,n} \tag{2.17}$$

$$v_{X,n} = v_{X,n-1} + \triangle v_{X,n} \tag{2.18}$$

From this velocity can both distance and depth be calculated. In equation 2.19 the depth is calculated and in 2.20 the distance travelled is calculated.

$$depth_{S,n} = depth_{S,n-1} + (v_{Z,n} \cdot \triangle t) \tag{2.19}$$

$$distance_{S,n} = distance_{S,n-1} + (v_{X,n} \cdot \triangle t) \tag{2.20}$$

Yaw is not included in the simulations, although it is often grouped with pitch and roll along the x, y, and z axes. There is no way to determine the yaw from pitch and roll alone. Assume that the glider's pitch and roll are fixed. SeaGul can achieve this fixed position in infinite ways. This is because SeaGul can have the same pitch and roll when rotated around the z-axis. To effectively calculate real yaw, an external measurement unit is needed to obtain directional data that can lock the angle at which SeaGul is translated around the z-axis, such as an IMU.

When the user stops the simulation, all the collected data is saved in a file.

### 2.4.1 Data transfer

The final step is to connect these components into one loop so that the front end, back end, attachment rig, and computer simulation operate together. To demonstrate this, follow how the data is packaged, sent, and unpackaged among the different parts. The explanations clarify the design logic and actions taken to meet the component requirements.

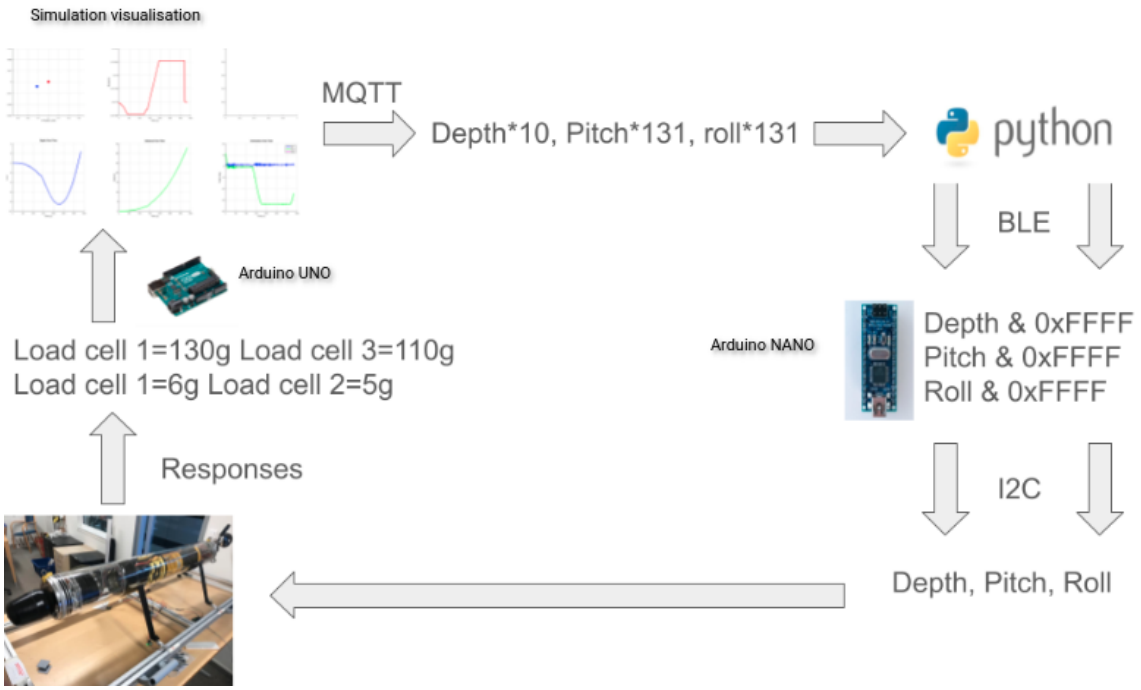



**Figure 2.19:** Back-end chain

Everything starts with the user setting target values in the webpage interface, which triggers the dive. Then, the HIL rig simulation starts, and the attachment platform first provides updated values to the simulation. Before it can be sent to the simulation, the load cell data is first passed through the Arduino UNO. This is needed to set the gain and how many readings per second for the load cells. Right now, the gain is set to 64 and to do 10 reads per second. Even if it is about 100ms between the readings, Arduino UNO resends the latest numbers every 50ms in between to always populate the simulation at a steady pace. Figure 2.20 illustrates how the multiplexer works, as to why the Arduino UNO is needed. Imagine how the computer can know what number was sent from the NAU7802 breakout board if they have the same address. Therefore, a multiplexer is needed that can decipher and send the correct value to a different address so that the computer can read each value correctly.

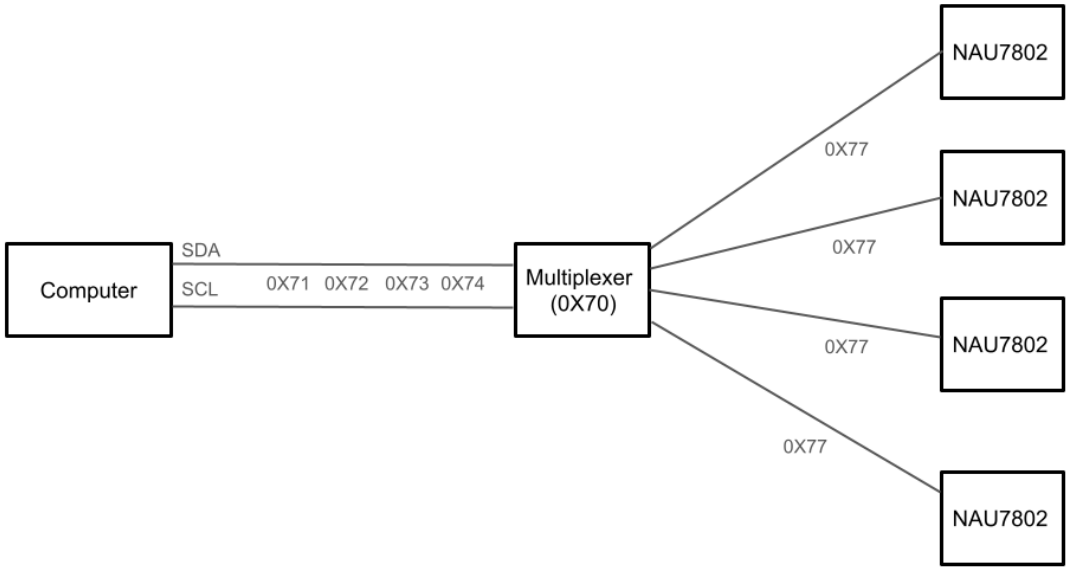



**Figure 2.20:** I²C Adress Illustration

In Octave, where the simulation is performed, the transmitted data is used to compute depth, pitch, and roll. The depth value is multiplied by 10 to obtain one decimal place, while pitch and roll are multiplied by 131, corresponding to the internal gain of the actual IMU.

Then it is forwarded in the correct format to a Python script. In that Python script are the updated values, then packaged and sent with BLE in the correct format to the correct Arduino NANO. In Python, they are made into 16-bit unsigned integers to be able to handle negative depth, pitch, and roll.

When the Arduino NANO simulating the depth sensor receives the 16-bit integer, is it first divided by 10 to get one decimal, then is the depth recalculated following the equation 2.21 derived from the sensor manual [KELLER AG für Druckmesstechnik, 2017] for internal calculation.

$$Output = \left( \frac{\left( \frac{depth_S \cdot \rho_w \cdot g + P_{atm}}{P_atm} - P_{min} - P_{mode} \right) \cdot 32768}{P_{max} - P_{min}} \right) + 16384 \tag{2.21}$$

This recalculates the value into pressure and sends the data to SeaGul. For the Arduino NANO simulating the IMU, it unpacks the data by first dividing the pitch and roll by 131 to even out the gain and gets two decimals. The data received are checked for whether it is a negative or positive number. This is done with the following method $pitch >= 0x8000, roll >= 0x8000$, because $0x8000$ is 32768, which is exactly halfway, and by subtracting 65535, the negative number is found.

The simulated sensors then package the recalculated value to fit the real sensors library and then send it to SeaGul, which reads the new values, gets the depth, pitch and roll, and makes a response from the information gathered.

## 2.5 How to evaluate HIL rig performance

Building a system is one hurdle, and evaluating it is critical to understand how good a system it is. Therefore, different tests will be performed on the hardware, software and data of the simulation, which will later be examined and compared to the data collected at Bornö during a real dive.

### 2.5.1 Evaluation of hardware

See that all core elements and subsystems work as intended.

- **Test 1. Back end**
  Verify that Octave transmits the correct data to the Raspberry Pi, which in turn sends the appropriate data to each respective simulated sensor.
- **Test 2. Front end**
  Check that the Arduino NANO is sending the correct data to SeaGul and that the webpage is functioning properly.
- **Test 3. Usage of load cell data**
  The simulation uses the data from the load cell to update and produce new depth and orientation outputs.
- **Test 4. Real time visualisation**
  Is the real time visualisation visible and updated during simulation.
- **Test 5. Data graphs**
  Graphs for the data, including orientation, depth, time, distance, and buoyancy, are produced at the end of the simulation.
- **Test 6. SeaGul**
  SeaGul is secured and still while rotating and translating its motors to avoid giving wrong readings.
- **Test 7: Current drawn**
  When switching from real sensors to simulated sensors, it is important to examine changes in current and voltage. To maintain the integrity of SeaGul, the power consumption of the simulated sensors should closely match that of the real ones. This ensures accurate estimation of battery life, evaluation of battery efficiency, and reliable testing of new energy saving dive cycles. This was carried out using a power supply unit capable of delivering a constant 24V while measuring the current drawn by all SeaGul components.
- **Test 8: Load cell accuracy**
  The load cells must remain accurate throughout the simulated dive, as the entire simulation relies on these values. Therefore, their precision is tested using known weights, assessing how closely the load cell readings match the expected values and whether they remain stable over a defined period of 10 minutes.

### 2.5.2 Evaluation of the HIL rigs output

Different tests are done to observe that all visualisation tools work as intended and which values for pitch and roll the simulation can produce.

- **Test 9: CoG visualisation**
  Verify that the centre of gravity visualisation updates correctly depending on the position of the motors.
- **Test 10: HIL rig output range**
  Shift the batteries to their maximum position and observe the maximum positive and negative pitch angle for the system and the maximum negative roll angle for the system.

These tests allow us to, in a controlled manner, understand possibilities for simulation, that the system behaves correctly, and what SeaGul's different responses translate to for the rotation and translation of batteries. Evaluate if they are reasonable and see if SeaGul's behaviour is intact.

### 2.5.3 Evaluation of a fully simulated dive

Simulate a dive and resurface cycle, running the same code as used at Bornö. The responses and data from the fully simulated dive can be examined against Bornö's pitch, roll, depth, and time data. The data output from the attachment rig during the simulated dives is compared to the real-life dive. The following sections will be examined in detail to determine the quality of the data produced within the HIL system and how it holds up against the real world.

Although real dive data do not include direct measurements of parameters such as distance travelled, time, or speed, these values can be estimated. Using time, it is known that SeaGul collects one sample every 105ms, which corresponds to approximately 9.5 samples per second. Therefore, the duration of events can be estimated by dividing the number of samples by 9.5. Since the system also logs the active state, it is possible to determine how long SeaGul remained in specific phases, namely *gliding down* (state 4) and *gliding up* (state 5). This enables calculating the time taken to reach the target depth, the duration of the turnaround, and the time required to return to the surface.

Estimating distance and speed is more complex. However, by using the available pitch and depth data, along with the estimated time derived from the sample rate, it is possible to approximate the general speed during each phase of the dive. This general speed is calculated using the average pitch angles, depth reached, and the time taken to descend an ascent. With these speeds and the corresponding durations for descent, turnaround, and ascent, the distance travelled in each phase can be estimated. The total of these values provides an approximation of the total distance travelled during the dive.

# 3

# Results

The results chapter will present the findings related to the research questions based on the methodology described. Graphs and images will be used to clearly illustrate the outcomes.

## 3.1 Result of HIL rig performance

In this chapter, the results of Tests 1–14 and the fully simulated dive will be presented following the same structure as outlined in the methodology

### 3.1.1 Evaluation of hardware

The following are the results of Tests 1–6, which focus primarily on the subsystems and the interactions of the HIL rig with other components.

#### 3.1.1.1 Test 1. Back end

Because the sensors receive data over BLE, each must have a unique identifier known as a Universally Unique Identifier (UUID). This ensures that communication with a sensor occurs only when the correct UUID is matched. If the data are processed incorrectly or recalculated in a way that does not align with the corresponding library, the output will be invalid, serving as a simple yet effective security measure.

To verify that the simulated sensors received the correct data transmitted from the Raspberry Pi were the two sensors connected to a power source, specifically a computer. Each simulated sensor had a specific programme uploaded: the depth sensor used simulatedDepthWithBLECommunication, and the IMU sensor used simulatedIMUWithBLECommunication. The Python script sendDataToArduino was then executed, successfully connecting both simulated sensors.

Once the simulation began, the Octave programme simulationSoftware correctly forwarded the depth and orientation data via the MQTT service to the Python script, which in turn transmitted the appropriate data to the simulated sensors over BLE, confirming that the communication loop is fully functional and accurate. The codes are included in the appendix A.

#### 3.1.1.2 Test 2. Front end

To observe how SeaGul interprets the values sent from the two simulated sensors, the web interface is used. Provides a live update view of SeaGul's data interpretation, as shown in Figure 3.1.

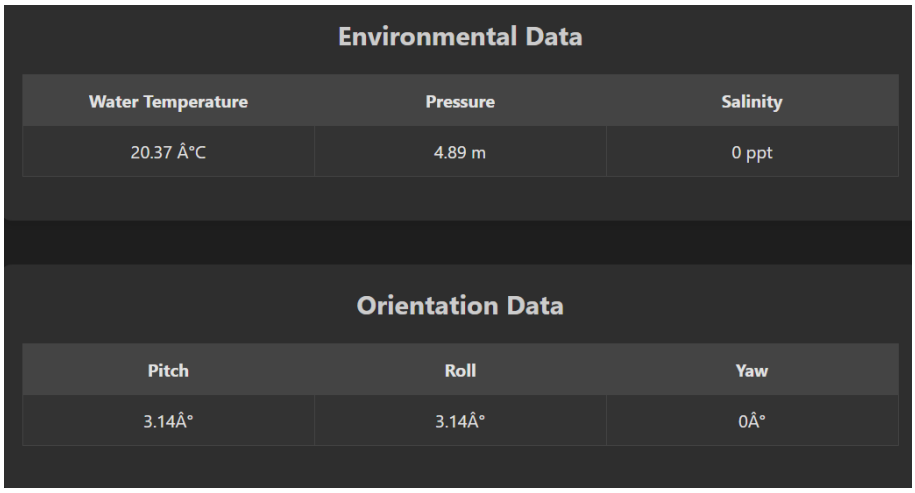



**Figure 3.1:** Live update of how SeaGul interprets simulated data

Additionally, the web interface should allow for manual control of SeaGul. Figure 3.2 shows the steps taken by the stepper motors, indicating that SeaGul can be operated manually.

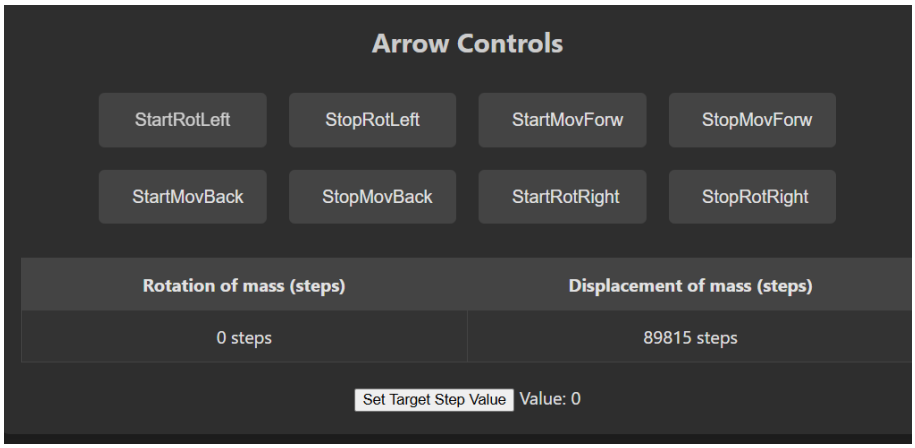



**Figure 3.2:** Interface to manually control SeaGul

#### 3.1.1.3 Test 3, 4 and 5. Usage of load cell data, real time visualisation and data graphs

During simulation, the visualisation interface is visible and continuously updated, as shown in Figure 3.3.

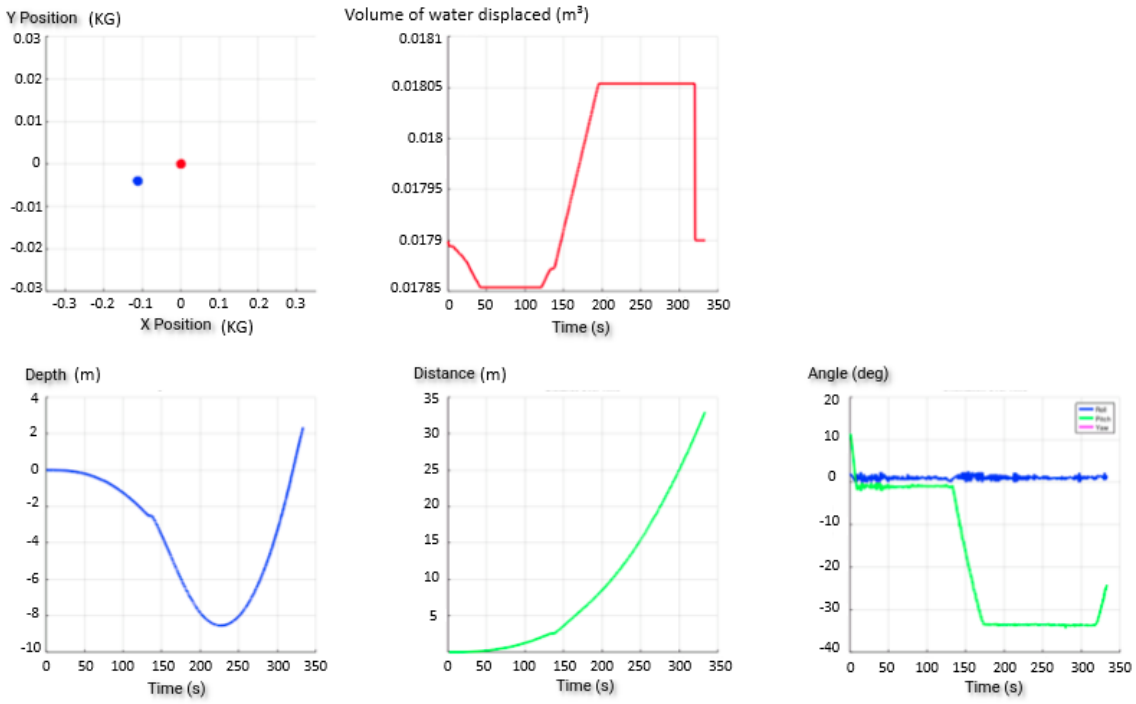



**Figure 3.3:** Simulation interface in real time

At the end of the simulation, data charts are generated as shown in Figure 3.4, including orientation, depth, time, distance, and buoyancy.

| Time (s) | Roll (°) | Pitch (°) | Yaw (°) | Buoyancy (N) | Depth (M) | Distance (M) |
|---|---|---|---|---|---|---|
| 0.00308299 | 1.90915 | 11.3099 | 0 | 0.0178991 | -3.40493e-07 | 2.79137e-07 |
| 0.425636 | 1.90915 | 11.3099 | 0 | 0.0178982 | -0.000178743 | 0.000146534 |
| 0.693951 | 1.90915 | 11.3099 | 0 | 0.0178973 | -0.000367745 | 0.000301479 |
| 0.753124 | 1.90915 | 11.3099 | 0 | 0.0178964 | -0.000650135 | 0.000532983 |
| 0.897772 | 1.90915 | 11.3099 | 0 | 0.0178955 | -0.00102615 | 0.000841238 |

**Figure 3.4:** Example of saved simulated data

By comparing the graphs in the visualisation interface with the data charts, it can be confirmed that the visualisation updates correctly and that the charts accurately capture the simulation data.

#### 3.1.1.4 Test 6. SeaGul

During simulation, the HIL rig is highly sensitive to external interference, such as people walking by. This is evident in Figure 3.5, where disturbances and noise are observed in the orientation graph.

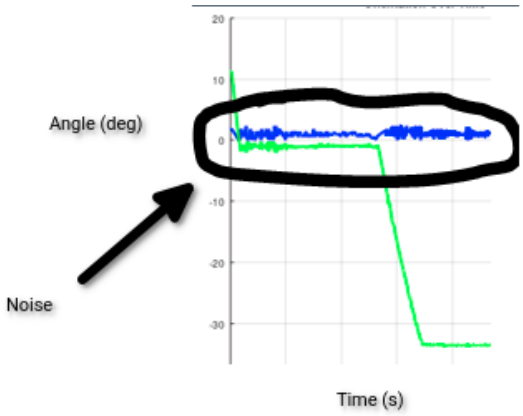



**Figure 3.5:** Highlight of how external interferences affect the HIL rig

The construction of the HIL rig makes it particularly vulnerable to edgeways movement, causing noise. However, under the right conditions, the graphs produced do not exhibit this external interference, as shown in Figure 3.6.

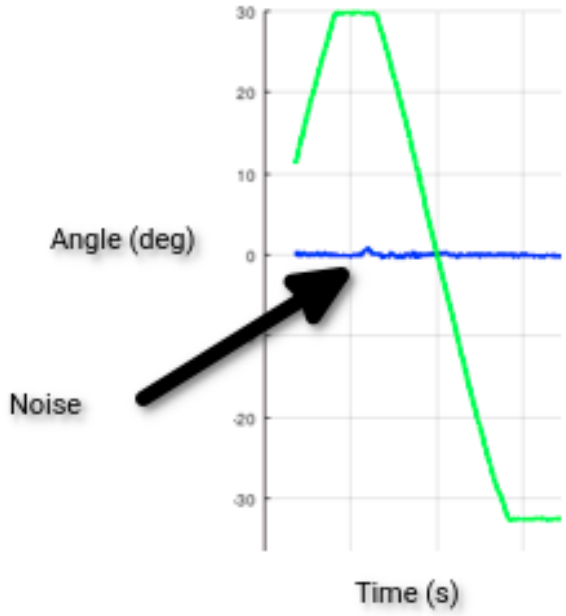



**Figure 3.6:** Highlight of performance with minimal external interferences

#### 3.1.1.5 Test 7: Current drawn

When switching from real sensors to simulated sensors, power consumption increases. The required voltage remains the same, as both types of sensors operate at 3.3V and are connected to the same power source. However, there is a difference in the current drawn. In Figure 3.7, the current values for the real sensors are shown on the left, while those for the simulated sensors are shown on the right.

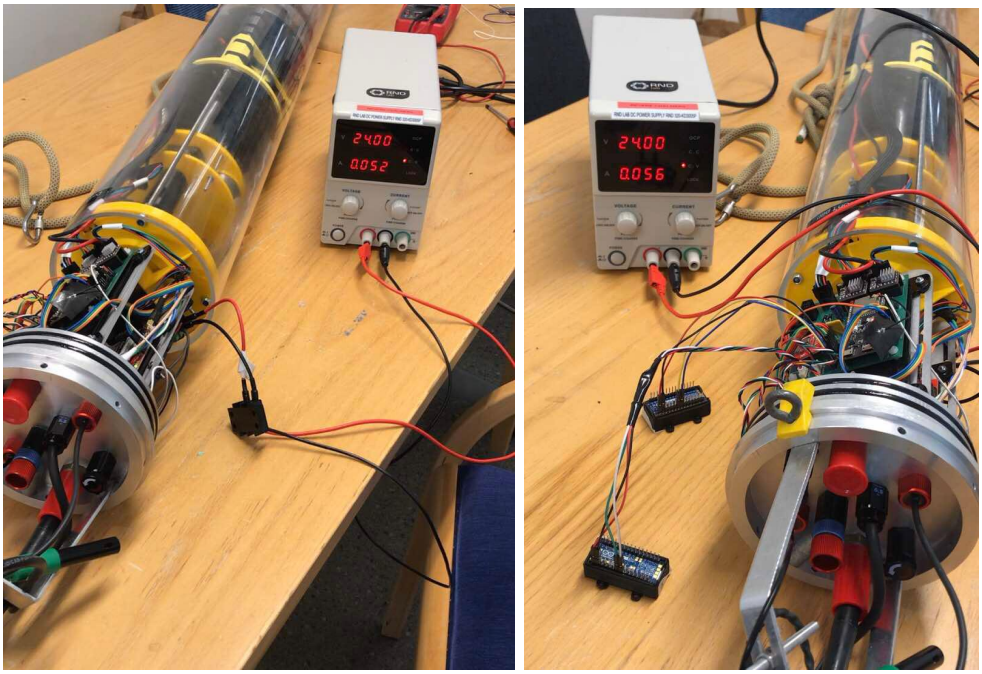



**Figure 3.7:** Current difference between real (to the left) and simulated sensors (to the right)

With real sensors connected, the system draws $0.052A$, whereas with simulated sensors, it draws $0.056A$, an increase of approximately 7.69%. In terms of power consumption, this corresponds to an increase from $1.248W$ to $1.344W$.

#### 3.1.1.6 Test 8: Accuracy of load cells

The load cell tests were conducted the same for each of the different weights. First, all load cells were calibrated to zero. Then, the known and verified weight was placed in the load cells for 10 minutes. After this period, the readings were compared with the initial values.

For the 100g weight test, the load cells initially display 99g, and after 10 minutes they still show approximately 99g, with occasional readings of 100g. This indicates that the load cells are accurate for lighter weights.

For the 750g weight test, the load cells showed a range of 755–758g, and after 10 minutes, the range remained unchanged, with a similar distribution of values. This indicates that while the load cells are less precise for medium weights, their readings

remain consistent and stable over time.

Initially, for the 2095g weight test, the load cells displayed a range of 2117–2119g, and after 10 minutes, the range remained consistent with a similar distribution of values. This suggests that while the load cells are less precise at higher weights, their readings remain stable and consistent over time

Table 3.1 summarises the results, including the calculated precision of the load cells based on how closely their readings match the verified weights.

| Weight (g) | After 0s | After 10 minutes | Accuracy (%) |
|---|---|---|---|
| 100 | 99.5 | 99.5 | $\approx 99,5$ |
| 750 | 756.5 | 756.5 | $\approx 99,1$ |
| 2095 | 2118 | 2118 | $\approx 98,9$ |

**Table 3.1:** Load cells test results

The test shows that as the weight increases, the load cells become less accurate but still maintain their initial readings consistently over time.

### 3.1.2 Examination of the HIL rigs output

To gather accurate feedback on the motor positions from the visualisation, it is essential that the interface is updated correctly.

#### 3.1.2.1 Test 9: CoG visualisation

This was tested by moving the motors to their maximum positions and observing how the CoG shifted. The results of Test 7 are shown in Figure 3.8.

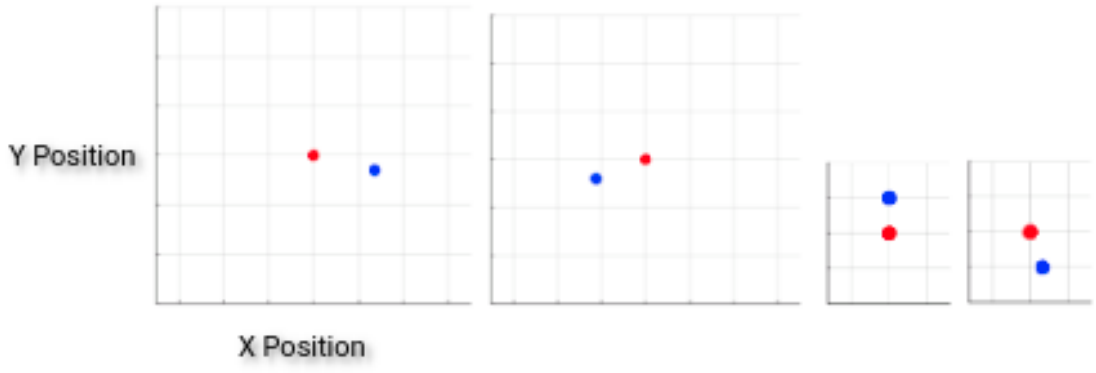



**Figure 3.8:** Batteries forward, backwards, left and right

This shows that the CoG updates correctly in the direction of motor movement. A comparison with Figure 2.14 provides further confirmation of the precision of these results.

#### 3.1.2.2 Test 10: HIL rig output range

The maximum positions correspond to the maximum pitch and roll angles achievable with the current setup. Table 3.2 presents the resulting angles that the HIL rig can produce.

| Max positive pitch | Max negative pitch | Max positive roll | Max negative roll |
|---|---|---|---|
| 40.56° | -40.56° | 3.34° | -3.34° |

**Table 3.2:** Maximum angles from maximum positions

These values are calculated using Equations 2.3, 2.4, 2.5, and 2.6, as referenced in the methodology to calculate $W_x$, $W_y$, $\theta$, and $\phi$. The calculations are based on the maximum resulting weight shift of $214g$ in the maximum forward and backward translation positions of the batteries, and the maximum weight shift of $14g$ when rotated to their maximum left and right positions. The distances between the load cells, $L_x$ and $L_y$, are set to $0.5m$ and $0.48m$, respectively, which correspond to their real-life configuration.

### 3.1.3 Examination of fully simulated dives

Finally, the results of the fully simulated dive test are presented. The current dive procedure for SeaGul operates as follows:

- **The user sets a target position for the stepper motors before diving.**
- **The stepper motors are adjusted to position SeaGul's CoG appropriately for diving**
- **The target potentiometer value is set**
- **A target depth is appointed**
- **The dive is initiated**

SeaGul begins the dive by opening the valve to reach the target potentiometer value, which depends on the initial amount of oil in the bladder. The motors remain fixed at their CoG position while SeaGul glides downward, entering the *gliding down* state.

Upon reaching the target depth, as specified by the user prior, SeaGul enters the *gliding up* state, during which it pumps oil into the bladder and repositions the stepper motors to the target position set before the dive.

Once SeaGul reaches the surface, it enters the *surface* state, moving the batteries fully to the front and maintaining positive buoyancy to stay afloat. The dive is then considered complete.

This is the same procedure SeaGul followed during real world dives at Bornö, where the collected data originated. The simulated dives follow the same sequence using the same programme, allowing direct comparison with the real dive data.

#### 3.1.3.1 8.5-meter simulated dive

In Figure 3.9 a simulated dive is shown at around 8 metres.

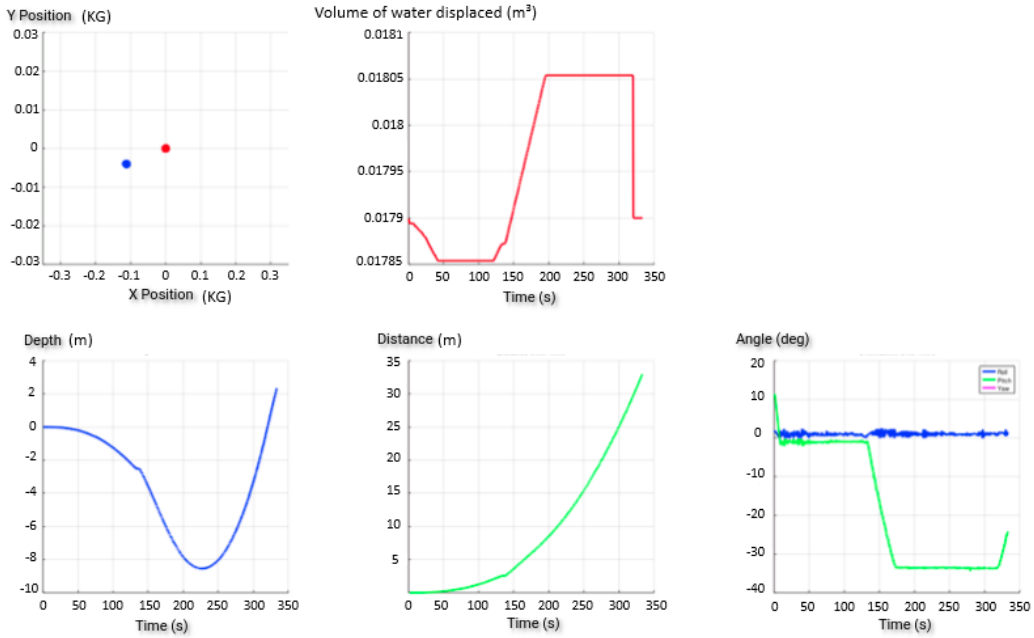



**Figure 3.9:** 8.5-meter simulated dive graphs

The entire dive took approximately 5 minutes and 25 seconds (325 seconds). The target depth was set at 2 metres. However, SeaGul reached a maximum depth of 8.5 metres, having 6.5 metres of turnaround.

SeaGul reached the target depth in 2 minutes and 5 seconds (125 seconds) and continued to descend, reaching its maximum depth after 3 minutes and 45 seconds (225 seconds). During the descent , the pitch angle started around -2°, remaining stable until about 2 metres, after which it began shifting as SeaGul transitioned into the *gliding up* state. The *gliding up* state was entered after passing the target depth. The turnaround time, from reaching the target depth to resurfacing, was 3 minutes and 20 seconds (200 seconds). During the ascent, the pitch angle gradually changed from -2°to steady -34°. The roll angle remained relatively stable, ranging between ±2°. The transition to negative buoyancy took 35 seconds, and from negative to positive buoyancy took approximately 70 seconds.

The total distance travelled during the dive was approximately 30 metres, with an overall average speed of 0.092 m/s. SeaGul resurfaced and entered the surface state, indicated by the pitch change at that point. The simulated buoyancy is also displayed on the graphs.

Two minor irregularities in the time axis are due to brief simulation freezes during the dive.

#### 3.1.3.2 14-meter simulated dive

Figure 3.10 shows another simulated dive, where the stepper motors were manually shifted using the web interface to the front and back positions to assist SeaGul during both the *diving down* and *diving up* states. In contrast, the first simulated dive without manual assistance provided only motor support during the *gliding up* state. This dive reached almost 14 metres of depth.

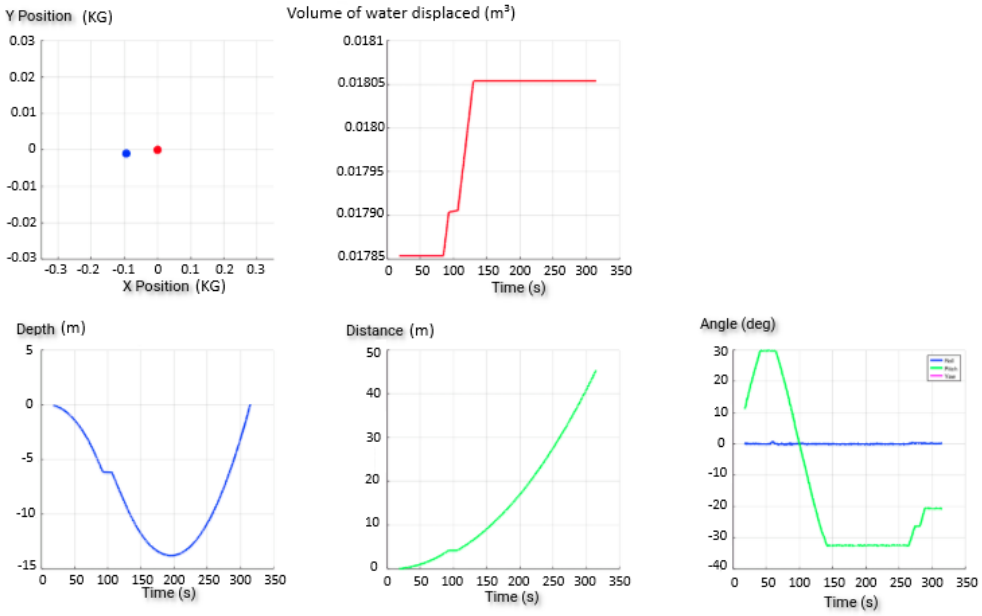



**Figure 3.10:** 14-meter simulated dive graphs

The entire dive took approximately 5 minutes and 10 seconds (310 seconds). The target depth for the dive was set to 5 metres. However, SeaGul reached a maximum depth of 14 metres, having a turn radius of 9 metres. SeaGul reached the target depth in 1 minute and 30 seconds (90 seconds) and continued to descend, reaching its maximum depth after 3 minutes and 10 seconds (190 seconds). The turnaround time, from reaching the target depth to resurfacing, was 3 minutes and 40 seconds (220 seconds).

During the *gliding down* state, the pitch angle remained steady around 30°, and during the *gliding up* state, the pitch angle remained steady around -32°. The roll angle stayed the same at a maximum of ±1°. Moving the motors backwards at approximately the 70 second mark. The motor shifted from fully forward to fully backward in about 75 seconds. SeaGul resurfaced and entered the surface state, as indicated by the change in pitch. The transitions between neutral, negative, and positive buoyancy are the same for the simulated dives.

The total distance travelled during the dive was approximately 45 metres, with an overall average speed of 0.145 m/s. This dive was faster than the one that reached 8 metres, thanks to the additional acceleration SeaGul gained by manually moving the batteries forward during descent and backward during ascent. In addition, starting

the motor shift approximately 20 seconds before reaching the target depth allowed for a faster turnaround.

One minor irregularity in the time axis is due to brief simulation freezes during the dive.

### 3.1.4 Examination of real dives

Figure 3.11 shows the real dive data from Bornö. Although some data points, such as the total distance travelled, are not included in these real dives, several key parameters overlap with the simulation data. These include pitch, roll, and depth. Enabling straightforward comparisons between simulated and real dives.

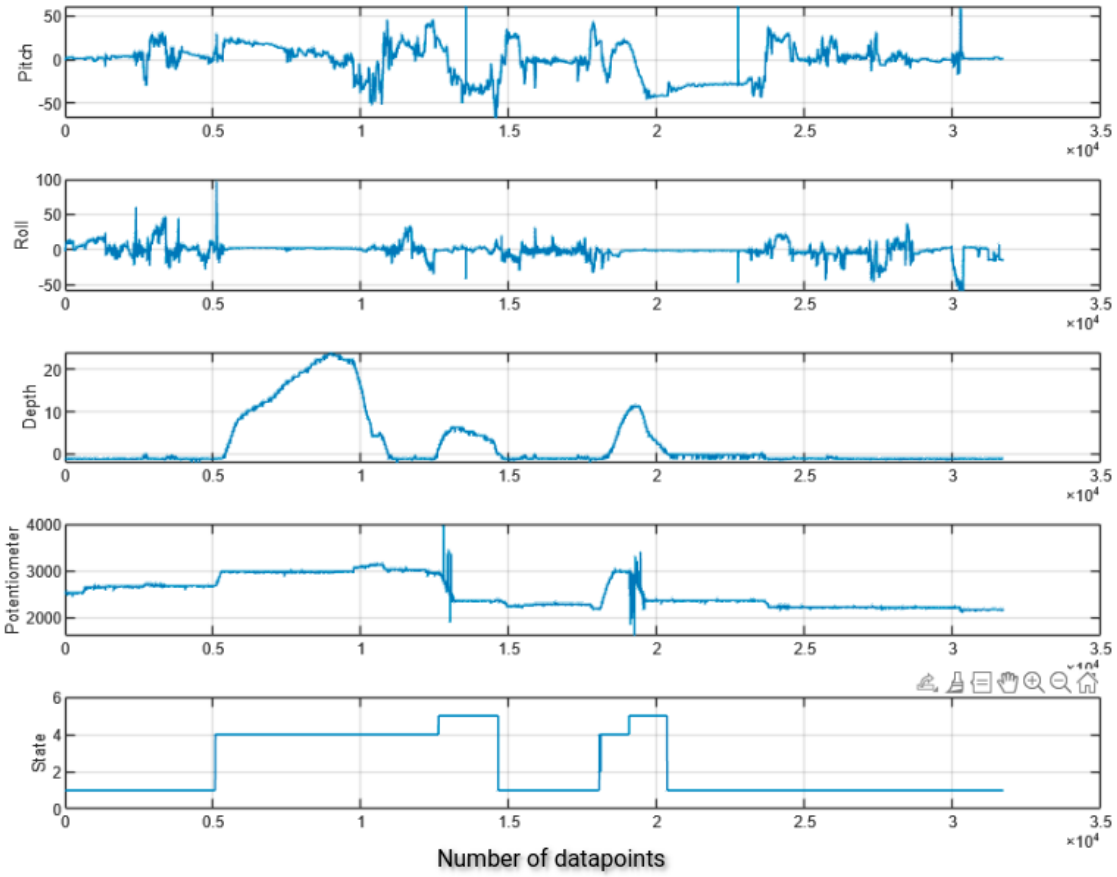



**Figure 3.11:** Data from real dives

Figure 3.12 presents the data from three dives, $23,37m$, $6,35m$, and $11m$, with a detailed explanation of the measurements and calculations provided for each dive in the following sections.

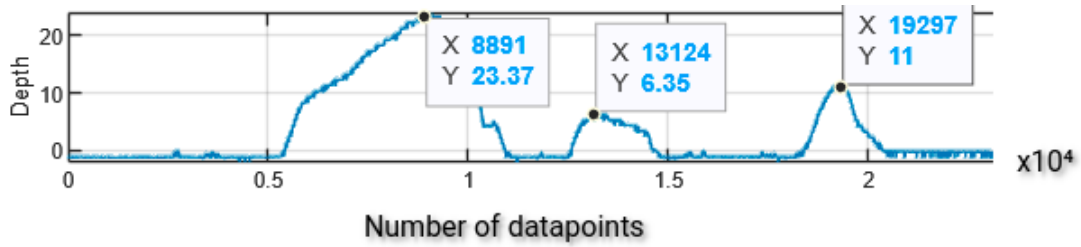



**Figure 3.12:** Three real dives at different depths

#### 3.1.4.1 23.4-meter real dive

The dive recorded 6,156 samples (from 11,068 to 4,912), corresponding to a duration of 10 minutes and 48 seconds (648 seconds).

The target depth was set at 25 metres. However, SeaGul reached a maximum depth of 23.4 metres. This suggests that the glider reached the seafloor before achieving the full target depth and, as a result, never entered the *gliding up* state. Consequently, it had to be manually retrieved. Due to this, data from the ascent phase are not usable, but the descent data remain valuable for analysis.

SeaGul reached its maximum depth in 6 minutes and 53 seconds (413 seconds). During descent, the pitch angle started between 18°and 22°until around 12 metres, then levelled to approximately 9°-13°in the last half of the dive. The roll remained relatively stable, ranging from 0°to 2°.

The potentiometer value, representing oil displacement, increased from 2,642 to 2,979 before stabilising around 3,000. In particular, the dive began when the potentiometer reached 2,979. It took approximately 18.4 seconds to go from neutral to negative buoyancy, with a total change of 337 in the potentiometer value.

The distance travelled during the descent was approximately 91.6 metres, resulting in an average descent speed of about 0.23 m/s.

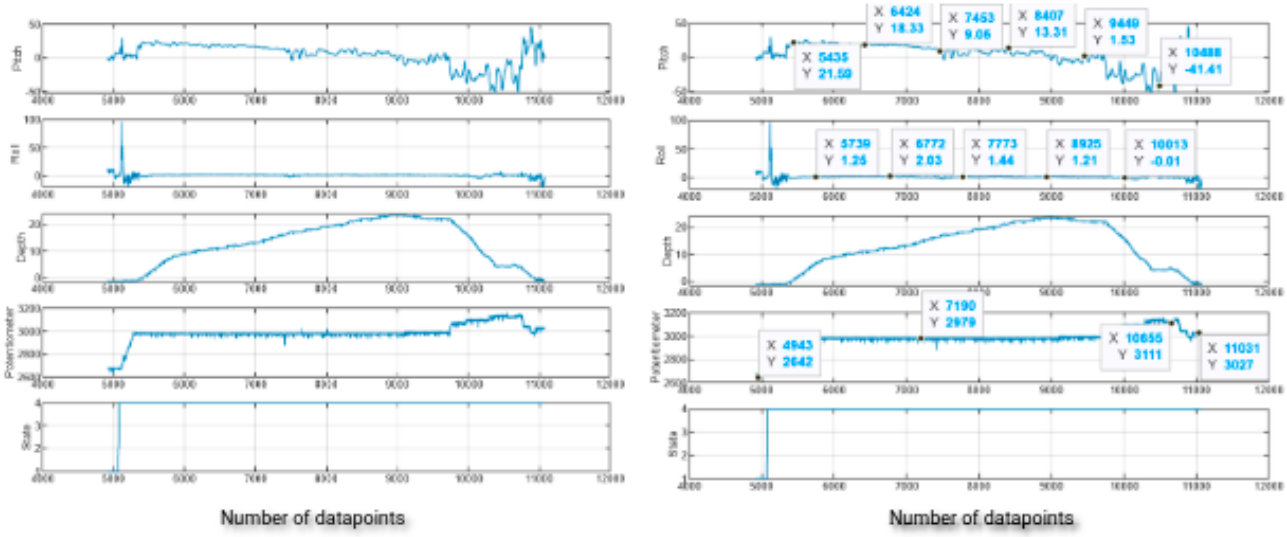



**Figure 3.13:** Detailed overview of 23,4m dive

#### 3.1.4.2 6.35-meter real dive

The dive recorded 2,580 samples (from 14,946 to 12,366), corresponding to a duration of 4 minutes and 32 seconds (272 seconds).

The target depth was set at 1 metre. However, SeaGul reached a maximum depth of 6.35 metres, having 5.35 metres of turn. Since SeaGul never entered the *gliding up* state during the previous dive (which reached 23.4 metres), this dive already began in the *gliding down* state, the potentiometer value starting at 2,963, as shown in Figure 3.14.

SeaGul reached the target depth in 14.6 seconds and continued to descend, reaching its maximum depth after 1 minute and 4 seconds (64 seconds). During descent, the pitch angle started around 23°, remaining stable until about 2 metres, after which it began to change as SeaGul transitioned to *gliding up* state. The *gliding up* state was entered after passing the target depth. The turnaround time, from reaching the target depth to resurfacing, was 3 minutes and 31 seconds (211 seconds). During the ascent, the pitch angle gradually changed from 23°to a steady -30°. The roll angle remained relatively stable, ranging between 2°and -4°.

The potentiometer value, representing oil displacement (and therefore buoyancy), began at 2,963 (indicating negative buoyancy). Upon reaching the target depth, SeaGul pumped oil into the bladder, reducing the value to 2,357 (positive buoyancy), which then stabilised around 2,300. The transition from negative to positive buoyancy took approximately 56 seconds, involving a total change of 606 units in

the potentiometer reading.

The descent covered approximately 17 metres, resulting in a descent speed of approximately 0.29 m/s. The pitch change from 23°to -30°took 78 seconds; however, it is not clear whether SeaGul moved forward during this transition. During the ascent, SeaGul travelled roughly 11 metres at a speed of 0.1 m/s.

If we account for the turnaround phase and assume that SeaGul moved forward during this period at an average speed between the descent and ascent speeds of 0.195 m/s, it would have travelled an additional 15 metres. This brings the estimated total distance travelled to approximately 43 metres, with an overall average speed of 0.16 m/s.

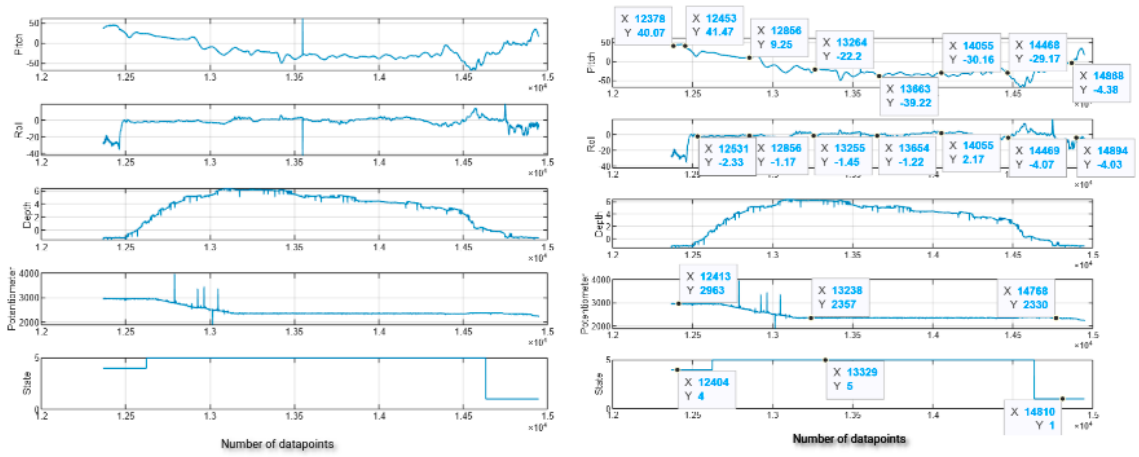



**Figure 3.14:** Detailed overview of 6,35m dive

### 3.1.4.3 11-meter real dive

The dive recorded 2,303 samples (from 20,349 to 18,046), corresponding to a duration of 4 minutes and 2 seconds (242 seconds).

The target depth was set to 10 metres. However, SeaGul reached a maximum depth of 11 metres, allowing for a one metre turnaround. From the pitch and roll graphs in Figure 3.15, some disturbances are noticeable near the surface, before SeaGul stabilises at approximately 1 metre depth.

SeaGul reached the target depth in 1 minute and 46 seconds (106 seconds) and continued to descend, reaching its maximum depth after 1 minute and 57 seconds (117 seconds). During descent, the pitch angle remained around 16–20°, remaining stable from 1 metre to 10 metres, after which it began to shift as SeaGul transitioned to the *gliding up* state. The *gliding up* state was entered shortly after surpassing the target depth. The turnaround time from reaching the target depth to resurfacing was 2 minutes and 16 seconds (136 seconds). During the ascent, the pitch angle gradually changed from 16–20°to steady between -40 and -42 °. The roll angle initially remained around -10°until it reached approximately 7.5 metres depth, where

it stabilised around -2°.

The potentiometer value, representing oil displacement (and therefore buoyancy), began at 2,195. Upon launching the dive, SeaGul released oil, increasing the value to 2,987 to achieve negative buoyancy. After reaching the target depth, SeaGul pumped the oil back into the bladder, lowering the value to 2,360 (positive buoyancy), which then stabilised around that value. The transition to negative buoyancy took 46 seconds and involved a change of 792 potentiometer units. The reverse transition, from negative to positive buoyancy, took approximately 60 seconds, with a change of 627 units.

The descent covered approximately 34 metres, resulting in a descent speed of about 0.3 m/s. The pitch transition from 18°to -41°occurred over 28 seconds; however, it is not clear whether SeaGul moved forward during this phase. During the ascent, SeaGul travelled approximately 13 metres at a speed of 0.17 m/s.

If we account for the turnaround phase and assume that SeaGul moved forward during this period at an average speed between the descent and ascent rates of 0.24 m/s, it would have travelled an additional 7 metres. This brings the estimated total distance travelled to approximately 54 metres, with an overall average speed of 0.22 m/s.

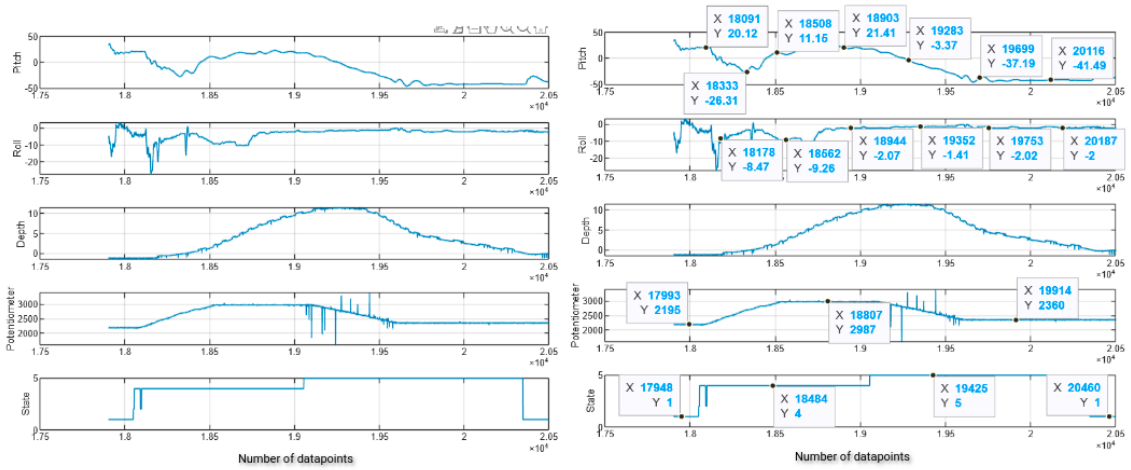



**Figure 3.15:** Detailed overview of 11m dive

### 3.1.5 Simulated and real dives overlapped

Below are overlapped plots of the simulated and real dives, to give a clear visual representation of the differences between the two types of dives. Furthermore, the x-axis is converted to time for real dives, making the two different types of dive share the same proportions and units. Thus, making it possible to compare. The comparison is done by showing depth, pitch, and roll separately from the three simulated dives, overlapped with the three real dives.

In Figures 3.16, 3.17 and 3.18 are the simulated and real depth represented and compared for dives to various water levels.

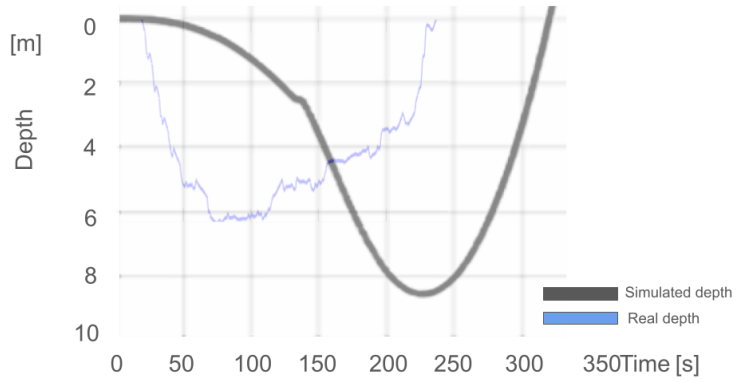



**Figure 3.16:** 6.35-meter real dive and 8.5-meter simulated dive overlapped depth

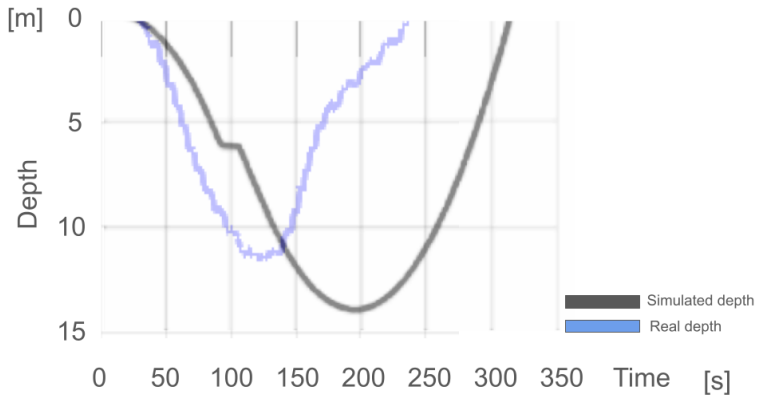



**Figure 3.17:** 11-meter real dive and 14-meter simulated dive overlapped depth

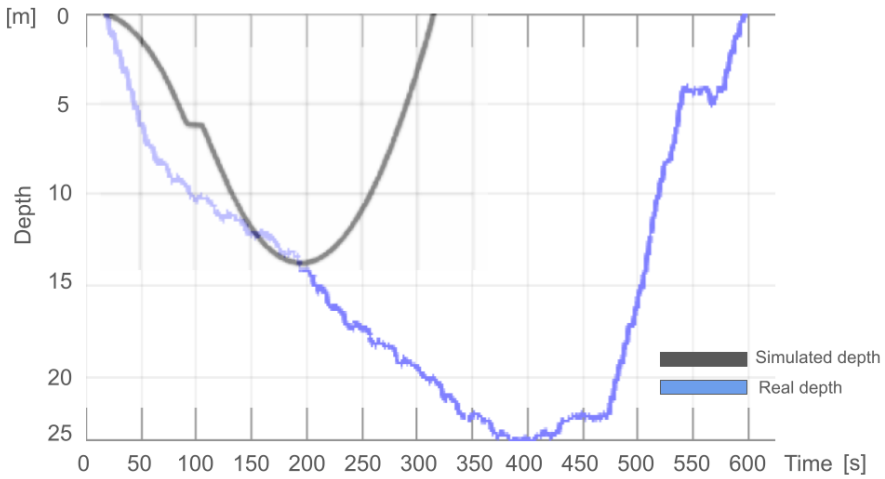



**Figure 3.18:** 23.4-meter real dive and 14-meter simulated dive overlapped depth

In Figures 3.19, 3.20 and 3.21 are the simulated and real pitch during dives to different depths.

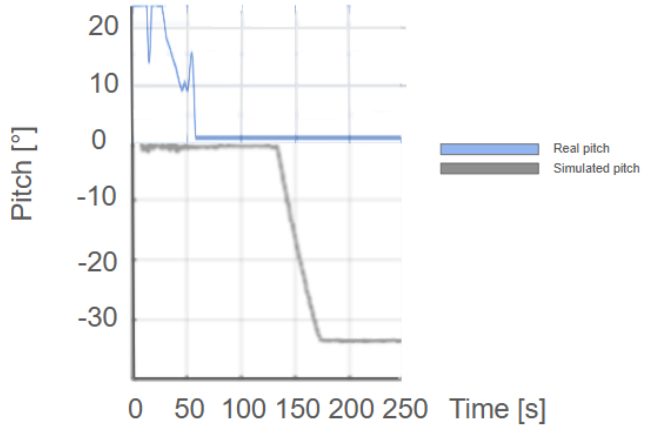



**Figure 3.19:** 6.35-meter real dive and 8.5-meter simulated dive overlapped pitch

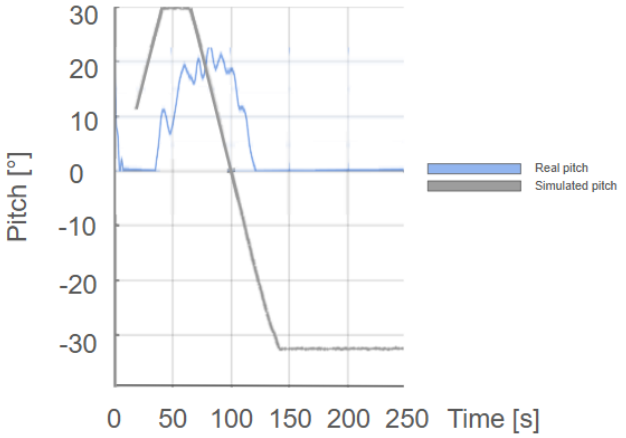



**Figure 3.20:** 11-meter real dive and 14-meter simulated dive overlapped pitch

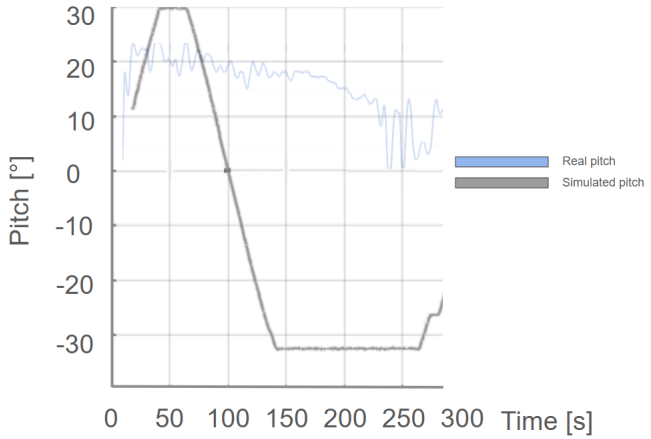



**Figure 3.21:** 23.4-meter real dive and 14-meter simulated dive overlapped pitch

In Figures 3.22, 3.23 and 3.24 are the roll during the two types of dives to different depths.

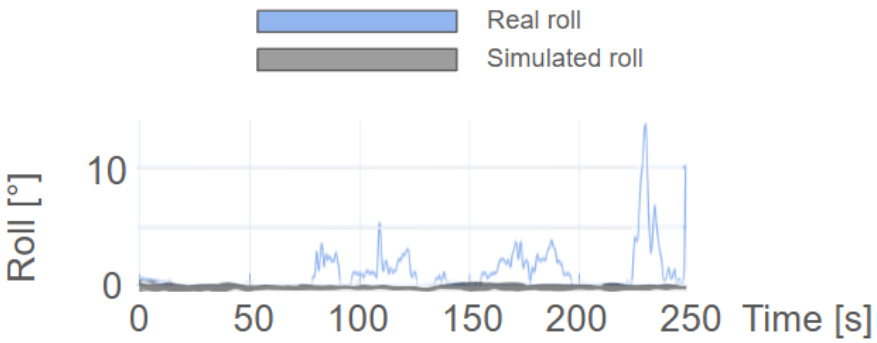



**Figure 3.22:** 6.35-meter real dive and 8.5-meter simulated dive overlapped roll

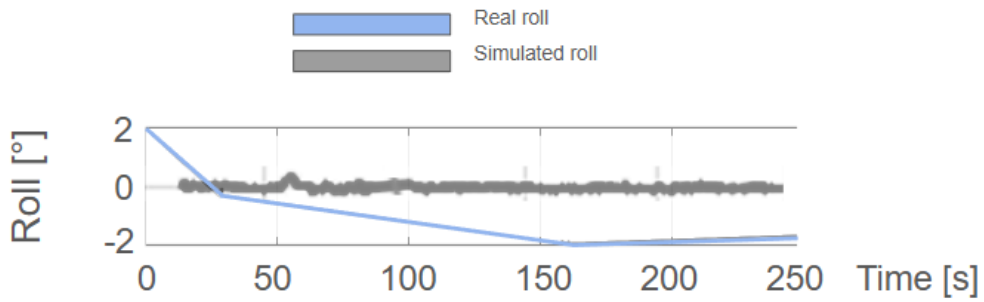



**Figure 3.23:** 11-meter real dive and 14-meter simulated dive overlapped roll

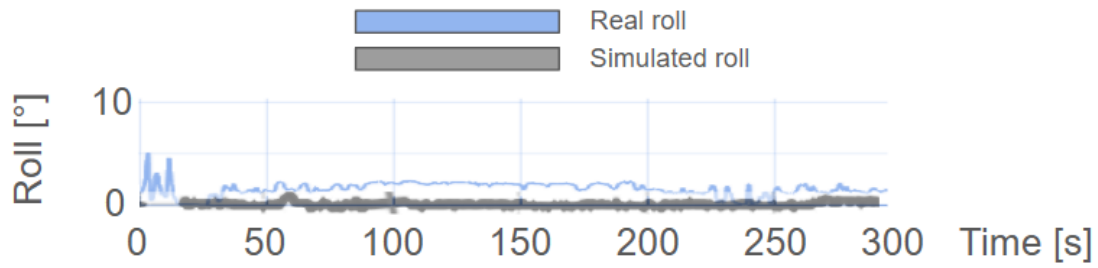



**Figure 3.24:** 23.4-meter real dive and 14-meter simulated dive overlapped roll

# 4

# Discussion

This chapter analyses and discusses the findings presented in the results section in greater depth.

## 4.1 Fully simulated dive

Notable differences exist between simulated and real dives. These discrepancies arise from several factors, including limitations in the simplified simulation model, real world disturbances, and the data collection method during the Bornö dives. Despite these shortcomings, there are clear similarities and valuable outcomes. Simulation has improved the code, facilitated neutral buoyancy calibration for SeaGul, and provided valuable pre dive insights, ultimately saving time and resources.

The comparison's focusses on the simulation. In simulated dives, pitch correlates directly with weight shift, determining the orientation of the SeaGul. Given the constraints of translational movement and the fixed weight configuration, the pitch range resembles that of real dives, excluding external forces.

A major limitation is the modelling of motor behaviour. In the simulation, changes in motor speed cause instantaneous shifts in pitch and acceleration. In contrast, real dives exhibit delayed responses from hydrodynamic forces and resistance, resulting in slower changes in pitch and acceleration.

The discrepancy is evident in the rotation modelling. The maximum and minimum rotation readings of the attachment rig restrict the angles to $\pm 3.34°$, while SeaGul may experience rotation angles exceeding $\pm 45°$. The simulation currently accounts only for the weight distribution and initial positioning, excluding the gradual spin from the motor rotation. Additionally, the absence of water currents in the model means that higher roll values cannot be induced naturally. Nevertheless, the simulated sensors offer a wide dynamic range ($-32{,}768$ to $32{,}768$, via unsigned 16-bit values), allowing simulation of SeaGul in all angles.

For a more realistic representation, forces such as drag, lift from wings, and buoyancy should be integrated. An important aspect to implement is the velocity limitation. Currently, SeaGul can accelerate infinitely in simulation, which is not feasible for longer dives.

Comparing the simulated dives to the real ones reveals that it is difficult to identify the source of differences, other than the simulation's shortcomings. No information was available about the number of steps the motors moved during real dives, whether SeaGul touched the bottom, and the presence of human interference, such as pulling SeaGul up or initiating dives in different states. In addition, the motors were initially manually set to achieve a forward pitch, and the estimations of speed, distance, and time were calculated.

Direct comparisons between simulated and real dives reveal several differences. The simulated speeds ranged between 0.09 and 0.145 m/s, whereas the real dive speeds reached up to 0.23 m/s (23.4 metres dive), although only the descent data are considered reliable. Without deceleration of the turnaround, the speed increases. The 6.35 metres dive showed a speed of 0.16 m/s, closer to the simulated values, but SeaGul began this dive already negatively buoyant, skipping the initial acceleration phase. The 11 metres dive recorded 0.22 m/s, with minimal turnaround.

These variations stem from the initial transition from idle to descent, the simulation model's constraints, and unknown real-world factors. In the 8.5 metre simulated dive, a depth of 2 metres was reached in 125 seconds, while the remaining 6.5 metres took only 100 seconds. In contrast, the real 6.35 metre dive reached a depth of 1 metre in just 14.6 seconds, completing the remainder in 49.4 seconds.

The 14-metre simulated dive reached 5 metres in 90 seconds and 14 metres (9 metres deeper) in an additional 100 seconds. The 11-metre dive reached the 5-metre target in 106 seconds, likely due to pumping 800 oil units, compared to 300 units for the 23.4-metre dive, which was estimated to take about 46 seconds instead of 18.4 seconds, an increase of 27.6 seconds. The maximum depth of this dive (1 metre above the target) was reached 11 seconds later.

Given the real diving conditions, it is difficult to accurately match or compare timings. However, adding estimated durations of the real dives, 272 seconds (6.35 metre) and 242 seconds (11 metre), still fall significantly below the simulation's times of 325 seconds (8.5 metre) and 310 seconds (14 metre). These disparities emphasise the slower initial acceleration and buoyancy shifts of the simulation.

In the simulation, the shift from neutral to negative buoyancy takes about 35 seconds, and the return to positive buoyancy takes 70 seconds. In contrast, real dives show transitions in 18.4 seconds (neutral to negative in 23.4 m) and 56–60 seconds (negative to positive in 6.35 metres and 11 metres dives). This slower change in the simulated forces delays the response to dive.

The precise amount of oil displaced in real dives is unknown, adding further uncertainty. It is possible that SeaGul pumped more oil than simulated, creating stronger upward/downward forces.

Similarly, the exact step counts of the stepper motors during real dives are unclear. Although motor speed is consistent across runs due to shared control code, during simulated dives, the motors shifted 35,000 to 40,000 steps over 75 seconds, resulting in pitches of $+30°$ (gliding down) and $-32°$ (gliding up). Because it is unclear how far the motors moved in the real dives, it means it is possible they moved less, indicated by the usual 20 degree angle during descent in all dives, the same amount backward, resulting in around -30 degrees, as indicated by the 6.35 metre, or more, as indicated by the pitch angle of around -40 degrees during the 11 metre dive. This variability impacts the amount of turnaround, speeds, time, and acceleration.

If motor and buoyancy shifts occur more slowly, the turnaround time will increase because of the delayed change in forces acting on the glider. This slower force transition extends the time to reverse direction and affects the overall duration and speed of the dive. These factors significantly influence the turnaround time and the total time required for a dive.

Roll angles were similar in both scenarios: $\pm 2$ in simulations and $-2°$ to $4°$ in real dives. Turnaround depths were comparable, 5.35 metres (6.35 metres dive) versus 6.5 metres (8.5 metres simulation). When the motors were set forward at the beginning of the 14m simulation, they reached the 5-metre target in 90 seconds, close to the 11-metre real dive 78 seconds (assuming 300 units of oil were pumped instead of 800).

Interestingly, longer dives were faster than shorter ones. This is due to the extended acceleration during deeper dives. However, turnaround distance decreased despite increasing acceleration in real dives, likely due to disturbances or environmental factors.

The distance travelled during the simulated dives is around 75-83% of the real dive, likely because the simulation has perfect pitch, and in the real dives fluctuates and therefore results in a longer distance travelled.

The differences between the two simulated dives demonstrate clear opportunities to improve the control code. For example, initiating battery shifts before reaching depth or adjusting target depths to account for turnaround distances are strategies that emerge from simulation insights.

The HIL rig provides significant benefits. It enables SeaGul to be tested beyond its mechanical limits and provides insight into previously inaccessible diving behaviour. The system enables targeted testing across various pitch and roll combinations, serving as a static controlled platform for specific cases. It can also be used to optimise weight distribution and achieve neutral buoyancy for SeaGul or other gliders.

## 4.2 Load cell accuracy

The heaviest weights had the lowest accuracy at 98.9%. The relatively high weight of the SeaGul glider of approximately 17.9 kg likely amplifies this effect. The stable values have minimal impact on the simulation. The main point of interest is the weight shift caused by the batteries, which is not an issue. According to Figure **??**, this weight shift reaches a maximum of about 214 g, resulting in an accuracy between 99.5% and 99.1%, which is deemed acceptable given the small weight change. The measured values range from 0 g to a maximum of 214 g, with smaller values showing greater precision.

## 4.3 Increase in current drawn

The current draw increased by 0.04A (from 0.052A to 0.056A). Although seemingly negligible initially, this becomes significant over a long duration AUG deployment when evaluated in watt hours (Wh). Even if a full deployment cycle lasting several days or weeks is simulated, it remains feasible, though less applicable than a real cycle, it could still provide valuable insights beyond power consumption.

## 4.4 Simulation of parts

Table 4.1 presents the SeaGul components, displaying both simulated and non-simulated results.

| Simulated | Not simulated |
|---|---|
| Depth sensor | Motors |
| Orientation sensor | Potentiometer sensor |
| | Valve |
| | Pump |
| | Dropweight |
| | Thruster |
| | Conductivity sensor |
| | Temperature sensor |

**Table 4.1:** Simulated and not simulated components

Most components in the non-simulated system operate autonomously based on code defined parameters. These include motors, potentiometers, valves, pumps, drop weights, and thrusters. Simulating the IMU and depth sensor is crucial for system functionality. Without the simulated sensors, the depth sensor would output 0 metres, preventing SeaGul from initiating depth dependent behaviours, as many components respond differently based on depth readings. Similarly, without simulated IMU data, being stationary during simulation would register near zero values for pitch and roll. This would inhibit battery movement and eliminate essential feedback on pitch and roll control. Therefore, it is essential to provide SeaGul with

simulated IMU and depth data to enable accurate behaviour emulation to collect pitch values required for acceleration, dive dynamics and send feedback from simulation to close the loop.

## 4.5 HIL as a PD tool

Although technically feasible, simulating an entire mission using the HIL rig is not always the most efficient use of the platform. Full mission simulations take as long as real operations and offer diminishing returns in terms of time to insight. The HIL rig is most valuable for rapid and focused testing, such as performing a single dive cycle, validating subsystem functions like surface communication or drop weight release, and debugging new control sequences. This allows developers to observe in real time whether SeaGul responds correctly at key decision points and in response to control signals, which is crucial when testing new functionality or control logic.

From a hardware perspective, the use of the Raspberry Pi offers a flexible and cost-effective foundation for system expansion. Its compatibility with various peripheral devices, including additional sensors or modules, allows testing of more complex hardware configurations. This enables incremental expansion and modular testing of new HIL rig components.

The HIL rig effectively tests individual dives and analyses the impact of control protocols on SeaGul's behaviour. Although longer simulations with multiple dive cycles are possible, they become time consuming and approach the duration of real-world testing. In such cases, the benefit lies more in replicating specific conditions or faults than in executing complete mission profiles. For development, focused and rapid subsystem tests or control routines are more efficient and valuable.

Compared to the other PD methods discussed in the introduction, the advantages of HIL are clear. For example:

- National Institute of Standards and Technology [2017] is based on a comprehensive physical test bed that requires significant resources and setup.
- Herman [2021], being entirely simulation based, fails to capture real world physical effects and error sources that influence system behaviour.
- Higuera et al. [2022], although conducted in water, uses a different vehicle with potentially different physical characteristics, reducing the reliability and transferability of the results to other systems

.

The power of the HIL rig as a PD tool lies in its integration of simulation with real hardware interaction. Provides development and testing capabilities in the same system, enabling faster iteration cycles. Insights can be gained quickly and accurately, reducing lead times, speeding up turnaround, and enabling earlier issue detection of issues, key benefits in product development or any agile or lean process.

One limitation is that the HIL rig is tailored to SeaGul and its specific sensor and hardware configurations. To promote broader applicability, a modular library

or development framework could be built. As many AUV systems share common components, such a library would facilitate faster integration and testing of similar systems across platforms, making the HIL rig a versatile and reusable PD asset.

# 5
# Conclusion

This chapter addresses the research questions and summarises the key aspects of the project.

## 5.1 Summary of Findings

The HIL rig runs correctly, and all hardware components work properly. Most improvement areas are within the simulation, despite the sensitivity to disturbances. Although the simulation does not accurately reflect reality, it provides valuable insight for developing SeaGul.

Differences in dive speeds, turnaround times, and acceleration between simulations and real dives underscore the need for improved modelling of buoyancy dynamics, drag, and velocity constraints. Additionally, limited data from actual dives, including unknown motor steps and oil displacement volumes, hinder direct comparison. Trends and patterns from dives indicate that the simulation effectively models key behaviours, including SeaGul's responses upon reaching target depth, surface behaviours, specific orientation actions, and battery translation and rotation. It can also provide approximate estimates of speed, pitch, roll, distance, and time, though not with high precision. This information is valuable and supports the success of real-world dives.

New real dives are needed to collect more data to improve the simulation.

## 5.2 Answering the Research Questions

- **Can SeaGul autonomously complete a simulated dive?**
  Yes, it can. Its autonomous responses actively influence the simulation.
- **Can SeaGul autonomously collect data during a simulated dive?**
  Yes. The simulated depth sensor can receive temperature data, and the real onboard sensors continue to operate and collect information as well.
- **Are HIL rigs useful for glider related product development?**
  Yes. Even a simplified model provides valuable insights, enabling rapid testing and offering an overview that can increase the likelihood of a successful real-world dive before committing physical resources.
- **Is SeaGul a feasible platform for data collection from a product development perspective?**
  Yes. Although it may not be suitable for collecting all types of data (e.g., video), it is highly energy- and cost efficient for parameters like temperature and salinity. It supports long deployment times and operates autonomously.

The development and evaluation of SeaGul's simulation and HIL rig have demonstrated both strengths and limitations. Although the simplified simulation diverges from actual diving behaviour due to factors such as missing forces, idealised motor responses, and varying environmental effects, it still provides significant value. It is effective for testing and improving control code, calibrating neutral buoyancy, and providing pre dive insights that save time and resources during real deployments.

The HIL rig is a valuable PD tool. It enables rapid, targeted testing of individual dive cycles, subsystem responses, and control strategies without requiring full scale missions. Simulating sensor feedback in a controlled environment accelerates development and aids in debugging and verification. The HIL rig offers a practical middle ground, balancing realism and efficiency compared to purely physical or fully simulated approaches.

The current system is tightly coupled to the specific SeaGul configuration. To enhance usefulness, a modular framework should be developed for greater compatibility across AUV platforms. This would transform the HIL rig from a custom test bed into a scalable, reusable asset for the development of underwater vehicles.

In summary, simulation and HIL systems provide a powerful combination for iterative development, control tuning, and design validation, although they do not replace field testing. They reduce lead times, support agile workflows, and improve the reliability of deployed systems.

### 5.2.1 Future work

A list of improvements for further system development.

- Integrate a sensor around the bladder to measure pressure changes and estimate the buoyancy of SeaGul in neutral, positive, or negative states in real time.
- Enhance the simulation by incorporating forces such as wing force, drag force, currents, and bladder-based buoyancy. Implement realistic speed limits and allow manual simulation to start instead of auto start on boot.
- Do more dives with the intent to gather data for simulation, develop a programme to analyse the new dive data by calculating speeds and distances between each data point, enabling more precise performance evaluations.
- Simulate additional sensors to allow testing of more complex behaviours and edge cases without requiring real world deployments.

# Bibliography


Di Bao, Rui Yang, Y. Ma, Benoit Clement, Ali Mansour, D. Hou, and Ming Li. Hardware-in-the-loop simulation applied to auv control. In *Proceedings of the 2018 Chinese Automation Congress (CAC)*, pages 1009–1013, Nov 2018. doi: 10.1109/CAC.2018.8623733.

BlueRobotics. BlueRobotics_KellerLD_Library: Arduino library for the Keller 4LD – 9LD I²C pressure and temperature sensors. `https://github.com/bluerobotics/BlueRobotics_KellerLD_Library/tree/master`, 2023. Latest release version 1.1.2 (Feb 16, 2023); Accessed: 2024-11-16.

J. Busquets-Mataix, J. V. Busquets-Mataix, and D. Busquets-Mataix. Combined gas-fluid buoyancy system for improved attitude and maneuverability control for application in underwater gliders. *IFAC-PapersOnLine*, 48(2):281–287, 2015. ISSN 2405-8963. doi: 10.1016/j.ifacol.2015.06.046. URL `https://www.sciencedirect.com/science/article/pii/S2405896315002852`.

Cornelia Falkhage, Joel Janson, Nellie Karlsson, Adam Oskarsson, Marcus Petersson, and Cevin Åström. Utveckling av elektronik- och säkerhetssystem för reveres glider seagul, 2024. Bachelor's thesis (unpublished).

P. Herman. Preliminary design of the control needed to achieve underwater vehicle trajectories. *Journal of Marine Science and Technology*, 26:986–998, 2021. doi: 10.1007/s00773-020-00784-9.

C. Higuera, J. Sandoval, L. N. Coria, and E. Bugarin. An autonomous unmanned underwater control test vehicle: platform description and experiments. *Journal of Mechanical Science and Technology*, 36(1):395–405, 2022. doi: 10.1007/s12206-021-1238-0.

Andreas Jonsson, Andreas Järlebratt, Benjamin Rask, Maja Sunesson, Oscar Schyum, and Oskar Persson. Design och konstruktion av en autonom undervattensglidare, 2023. URL `https://odr.chalmers.se/items/c17e15ff-32b3-4988-97ab-8059e0180a2e`.

KELLER AG für Druckmesstechnik. *Description of the Communication Protocol: Series 4 LD... 9 LD OEM Pressure Transmitter (Version 2.6)*, August 2017. Version 2.6, published August 30 2017; Accessed: 2024-12-03.

Yeojun Kim, Samuel Tay, Jacopo Guanetti, Francesco Borrelli, and Ryan Miller. Hardware-in-the-loop for connected automated vehicles testing in real traffic, 2019. URL `https://arxiv.org/abs/1907.09052`.

T. Liblik, J. Karstensen, P. Testor, P. Alenius, D. Hayes, S. Ruiz, K. J. Heywood, S. Pouliquen, L. Mortier, and E. Mauri. Potential for an underwater glider component as part of the global ocean observing system. *Methods in Oceanography*, 17:50–82, 2016. ISSN 2211-1220. doi: 10.1016/j.mio.2016.05.001. URL https://www.sciencedirect.com/science/article/pii/S2211122016300056.

Yifan Liu, Jialei Zhang, Xianbo Xiang, and Jiaxun Liu. Design, implementation and verification of hardware-in-the-loop control system for work-class rovs. *Ocean Engineering*, 313:119605, 2024. ISSN 0029-8018. doi: 10.1016/j.oceaneng.2024.119605. URL https://www.sciencedirect.com/science/article/pii/S0029801824029433.

National Institute of Standards and Technology. Aquatic vehicle tests: Development of standard test methods for underwater remotely operated vehicles. https://www.nist.gov/el/intelligent-systems-division-73500/standard-test-methods-response-robots/aquatic-vehicle-tests, 2017. Published approximately February 2017; Accessed: 2024-11-16.

NOAA Atlantic Oceanographic and Meteorological Laboratory (AOML). Underwater gliders. PDF, 1 page, posted on NOAA AOML website, July 2022. URL https://www.aoml.noaa.gov/wp-content/uploads/2022/07/Glider-One-Page_July2022.pdf. "Glider-One-Page_July2022.pdf".

Marcel Otte, Fabian Leimgruber, Roland Brundlinger, Sebastian Rohjans, Aadil Latif, and Thomas I. Strasser. Hardware-in-the-loop co-simulation based validation of power system control applications. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, page 1229–1234. IEEE, June 2018. doi: 10.1109/isie.2018.8433761. URL http://dx.doi.org/10.1109/ISIE.2018.8433761.

P. N. Patil. A comparative analysis of raspberry pi hardware with arduino, phidgets, beaglebone black and udoo, 2016.

F. Prochazka, S. Krüger, G. Stomberg, and et al. Development of a hardware-in-the-loop demonstrator for the validation of fault-tolerant control methods for a hybrid uav. *CEAS Aeronautical Journal*, 12:549–558, 2021. doi: 10.1007/s13272-021-00509-7.

Daniel Rudnick, Russ Davis, Charles Eriksen, David Fratantoni, and Mary Jane Perry. Underwater gliders for ocean research. *Marine Technology Society Journal*, 38:73–84, 2004. doi: 10.4031/002533204787522703.

Z. Y. Shao, Y. Feng, J. Jin, and J. N. Bian. Study on hardware-in-the-loop simulation system for navigation and control of uuv. In *Advanced Materials Research*, volume 875–877, pages 2196–2200. Trans Tech Publications, Ltd., 2014. doi: 10.4028/www.scientific.net/amr.875-877.2196.

SparkFun Electronics. Sparkfun_icm-20948_arduinolibrary: Arduino support for the icm-20948 9-dof imu with digital motion processor. https://github.com/sparkfun/SparkFun_ICM-20948_ArduinoLibrary, 2025. Latest release v1.3.2 (May 7, 2025); Accessed 2024-12-2.

Stephen Wood and Cheryl Mierzwa. State of technology in autonomous underwater gliders. *Marine Technology Society Journal*, 47:84–96, 2013. doi: 10.4031/MTSJ.47.5.4.

Zhizun Xu, Yang Song, Jiabao Zhu, and Weichao Shi. Ugsim: Autonomous buoyancy-driven underwater glider simulator with lqr control strategy and recursive guidance system, 2025. URL `https://arxiv.org/abs/2501.17851`.

Z. Zhang, W. Mi, J. Du, Z. Wang, W. Wei, Y. Zhang, Y. Yang, and Y. Ren. Design and implementation of a modular uuv simulation platform. *Sensors*, 22(20):8043, 2022. doi: 10.3390/s22208043.

# A

# HIL rig code

## A.1 Load sensor code

**Listing A.1:** Main code Arduino UNO

```
#include <Adafruit_NAU7802.h>
#define TCAADDR 0x70

void nauSetLDO(Adafruit_NAU7802 x) {
  //Serial.print("LDO voltage set to ");
  x.setLDO(NAU7802_3V3);
}

void nauSetGain(Adafruit_NAU7802 x) {
  x.setGain(NAU7802_GAIN_64);
}

void nauSetRate(Adafruit_NAU7802 x) {
  x.setRate(NAU7802_RATE_10SPS);
}

void nauCalibration(Adafruit_NAU7802 x) {
  for (uint8_t i = 0; i < 10; i++) {
    while (!x.available()) delay(1);
    x.read();
  }

  while (!x.calibrate(NAU7802_CALMOD_INTERNAL)) {
    //Serial.println("Failed to calibrate internal offset, retrying
        !");
    delay(1000);
  }
  //Serial.println("Calibrated internal offset");

  while (!x.calibrate(NAU7802_CALMOD_OFFSET)) {
    //Serial.println("Failed to calibrate system offset, retrying
        !");
    delay(1000);
  }
  //Serial.println("Calibrated system offset");
}

void tcaselect(uint8_t i) {
  if (i > 7) return;
```

```
38    Wire.beginTransmission(TCAADDR);
39    Wire.write(1 << i);
40    Wire.endTransmission();
41  }
42
43  Adafruit_NAU7802 nau1, nau2, nau3, nau4;
44  int32_t preSensor1 = 0, preSensor2 = 0, preSensor3 = 0, preSensor4
        = 0;
45  void setup() {
46    Serial.begin(115200);
47    Wire.begin();
48    // Serial.println("\nTCAScanner ready!");
49    for (uint8_t t = 0; t < 8; t++) {
50      tcaselect(t);
51      for (uint8_t addr = 0; addr <= 127; addr++) {
52        if (addr == TCAADDR) continue;
53
54        Wire.beginTransmission(addr);
55        if (!Wire.endTransmission()) {
56        }
57      }
58    }
59
60    tcaselect(1);
61    //Serial.println("Starting: NAU 0");
62    if (!nau1.begin()) {
63      //Serial.println("Failed to find NAU7802: 0");
64      while (1) delay(10);  // Don't proceed.
65    }
66    //Serial.println("Found NAU7802: 0");
67    nauSetLDO(nau1);
68    nauSetGain(nau1);
69    nauSetRate(nau1);
70    nauCalibration(nau1);
71    // Serial.println("NAU 0 Calibrated and ready!");
72
73    tcaselect(2);
74    //Serial.println("Starting: NAU 1");
75    if (!nau2.begin()) {
76      // Serial.println("Failed to find NAU7802: 1");
77      while (1) delay(10);  // Don't proceed.
78    }
79    // Serial.println("Found NAU7802: 1");
80    nauSetLDO(nau2);
81    nauSetGain(nau2);
82    nauSetRate(nau2);
83    nauCalibration(nau2);
84
85    // Serial.println("NAU 1 Calibrated and ready!");
86    tcaselect(3);
87    // Serial.println("Starting: NAU 4");
88    if (!nau3.begin()) {
89      // Serial.println("Failed to find NAU7802: 4");
90      while (1) delay(10);  // Don't proceed.
91    }
92    //  Serial.println("Found NAU7802: 4");
```

```
93    nauSetLDO(nau3);
94    nauSetGain(nau3);
95    nauSetRate(nau3);
96    nauCalibration(nau3);
97    // Serial.println("NAU 4 Calibrated and ready!");
98
99    tcaselect(4);
100   // Serial.println("Starting: NAU 6");
101   if (!nau4.begin()) {
102     // Serial.println("Failed to find NAU7802: 6");
103     while (1) delay(10);  // Don't proceed.
104   }
105   // Serial.println("Found NAU7802: 6");
106   nauSetLDO(nau4);
107   nauSetGain(nau4);
108   nauSetRate(nau4);
109   nauCalibration(nau4);
110   // Serial.println("NAU 6 Calibrated and ready!");
111 }
112 void loop() {
113   Serial.flush();
114   if (nau1.available() && nau2.available() && nau3.available() &&
          nau4.available()) {
115     tcaselect(1);
116     int32_t val1 = nau1.read();
117     tcaselect(2);
118     int32_t val2 = nau2.read();
119     tcaselect(3);
120     int32_t val3 = nau3.read();
121     tcaselect(4);
122     int32_t val4 = nau4.read();
123
124     int32_t sensor1 = val1 / 100;
125     int32_t sensor2 = val2 / 100;
126     int32_t sensor3 = val3 / 100;
127     int32_t sensor4 = val4 / 100;
128
129     preSensor1 = sensor1;
130     preSensor2 = sensor2;
131     preSensor3 = sensor3;
132     preSensor4 = sensor4;
133
134     Serial.print(sensor1);
135     Serial.print("␣");
136     Serial.print(sensor2);
137     Serial.print("␣");
138     Serial.print(sensor3);
139     Serial.print("␣");
140     Serial.println(sensor4);
141
142   }
143    else {
144     Serial.print(preSensor1);
145     Serial.print("␣");
146     Serial.print(preSensor2);
147     Serial.print("␣");
```

```
148     Serial.print(preSensor3);
149     Serial.print("␣");
150     Serial.println(preSensor4);
151     delay(50);
152   }
153 }
```

## A.2 Simulated depth sensor

**Listing A.2:** Main code Arduino NANO

```
1  #include <ArduinoBLE.h>
2  #include <Wire.h>
3  // I2C Address of the Bar100 sensor
4  #define SENSOR_ADDRESS 0x40
5  // Memory address locations
6  #define LD_SCALING1 0x13
7  #define LD_SCALING2 0x14
8  #define LD_SCALING3 0x15
9  #define LD_SCALING4 0x16
10 #define LD_SCALING0 0x12
11 #define LD_CUST_ID0 0x00
12 #define LD_CUST_ID1 0x01
13 double desiredDepth = 0;       //Max is 151 bar/around 1510m
14 uint32_t P_max = 0x42C80000;  //100 in IEEE-754 singel
15 //uint16_t P_max_low =0x0000;
16 uint16_t P_min = 0x00000000;  //0 in IEEE-754 singel
17 //uint16_t P_min_low =0x0000;
18 uint16_t cust_id1 = 0b0000010011010010;  //arbitrary binary number
      to satisfy the libary
19 uint16_t cust_id0 = 0b1011010100101100;  //45, just need to be
      smaller than 63 for sensor to be connected
20 uint16_t scaling0 = 0b0000000000000001;  //Change last number to 1
      if you want to have mode 1, 0 for mode 0
21 // Define service and characteristic
22 BLEService myService("19b10010-e8f2-537e-4f6c-d104768a1214");
23 BLECharacteristic myCharacteristic("19b10011-e8f2-537e-4f6c-
      d104768a1214", BLERead | BLEWrite, 2);
24 BLEDevice central;
25 uint16_t rawValue = 0;
26 uint16_t currentValueDepth = 1;
27 double value = 0;
28 void setup() {
29   // Initialize I2C as a perihial
30   Wire.begin(SENSOR_ADDRESS);
31   //writeScalingValues(0, 100);
32   // Set the function to handle requests
33   Wire.onRequest(requestEvent);
34   // Start Serial Communication
35   Serial.begin(115200);
36   // while (!Serial)
37   //   ;  // Wait for Serial Monitor to be ready
38   Serial.println("Depth␣i2c␣ready␣to␣recive␣data");
```

```
39    // Initialize BLE
40    if (!BLE.begin()) {
41      Serial.println("Starting␣depth␣BLE␣failed!");
42      return;  // Exit setup if BLE initialization fails
43    }
44
45    Serial.println("Simulated␣Bar100␣Sensor␣started");
46    // Set up BLE peripheral
47    BLE.setLocalName("DepthNano33BLE");
48    BLE.setAdvertisedService(myService);
49    myService.addCharacteristic(myCharacteristic);
50    BLE.addService(myService);
51    // Start BLE advertising
52    BLE.advertise();
53    Serial.println("Depth␣BLE␣ready");
54    // Check if a central device is connected
55    while (!central) {
56      central = BLE.central();
57      Serial.println("Central␣not␣connected");
58      delay(100);
59    }
60    Serial.println("Central␣connected!");
61  }
62
63  void loop() {
64    // Keep checking if the central device is still connected
65    while (central.connected()) {
66      Serial.flush();
67      // If characteristic has been written, read the value
68      if (myCharacteristic.written()) {
69        uint8_t dataRecived[2];  // Buffer for 2 bytes
70        myCharacteristic.readValue(dataRecived, 2);
71        if (dataRecived[1] != currentValueDepth) {
72          rawValue = dataRecived[1] | (dataRecived[0] << 8);
73          value = (double)rawValue / 10;
74          currentValueDepth = dataRecived[1];
75
76          Serial.print("Updated␣value:␣");
77          Serial.print(value);
78          Serial.print("␣m");
79          Serial.println("␣");
80        }
81      }
82      if (!central.connected()) {
83        Serial.println("Connection␣lost");
84        delay(1000);
85      }
86    }
87  }
88
89  void requestEvent() {
90    if (!central.connected()) {
91      Serial.println("Simulation␣stopped");
92      delay(1000);
93    } else {
94      uint8_t requestID = Wire.read();  //gets request from libary
```

```
95      //Serial.println(requestID);
96      if (requestID == LD_CUST_ID0) {  //if initiated
97        //Serial.println("if sats 1");
98        uint8_t data[3];
99        data[0] = 0;
100       data[1] = cust_id0 << 8 & 0xFF;
101       data[2] = cust_id0 & 0xFF;
102       Wire.write(data, 3);
103     } else if (requestID == LD_CUST_ID1) {  //if initiated
104       //Serial.println("if sats 2");
105       uint8_t data[3];
106       data[0] = 0;
107       data[1] = cust_id1 >> 8 & 0xFF;
108       data[2] = cust_id1 & 0xFF;
109       Wire.write(data, 3);
110     } else if (requestID == LD_SCALING0) {  //mode
111       //Serial.println("if sats 3");
112       uint8_t data[3];
113       data[0] = 0;
114       data[1] = scaling0 << 8 & 0xFF;
115       data[2] = scaling0 & 0xFF;
116       Wire.write(data, 3);
117     } else if (requestID == LD_SCALING1) {  //MSP of Pmin
118       //Serial.println("if sats 4");
119       uint8_t data[3];
120       data[0] = 0;
121       data[1] = P_min >> 24 & 0xFF;
122       data[2] = P_min >> 16 & 0xFF;
123       Wire.write(data, 3);
124     } else if (requestID == LD_SCALING2) {  //LSP of Pmin
125       //Serial.println("if sats 5");
126       uint8_t data[3];
127       data[0] = 0;
128       data[1] = P_min >> 8 & 0xFF;
129       data[2] = P_min & 0xFF;
130       Wire.write(data, 3);
131     } else if (requestID == LD_SCALING3) {  //MSP of Pmax
132       //Serial.println("if sats 6");
133       uint8_t data[3];
134       data[0] = 0;
135       data[1] = P_max >> 24 & 0xFF;
136       data[2] = P_max >> 16 & 0xFF;
137       Wire.write(data, 3);
138     } else if (requestID == LD_SCALING4) {  //LSP of Pmax
139       //Serial.println("if sats 7");
140       uint8_t data[3];
141       data[0] = 0;
142       data[1] = P_max >> 8 & 0xFF;
143       data[2] = P_max & 0xFF;
144       Wire.write(data, 3);
145     } else {
146       //Serial.println("if sats 8");
147       uint8_t data[5];
148       float desiredC = 23.88;
149       float P_max = 100.00;
150       float P_min = 0.0;
```

```
151         float P_mode = 1.00;
152         float gravity = 9.80665;
153         float Pa = 100000;
154         float atmosPressure = 101325;
155         float waterDenisty = 997;
156 
157         desiredDepth = value;
158         uint16_t pressure = (uint16_t)(((((((desiredDepth *
               waterDenisty * gravity) + atmosPressure) / Pa) - P_min -
               P_mode) * 32768) / (P_max - P_min) + 16384));
159         uint16_t temperature = (uint16_t)((desiredC + 51.2) /
               0.003125);
160         // prepare to send calulated value
161         data[0] = 0;
162         data[1] = ((pressure >> 8) & 0xFF);
163         data[2] = (pressure & 0xFF);
164         data[3] = ((temperature >> 8) & 0xFF);
165         data[4] = (temperature & 0xFF);
166         // Send calulated values to controller
167         Wire.write(data, 5);
168         // Debugging
169         Serial.println("Data␣sent␣to␣controller:");
170         Serial.print("Temperature:␣");
171         Serial.println(temperature);
172         Serial.print("Pressure:␣");
173         Serial.println(pressure);
174         Serial.print("Desired␣depth␣");
175         Serial.println(desiredDepth);
176       }
177     }
178 }
```

## A.3 Simulated IMU sensor

**Listing A.3:** Main code Arduino NANO

```
1  #include <ArduinoBLE.h>
2  #include <Wire.h>
3 
4  #define ICM20948_ADDRESS 0x69  // Default I2C address
5  #define REG_0 0x00             // WHO_AM_I register
6  #define REG_0_ANSWER 0xEA
7  #define REG_6 0x6
8  #define REG_6_ANSWER_1 0x1
9  #define REG_6_ANSWER_2 0x41
10 #define REG_6_ANSWER_3 0x1
11 #define REG_F 0xF
12 #define REG_F_ANSWER_1 0x0
13 #define REG_F_ANSWER_2 0x0
14 #define REG_3D_7A_7D
15 #define REG_3D_7A_7D_ANSWER 0x0
16 #define REG_1 0x1
17 #define REG_1_ANSWER_1 0x0
```

```
18 #define REG_1_ANSWER_2 0x1
19 #define REG_1_ANSWER_3 0x1
20 #define REG_1_ANSWER_4 0x1
21 #define REG_1_ANSWER_5 0x39
22 #define REG_1_ANSWER_6 0x39
23 #define REG_1_ANSWER_7 0x38
24 #define REG_3 0x3
25 #define REG_3_ANSWER 0x0
26 #define REG_1_10_11_55
27 #define REG_1_10_11_55_ANSWER F1 + 0  //NACK
28 #define REG_69_1D_74_77
29 #define REG_69_1D_74_77_ANSWER 0x40
30 #define REG_17 0x17
31 #define REG_17_ANSWER_1 0x0
32 #define REG_17_ANSWER_2 0x40
33 #define REG_17_ANSWER_3 0x48
34 #define REG_17_ANSWER_4 0x48
35 #define REG_17_ANSWER_5 0x40
36 #define REG_17_ANSWER_6 0x9
37 #define REG_17_ANSWER_7 0x0
38 #define REG_17_ANSWER_8 0x40
39 #define REG_5 0x5
40 #define REG_5_ANSWER_1 0x40
41 #define REG_5_ANSWER_2 0x40
42 #define REG_14 0x14
43 #define REG_14_ANSWER_1 0x1
44 #define REG_14_ANSWER_2 0x1
45 #define REG_14_ANSWER_3 0x1
46 #define REG_14_ANSWER_4 0x39
47 #define REG_14_ANSWER_5 0x39
48 #define REG_14_ANSWER_6 0x38
49 // Define service and characteristic
50 BLEService myService("18b10000-e8f2-537e-4f6c-d104768a1214");
51 BLECharacteristic myCharacteristic("18b10001-e8f2-537e-4f6c-
      d104768a1214", BLERead | BLEWrite, 6);
52 BLEDevice central;
53 uint16_t rawPitchValue = 0;
54 uint16_t rawRollValue = 0;
55 uint16_t rawYawValue = 0;
56 uint16_t currentValuePitch = 1;
57 uint16_t currentValueRoll = 1;
58 uint16_t currentValueYaw = 1;
59
60 uint8_t reg_17_array[8] = { REG_17_ANSWER_1, REG_17_ANSWER_2,
      REG_17_ANSWER_3, REG_17_ANSWER_4, REG_17_ANSWER_5,
      REG_17_ANSWER_6, REG_17_ANSWER_7, REG_17_ANSWER_8 };
61 int z = 0;
62 uint8_t reg_14_array[6] = { REG_14_ANSWER_1, REG_14_ANSWER_2,
      REG_14_ANSWER_3, REG_14_ANSWER_4, REG_14_ANSWER_5,
      REG_14_ANSWER_6 };
63 int y = 0;
64 uint8_t reg_5_array[2] = { REG_5_ANSWER_1, REG_5_ANSWER_2 };
65 int x = 0;
66 uint8_t reg_1_array[7] = { REG_1_ANSWER_1, REG_1_ANSWER_2,
      REG_1_ANSWER_3, REG_1_ANSWER_4, REG_1_ANSWER_5, REG_1_ANSWER_6,
      REG_1_ANSWER_7 };
```

```
67  int k = 0;
68  uint8_t reg_6_array[3] = { REG_6_ANSWER_1, REG_6_ANSWER_2,
        REG_6_ANSWER_3 };
69  int i = 0;
70  uint8_t reg_f_array[2] = { REG_F_ANSWER_1, REG_F_ANSWER_2 };
71  int p = 0;
72
73  uint8_t data[23];
74  uint8_t pitchData[2];
75  uint8_t rollData[2];
76  uint8_t yawData[2];
77
78  void setup() {
79    Wire.begin(ICM20948_ADDRESS);
80    Wire.onRequest(requestEvent);
81    Serial.begin(115200);
82    // while (!Serial)
83    //   ;  // Wait for Serial Monitor to be ready
84    Serial.println("IMU i2c ready to recive data");
85    // Initialize BLE
86    if (!BLE.begin()) {
87      Serial.println("Starting IMU BLE failed!");
88      return;  // Exit setup if BLE initialization fails
89    }
90    // Set up BLE peripheral
91    BLE.setLocalName("IMUNano33BLE");
92    BLE.setAdvertisedService(myService);
93    myService.addCharacteristic(myCharacteristic);
94    BLE.addService(myService);
95    // Start BLE advertising
96    BLE.advertise();
97    Serial.println("IMU BLE ready");
98    while (!central) {
99      central = BLE.central();
100     Serial.println("Central not connected");
101     delay(100);
102   }
103   Serial.println("Central connected!");
104   // Initialise peripheral adress
105 }
106
107 void loop() {
108   while (central.connected()) {
109     Serial.flush();
110     // If characteristic has been written, read the value
111     if (myCharacteristic.written()) {
112       uint8_t dataRecived[6];
113       myCharacteristic.readValue(dataRecived, 6);
114       // Checks if values have changed
115       if ((dataRecived[1] != currentValueRoll) || (dataRecived[3]
              != currentValuePitch) || (dataRecived[5] !=
              currentValueYaw)) {
116         //Saves new values accordingly
117         rollData[0] = dataRecived[0];
118         rollData[1] = dataRecived[1];
119         pitchData[0] = dataRecived[2];
```

```
120         pitchData[1] = dataRecived[3];
121         yawData[0] = dataRecived[4];
122         yawData[1] = dataRecived[5];
123
124         //Prepares to send new values to the glider
125         data[6] = rollData[0];
126         data[7] = rollData[1];
127         data[8] = pitchData[0];
128         data[9] = pitchData[1];
129         data[10] = yawData[0];
130         data[11] = yawData[1];
131         //Updates current to be able to look for changes
132         currentValueRoll = dataRecived[1];
133         currentValuePitch = dataRecived[3];
134         currentValueYaw = dataRecived[5];
135
136         //Adds the two bytes to be able to print out the decimal
137         rawRollValue = rollData[1] | (rollData[0] << 8);
138         rawPitchValue = pitchData[1] | (pitchData[0] << 8);
139         rawYawValue = yawData[1] | (yawData[0] << 8);
140
141         //Decodes into decimal numbers and prints them
142         Serial.print("Received␣values:␣");
143         if (rawRollValue >= 0x8000) {
144           double rollValue = (double)(rawRollValue - 65536) / 131;
145           Serial.print("Roll=");
146           Serial.print(rollValue);
147           Serial.print("␣");
148         } else {
149           double rollValue = (double)rawRollValue / 131;
150           Serial.print("Roll=");
151           Serial.print(rollValue);
152           Serial.print("␣");
153         }
154         if (rawPitchValue >= 0x8000) {
155           double pitchValue = (double)(rawPitchValue - 65536) /
                131;
156           Serial.print("Pitch=");
157           Serial.print(pitchValue);
158           Serial.print("␣");
159         } else {
160           double pitchValue = (double)rawPitchValue / 131;
161           Serial.print("Pitch=");
162           Serial.print(pitchValue);
163           Serial.print("␣");
164         }
165         if (rawYawValue >= 0x8000) {
166           double yawValue = (double)(rawYawValue - 65536) / 131;
167           Serial.print("Yaw=");
168           Serial.print(yawValue);
169         } else {
170           double yawValue = (double)rawYawValue / 131;
171           Serial.print("Yaw=");
172           Serial.print(yawValue);
173         }
174         Serial.println("␣");
```

```
175       }
176     }
177     if (!central.connected()) {
178       Serial.println("Connection lost");
179       delay(1000);
180     }
181   }
182 }
183 // Handles controllers data request
184 void requestEvent() {
185   if (!central.connected()) {
186     Serial.println("Simulation stopped");
187     delay(1000);
188   } else {
189     uint8_t requestID = Wire.read();
190     // Serial.print("Address: ");
191     // Serial.print(requestID, HEX);
192     // Serial.println(" ");
193     if (requestID == REG_0) {
194       Wire.write(REG_0_ANSWER);  // Return the device ID for ICM
          -20948
195       // Serial.print("Printed ");
196       // Serial.print(REG_0_ANSWER);
197       // Serial.println(" ");
198     } else if (requestID == REG_6) {
199       Wire.write(reg_6_array[i]);  // Return the device ID for ICM
          -20948
200       // Serial.print("Printed ");
201       // Serial.print(reg_6_array[i], HEX);
202       // Serial.println(" ");
203       i++;
204     } else if (requestID == REG_F) {
205       Wire.write(reg_f_array[p]);  // Return the device ID for ICM
          -20948
206       // Serial.print("Printed ");
207       // Serial.print(reg_f_array[p], HEX);
208       // Serial.println(" ");
209       p++;
210     } else if (requestID == 0x3D) {
211       Wire.write(0x0);  // Return the device ID for ICM-20948
212       // Serial.print("Printed 0x3D");
213       // Serial.print("Adress: ");
214       // Serial.println(" ");
215     } else if (requestID == 0x7A) {
216       Wire.write(0x0);  // Return the device ID for ICM-20948
217       // Serial.print("Printed 0x7A");
218       // Serial.print("Adress: ");
219       // Serial.println(" ");
220     } else if (requestID == 0x7D) {
221       Wire.write(0x0);  // Return the device ID for ICM-20948
222       // Serial.print("Printed 0x7D");
223       // Serial.print("Adress: ");
224       // Serial.println(" ");
225     } else if (requestID == REG_1) {
226       Wire.write(reg_1_array[k]);  // Return the device ID for ICM
          -20948
```

```
227         // Serial.print("Printed ");
228         // Serial.print(reg_1_array[k], HEX);
229         // Serial.println(" ");
230         k++;
231       } else if (requestID == REG_3) {
232         Wire.write(REG_3_ANSWER);  // Return the device ID for ICM
                -20948
233         // Serial.print("Printed ");
234         // Serial.print(REG_3_ANSWER);
235         // Serial.println(" ");
236       } else if (requestID == 0x69) {
237         Wire.write(0x40);  // Return the device ID for ICM-20948
238         // Serial.print("Printed 0x69");
239         // Serial.print("Adress: ");
240         // Serial.println(" ");
241       } else if (requestID == 0x1D) {
242         // Wire.write(0x40);  // Return the device ID for ICM-20948
243         // Serial.print("Printed 0x1D");
244         // Serial.print("Adress: ");
245         // Serial.println(" ");
246       } else if (requestID == 0x74) {
247         Wire.write(0x40);  // Return the device ID for ICM-20948
248         // Serial.print("Printed 0x74");
249         // Serial.print("Adress: ");
250         // Serial.println(" ");
251       } else if (requestID == 0x77) {
252         Wire.write(0x40);  // Return the device ID for ICM-20948
253         // Serial.print("Printed 0x77");
254         // Serial.print("Adress: ");
255         // Serial.println(" ");
256       } else if (requestID == REG_17) {
257         Wire.write(reg_17_array[z]);
258         // Serial.print("Printed ");
259         // Serial.print(reg_17_array[z], HEX);
260         // Serial.println(" ");  // Return the device ID for ICM
                -20948
261         z++;
262       } else if (requestID == REG_5) {
263         Wire.write(reg_5_array[x]);
264         // Return the device ID for ICM-20948
265         // Serial.print("Printed ");
266         // Serial.print(reg_5_array[x], HEX);
267         // Serial.println(" ");
268         x++;
269       } else if (requestID == REG_14) {
270         Wire.write(reg_14_array[y]);
271         // Serial.print("Printed ");
272         // Serial.print(reg_14_array[y], HEX);
273         // Serial.println(" ");  // Return the device ID for ICM
                -20948
274         y++;
275       } else if (requestID == 0x1A) {
276         Wire.write(0x1);
277         // Serial.print("Printed ");
278         // Serial.print(0x1);
279         // Serial.println(" ");  // Return the device ID for ICM-2094
```

```
280     } else if (requestID == 0x2D) {
281       data[0] = 0x40;  //acc x h, if we want 1 g: x_acc_hexa =
              1000*desired_x_acc*16.384 = 16384 = 0x4000
282       data[1] = 0x00;  //acc x l
283       data[2] = 0x00;  //acc y h, same equation as acc_x
284       data[3] = 0x21;  //acc y l
285       data[4] = 0x00;  //acc z h, same equation as acc_x
286       data[5] = 0x22;  //acc z l
287
288       // data[6] = 0x00;   //gyr x h, if we want 1 DPS(degree per
              secound): x_gyro_hex = desired_X_angular_rate*131 = 131 =
              0x0083
289       // data[7] = 0x83;   //gyr x l
290       // data[8] = 0x00;   //gyr y h, same equation as gyr_x
291       // data[9] = 0x29;   //gyr y l
292       // data[10] = 0x00;  //gyr z h, same equation as gyr_x
293       // data[11] = 0x29;  //gyr z l
294
295       data[12] = 0x18;  //temp h, if we want 40 C: tempHEX = ((
              desiredTemp-21)*333.87) +21 = 6364 = 0x18DC
296       data[13] = 0xDC;  //temp l
297       data[14] = 0x01;  //magStat1, Data is ready
298
299       //mag is little endian
300       data[15] = 0x6D;  //mag x l, if we want 22uT: x_mag_hex =(-1)
              *desiredMag (uT)/0.15 = -147 = 0xFF6D
301       data[16] = 0xFF;  //mag x h
302       data[17] = 0x00;  //mag y l, same equation as mag_x
303       data[18] = 0x93;  //mag y h
304       data[19] = 0x00;  //mag z l, same equation as mag_x
305       data[20] = 0xFF;  //mag z h
306
307       data[22] = 0x00;  //magStat2, No overflow
308
309       Wire.write(data, 23);
310       //Serial.println("Data sent to controller");
311     } else {
312       Wire.write(0x0);
313       // Serial.println(" ");
314     }
315   }
316 }
```

## A.4 Simulation code

**Listing A.4:** Main code Octave

```
1 clear;
2 clc;
3 close all;
4 pkg load instrument-control
5 % Serial setup
6 serial_port="/dev/ttyACM0";
```

```
topic="octave/data";
baudRate = 115200;
s=serial(serial_port, baudRate);
fopen(s);

% Rectangle dimensions (M)
Ly = 0.48;  % Y length
Lx = 0.5; % X length
% Create figure
mainPlot=figure;

% Subplot 1: CoG visualization
subplot(2,3,1); hold on;
axis([-0.35, 0.35, -0.03, 0.03]);
grid on;
title('CoG Over Time');
xlabel('X Position (CM)');
ylabel('Y Position (CM)');
scatter(0, 0, 100, 'r', 'filled');
cog_point = scatter(0, 0, 100, 'b', 'filled');

% Subplot 2: Buoyancy visualization
subplot(2,3,2); hold on;
title('Buoyancy Over Time');
xlabel('Time (s)');
ylabel('Buoyancy');
grid on;
buoyancy_line = plot(nan, nan, 'r-', 'LineWidth', 2);

%Subplot empty
subplot(2,3,3); hold on;

%Subplot 4: Depth
subplot(2,3,4); hold on;
title('Depth Over Time');
xlabel('Time (s)');
ylabel('Depth');
grid on;
depth_line = plot(nan, nan, 'b-', 'LineWidth', 2);

%Subplot 5: Distance
subplot(2,3,5); hold on;
title('Distance Over Time');
xlabel('Time (s)');
ylabel('Distance');
grid on;
distance_line = plot(nan, nan, 'g-', 'LineWidth', 2);

% Subplot 6: Orientation
subplot(2,3,6); hold on;
title('Orientation Over Time');
xlabel('Time (s)');
ylabel('Angle (deg)');
grid on;
roll_line = plot(nan, nan, 'b-', 'LineWidth', 2, 'DisplayName', '
    Roll');
```

```
62 pitch_line = plot(nan, nan, 'g-', 'LineWidth', 2, 'DisplayName', '
       Pitch');
63 yaw_line = plot(nan, nan, 'm-', 'LineWidth', 2, 'DisplayName', 'Yaw
       ');
64 legend;
65
66 % Time tracking & Data storage
67 data_log = []; % Store simulation data
68 time_data = [];
69 bouyancy_data = [];
70 depth_data = [];
71 distance_data = [];
72 roll_data = [];
73 pitch_data = [];
74 yaw_data = [];
75 t0 = tic; % Start timer
76
77 depth = 0;
78 traveled=0;
79 lastT=0;
80 gravity=9.82;
81 vx=0;
82 vy=0;
83 newvx=0;
84 newvy=0;
85 prevx=0;
86 prevy=0;
87 newax=0;
88 neway=0;
89 preax=0;
90 preay=0;
91 deltaax=0;
92 deltaay=0;
93 newdepth=0;
94 newtraveled=0;
95 predepth=0;
96 pretraveled=0;
97 ay=0;
98 ax=0;
99 yaw=0;
100 F_total=0;
101 waterDens=997;
102 mass=17.9;
103 volCur=0.017953862;
104 volAcc=volCur+0.0001;
105 volDec=volCur-0.0001;
106 depthReached=false;
107 targetdepth=-2; %negative number
108 Fx=0;
109 Fy=0;
110 step=0.1;
111
112
113 % Simulation loop
114 while true
115   flushoutput(s);
```

```
116     % Read serial data
117   if ishandle(mainPlot)
118     raw_data = char(srl_read(s, 16));
119     %disp(raw_data);
120     forces = str2num(raw_data); % [loadSensor1, loadSensor2,
          loadSensor3, loadSensor4, bladderSensor]
121
122     %disp(forces);
123     if length(forces) ~= 4
124       continue; % Skip invalid data
125     end
126
127
128     % Assign forces
129     %S2    y    S1
130     %      ^
131     %      |
132     %      |
133     %      |
134     %----------> x
135     %      |
136     %      |
137     %S4    |     S3
138     loadSensor1 = forces(1); % Sensor top left
139     loadSensor2 = forces(2); % Sensor bottom left
140     loadSensor3 = forces(3); % Sensor top right
141     loadSensor4 = forces(4); % Sensor bottom right
142
143     Fx = (1/1000)*(-loadSensor1-loadSensor3);%*gravity
144     Fy = (1/1000)*(loadSensor1+loadSensor2);%*gravity;
145
146
147     % Calulate roll, pitch and yaw to send
148     roll=atand(2*Fy/Ly);
149     pitch=atand(2*Fx/Lx);
150
151     t = toc(t0);
152     dt=t-lastT;
153     lastT=t;
154     if dt>0.08
155       dt=0.07;
156     endif
157
158     if depth>=0
159       volCur=0.0179;
160     endif
161
162     if depth <= targetdepth
163       depthReached=true;
164       volCur=volCur+(volCur*0.00005);
165       if volCur >= volAcc
166         volCur=volAcc;
167       endif
168     elseif depthReached==false
169       volCur=volCur-(volCur*0.00005);
170       if volCur<=volDec
```

```
171         volCur=volDec;
172       endif
173     end
174     Fgravit=mass*gravity;
175     Fbouy=waterDens*volCur*gravity;
176
177     Fnet=Fbouy-Fgravit;
178     bouAcc=Fnet/mass;
179
180     %we want ay to be positive when accending and negative when
            diving, even if the pitch changes
181     if depthReached==false
182       ay=(bouAcc*sind(pitch))+bouAcc
183     elseif depthReached==true
184       ay=(-bouAcc*sind(pitch))+bouAcc
185     end
186     ax=bouAcc*cosd(pitch);
187     trueax=abs(ax);
188
189     newvx=newvx+(trueax*dt);
190     newvy=newvy+(ay*dt);
191     vx=newvx-prevx;
192     vy=newvy-prevy
193     prevx=vx;
194     prevy=vy;
195     traveled=traveled+(vx*dt);
196     depth=depth+(vy*dt);
197
198     % Store data in a matrix
199     data_log = [data_log; t, roll, pitch, yaw, volCur, depth,
            traveled];
200     %Send data to Arduino Nanos
201     command=sprintf("mosquitto_pub -h localhost -t %s -m [%f,%f,%f
            ,%f]", topic, -depth*10, -pitch*131, roll*131, yaw*131);
202     system(command);
203
204     % Store for plotting
205     time_data = [time_data, t];
206     bouyancy_data = [bouyancy_data, volCur];
207     depth_data = [depth_data, depth];
208     distance_data = [distance_data, traveled];
209     roll_data = [roll_data, roll];
210     pitch_data = [pitch_data, pitch];
211     yaw_data = [yaw_data, yaw];
212
213     % Limit time window to last 1000 points
214     if length(time_data) > 1500
215         time_data(1) = [];
216         bouyancy_data(1) = [];
217         depth_data(1) = [];
218         distance_data(1) = [];
219         roll_data(1) = [];
220         pitch_data(1) = [];
221         yaw_data(1) = [];
222     end
223
```

```
224     if ishandle(cog_point) && ishandle(buoyancy_line) && ishandle(
            depth_line) && ...
225       ishandle(roll_line) && ishandle(pitch_line) && ishandle(
              yaw_line) && ishandle(distance_line)
226     % Update plots
227       set(cog_point, 'XData', Fx, 'YData', -Fy);
228       set(buoyancy_line, 'XData', time_data, 'YData', bouyancy_data
              );
229       set(depth_line, 'XData', time_data, 'YData', depth_data);
230       set(distance_line, 'XData', time_data, 'YData', distance_data
              );
231       set(roll_line, 'XData', time_data, 'YData', roll_data);
232       set(pitch_line, 'XData', time_data, 'YData', pitch_data);
233     end
234     drawnow;
235
236   else
237     close all;
238     break;
239   end
240 end
241 % Display collected data as table with units
242 headers={'Time␣(s)','Roll␣(  )', 'Pitch␣(  )','Yaw␣(  )','Buoyancy␣
      (N)','Depth␣(M)','Distance␣(M)'};
243 data_table=[headers; num2cell(data_log)];
244 %disp(data_table);
245
246 fid=fopen('simulation_results.txt', 'w');
247 fprintf(fid,'%s\t%s\t%s\t%s\t%s\t%s\t%s\t\n', headers{1}, headers
      {2}, headers{3}, headers{4}, headers{5}, headers{6}, headers{7})
      ;
248
249   for i=2:size(data_table,1)
250     fprintf(fid,'%d\t%d\t%d\t%d\t%d\t%d\t%d\t\n', data_table{i,1},
          data_table{i,2}, data_table{i,3}, data_table{i,4},
          data_table{i,5}, data_table{i,6}, data_table{i,7});
251   end
252
253 disp("Data saved to 'simulation_results.txt'");
254 fclose(fid);
255 fclose(s);
```

## A.5 Raspberry PI code

**Listing A.5:** Main code Raspberry PI

```
1 from bluepy.btle import Peripheral, UUID, BTLEException
2 import struct
3 import paho.mqtt.client as mqtt
4 import json
5
6 status=True
7
```

```
#Arduino Nano's address
depth_nano_address = "0D:FC:3C:41:F0:DC"  # Find this using `bluetoothctl scan on`
depth_service_uuid = UUID("19b10010-e8f2-537e-4f6c-d104768a1214")
depth_char_uuid = UUID("19b10011-e8f2-537e-4f6c-d104768a1214")

IMU_nano_address = "21:A4:43:4A:E5:3C"  # Find this using `bluetoothctl scan on`
IMU_service_uuid = UUID("18b10000-e8f2-537e-4f6c-d104768a1214")
IMU_char_uuid = UUID("18b10001-e8f2-537e-4f6c-d104768a1214")

depth=0 # max one decimal, ex depth=1050.8m, max number 65535, max depth around 1510.0m (15100)
roll=0 #max number 500 deg
pitch=0 #max number 500 deg
yaw=0 #max number 500 deg

try:
    print("Connecting to Depth_Nano 33 BLE...")
    depth_device = Peripheral(depth_nano_address, addrType="public")
    print("Depth Connected ")

    print("Connecting to IMU_Nano 33 BLE...")
    IMU_device = Peripheral(IMU_nano_address, addrType="public")
    print("IMU Connected")

    # Get the service and characteristic
    depth_service = depth_device.getServiceByUUID(depth_service_uuid)
    depth_characteristic = depth_service.getCharacteristics(depth_char_uuid)[0]

    IMU_service = IMU_device.getServiceByUUID(IMU_service_uuid)
    IMU_characteristic = IMU_service.getCharacteristics(IMU_char_uuid)[0]


    if "WRITE" not in depth_characteristic.propertiesToString():
##        print("Depth Characteristic is not writable. Check BLE adresses and Arduino setup.")
        status=False
    else:
        print("Depth Characteristic is writable.")

    if "WRITE" not in IMU_characteristic.propertiesToString():
##        print("IMU Characteristic is not writable. Check BLE adresses  and Arduino setup.")
        status=False
    else:
##        print("IMU Characteristic is writable.")


        depth_char_handle = depth_characteristic.getHandle()
##        print(f"Depth Characteristic Handle: {depth_char_handle}")
```

```
54
55          IMU_char_handle = IMU_characteristic.getHandle()
56  ##          print(f"IMU Characteristic Handle: {IMU_char_handle}")
57
58      while status:
59          # Write data
60
61          def on_message(client, userdata, message):
62              global depth, roll, pitch, yaw
63
64              array_str = message.payload.decode()
65              #print("here")
66              try:
67              #print("her1e")
68              # Convert the received JSON string into a Python list
69                  new_values = json.loads(array_str)
70
71              # Update the variables with the latest values from the
                    array
72              # Here we assume that the array always has 6 elements
73                  if len(new_values) >= 1:
74                      depth = new_values[0]
75                  if len(new_values) >= 2:
76                      roll = new_values[1]
77                  if len(new_values) >= 3:
78                      pitch = new_values[2]
79                  if len(new_values) >= 4:
80                      yaw = new_values[3]
81
82                  updatedDepth=int(depth) & 0xFFFF
83                  updatedRoll=int(roll) & 0xFFFF
84                  updatedPitch=int(pitch) & 0xFFFF
85                  updatedYaw=int(yaw) & 0xFFFF
86
87                  data_set=[updatedDepth, updatedRoll, updatedPitch,
                        updatedYaw]
88                  depth_data = struct.pack(">H", data_set[0])
89                  orientation_data= struct.pack(">HHH", *data_set
                        [1:4])
90                  depth_device.writeCharacteristic(depth_char_handle,
                         depth_data, withResponse=True)
91                  IMU_device.writeCharacteristic(IMU_char_handle,
                        orientation_data, withResponse=True)
92                  # Print the updated values for each variable
93                  print(f"Updated␣variables:␣Depth␣=␣{updatedDepth},␣
                        Roll␣=␣{updatedRoll},␣Pitch␣=␣{updatedPitch},␣"
94                        f"Yaw␣=␣{updatedYaw}")
95
96              except json.JSONDecodeError as e:
97                  print(f"Error␣decoding␣JSON:␣{e}")
98                  print("Received␣invalid␣JSON:", message.payload.
                        decode())
99
100         # MQTT settings
101         broker = "localhost"
102         topic = "octave/data"
```

```
103
104         # Create the MQTT client
105         client = mqtt.Client(callback_api_version=mqtt.
                CallbackAPIVersion.VERSION2)
106         # Set the callback for handling messages
107         client.on_message = on_message
108         # Connect to the broker and subscribe to the topic
109         client.connect(broker)
110         client.subscribe(topic)
111         # Start listening for messages
112         print("Listening")
113         client.loop_forever()
114 ##        print("Write operation completed.")
115         status=False
116
117     if not status:
118             #Disconnect devices
119             depth_device.disconnect()
120             print("Depth Disconnected.")
121             IMU_device.disconnect()
122             print("IMU Disconnected.")
123             client.disconnect()
124
125 except BTLEException as e:
126     print("Bluetooth error:", e)
```

**DEPARTMENT OF MECHANICS AND MARITIME SCIENCES**
**CHALMERS UNIVERSITY OF TECHNOLOGY**
Gothenburg, Sweden 2026
www.chalmers.se


CHALMERS
UNIVERSITY OF TECHNOLOGY