Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Datateknologi	Spring Semester 2023 Åpen
Forfatter(e): Chiran Pokhrel, Filip Sølvberg Herrera, Daniel Edvardsen	
Fagansvarlig: Erlend Tøssebro Veileder: Erlend Tøssebro	
Tittel på bacheloroppgaven: Bildebehandling og autonomi Engelsk tittel: Image processing and autonomy	
Studiepoeng: 3*20	
Emneord: image processing, autonomi, python,	Sidetall: 109 vedlegg/annet: 1 Stavanger 15. mai 2023

1 Abstract

Abstract. This bachelor's thesis is about creating and implementing a software program on an underwater robot (ROV), that utilizes image-processing and computer vision to perform autonomous tasks. This is done as part of an interdisciplinary project in the student organization UiS Subsea. The ROV is built with the purpose of competing in the MATE ROV World Championship, which is held in Colorado, USA, on the 20-24th of June 2023.

This thesis describes how camera feeds are received on the topside system, from the ROV, how image processing is utilized to solve autonomous tasks, the implementation of the program into a graphical user interface (GUI), and the sending of driving commands down to the ROV. Additionally, an attempt was made at using computer vision to create a 3D model of a coral head.

A modular program has been made, receiving multiple camera feeds, performing autonomous tasks, and sending driving commands. This program was implemented into a GUI, and tested on land. Clearly defined image-processing tasks were given by MATE. All of them were solved except for 3D modeling. The program was tested on land and behaved as intended. However, the programs were not tested in water. This was due to the ROV not being finished in time for our group to test our programs.

Testing and improving the program will continue in preparation for MATE. Due to the submission date for the bachelor's thesis, this will not be included in this paper.

See our GitHub repository for the code:
https://github.com/UiS-Subsea/Bachelor_Bildebehandling
See the GUI group's repository for our code implemented into their code:
https://github.com/UiS-Subsea/Bachelor_GUI

2 Acknowledgement

This project would not have been possible without the help of others, in this paragraph, we would like to express our gratitude towards the many people who have provided assistance during the development. Firstly We would like to thank our supervisor Erlend Tøssebro for guiding us and giving continuous feedback throughout the process of this thesis. Further, we would like to thank UiS Subsea, for the opportunity to partake in this year's project. Particularly we would like to thank this year's board members: Thomas R. Choat, Otto N. Ljosdal, and Vebjørn L. Riiser. Especially Vebjørn L. Riiser who provided invaluable assistance in the development and integration of the main program. Finally, we would also like to thank the subsea groups we collaborated closely with, GUI, Communications, and Sensor.

Contents

3 Keywords

This section lists all important keywords and expressions used throughout the document

- **ROV** - Remotely Operated (Underwater) Vehicle
- **AUV** - Autonomous Underwater Vehicle
- **MATE** - Marine Advanced Technology Education. In this context, it is a reference to the MATE ROV competition in the USA.
- **FPS** - Frames per second
- **RGB** - A color model that can produce all colors using a combination of red, green, and blue.
- **BGR** - Same as RGB, only with the colors in a different order. Used by frameworks like OpenCV
- **DFE** - Design for environment
- **FEM** - Finite element method analysis
- **CAN** - Controller Area Network
- **TDD** - Test Driven Development
- **DDD** - Domain Driven Design
- **FOV** - Field of view
- **GUI** - Graphical User Interface. An interface/window the user can interact with.
- **IMU** - Inertial Measurement Unit. A gyroscopic sensor to measure orientation.

4 Introduction

This section presents UiS Subsea as an organization and introduces the bachelor groups working on this year's project. Additionally, it provides an overview of the MATE ROV competition, which is the foundation of this year's project. Lastly, it will put forward the specific tasks this bachelor's thesis tried to solve. This is a joint introduction with the rest of the UiS Subsea members writing their thesis.

4.1 About UiS SubSea

UiS Subsea is a student organization at the University of Stavanger, engaging students in underwater technology since 2013. The primary goal of this organization is to provide students with the experience of working in a team consisting of different engineering disciplines.

This year, at UiS Subsea, there are a total of nine bachelor groups working together with the common goal of designing and building a complete ROV. This year's ambitions are to build a new and improved ROV based on last year's project. Doing so makes the ROV easier to maintain, and more efficient, while also upgrading the software to function similarly to an AUV. Though it will not be completely autonomous, it should be able to perform autonomous tasks.

Two groups consist of mechanical engineering students, responsible for designing, crafting, and constructing the chassis, a manipulator, and the electronic enclosure, for the ROV. Five groups of electronic engineers are responsible for sensors, control systems, connections, circuits, and the float. They are also responsible for the communication between the ROV and the topside system. Topside there are two computer science groups responsible for sending and receiving commands and data, displaying information and camera output in a GUI, handling controls, and utilizing image processing to perform autonomous tasks.

Through several years, Subsea has built ROVs and participated in international competitions. This year Subsea will be partaking in the MATE ROV competition. These competitions provide a basis for more advanced problem-solving and teamwork, with the purpose of creating a positive and healthy environment for learning and developing technical skills.

Figure 1: UiS Subsea logo

UiS Subsea also opens up the opportunity for students to collaborate with industrial companies. Several companies are greatly interested in these projects, providing components and other resources through sponsorship deals. To further improve the relations between the organizations and the industry, Subsea annually holds an event called 'Subsea Day', where companies from the industry can have their own stands in a venue and promote themselves. Both UiS Subsea and the companies generate a lot of exposure from this event.

In previous years the organization has suffered from a lack of continuity. This is due to students who participated in the previous project leaving. Most of the practical experience gained by the students from the Subsea project is not transferred to the next year's project. This year, the previous leader and second in command of UiS Subsea decided to stay one more year to help guide the students and the overall course of the project. This reduces the learning curve for the new students, while also providing a stream of previous knowledge and experiences to the coming project.

A project board was created which consists of the students participating in the project this year.

This year's board consists of the following roles and leaders:

- **Project manager:** Joar Rodrigues de Miranda
- **Second Project manager:** Thomas Matre
- **Technical leader Electro:** Jesper A. Flatheim
- **Technical leader Computer:** Filip Sølvberg Herrera
- **Technical leader Machine:** Haakon Aleksander Schei

4.2 Mechanical Groups

4.2.1 Design of ROV frame, electronic enclosure, and float

The objective of this group is to design and build the frame of the ROV, as well as the electronic housing and the shell of the float, following the product development process. The primary focus of the design is to mesh together all the individual parts into one functioning ROV, while also ensuring it follows MATE's physical restrictions regarding weight and volume, while still being able to perform the tasks in the MATE ROV competition.

Due to the present-day environmental challenges, a secondary focus will be put on sustainability, usage of recyclable materials, and Design for Environment (DFE) in the development process. Effective use of DFE can also help reduce cost and production time while increasing product quality. Doing so requires structural analysis, flow analysis, FEM analysis, buoyancy analysis, and material choice analysis.

4.2.2 Manipulator

The main task is to develop and design a functional manipulator for the ROV to perform tasks. The goal is for the mechanical arm to be functional for completing tasks given by MATE while being simple enough for production and maintenance. Problems that need to be solved here are deciding the degrees of freedom the manipulator needs, which material that is best to use, and which mechanical principles to implement. For a solid end product, it is important to complete stress, bending, and shear analysis. In addition, cooperation with the other mechanical group, and the electrical engineering groups is essential for manipulator compatibility with the rest of the ROV.

4.3 Electronic Groups

4.3.1 Power module

The power module groups' task is to regulate and disperse input voltage supplied from the topside system, whilst also ensuring a fail-safe against overloading components and preventing short circuits. With an input voltage of 48V, it is essential that the system regulates the voltage down to such a degree that different components are supplied with the correct amount.

4.3.2 Communication

The communication group has the task of creating a common system so that electrical circuit boards are connected together and are able to communicate with the rest of the system. In addition to this, they are also responsible for communication between the ROV and the topside system. This entails control of signals from topside to ROV, processing data and video feed from ROV to topside which needs to be processed efficiently with minimal delay. Additionally, the internal design and layout of the electronic enclosure are also part of this group's responsibilities.

4.3.3 Regulation system

The task is to create a system for navigation and regulation of the ROV. The most central parts of the system are the thruster configuration, the manipulator motor, and the circuit board. These are essential for the physical limitations and attributes of the ROV and how it interacts with the environment.

Another central part of this group's responsibility is developing a control system. This should interpret commands and sensor measurements from the topside and other circuit boards so that the ROV is able to operate and maneuver. In addition, regulation needs to be able to maintain the stability, orientation, and depth of the ROV.

4.3.4 Sensor system

The main task of the sensor system is to maintain and disperse information from the different sensors and act upon the vital data. These are orientation(IMU), leakage, pressure sensors, and temperature sensors. The IMU should retrieve angle data and axis relations, the leakage sensors should be placed inside of the electronic enclosure to detect eventual leaks and aptly react, the pressure sensor should be mounted externally, and the temperature sensors should be mounted in three key locations inside the enclosure.

4.3.5 Float

The float is the only component that is not attached to the main ROV. It is essentially its own Autonomous Underwater Vehicle (AUV), with a predefined flight path and its own power supply. It is used to gather vital information about ocean health and the underwater environment.

The competition requires that the float completes two vertical profiles: Sink to the bottom and return to the surface. Afterward, it has to relay the time of completion with a ping to the topside system, in addition to temperature and pressure data. This will be displayed for the operator to view.

4.4 Computer Groups

4.4.1 Image Processing (This bachelor's thesis):

The image processing group is responsible for solving specific tasks given by MATE which will be touched upon later. These tasks require image processing to measure depths, count objects, and perform autonomous tasks. These are the tasks:

- **Autonomous Docking**. Drive the ROV in front of a docking station, and autonomously maneuver the ROV into the docking station.
- **Count frogs along a transect**. The ROV should autonomously follow a transect and count the number of frogs within it.
- **Monitor and analyze the growth/decay of seaweed**. In this task, the ROV should analyze and compare two seaweed fields on the ocean floor, and determine the growth/decay.
- **3D Modelling of sick coral**: The ROV should create a 3D model of a sick coral, as well as determine its size.

4.4.2 GUI:

The primary objective is to develop a monitoring and control system for the ROV (Remotely Operated Vehicle). To achieve this, a system needs to be implemented that enables the transmission of commands and control data from the topside to the ROV. Furthermore, it should present all relevant information and video feeds to the user in real time through a custom graphical user interface (GUI).

4.5 MATE - Marine Advanced Technology Education

The information beneath is retrieved from the websites of the MATE organization [21]

The ROV created by UiS Subsea this year follows the specification determined by the international competition, *MATE ROV COMPETITION*. This is a competition hosted by the organization *MATE (The Marine*

Advanced Technology Education). They have a partnership called MATE Center, which consists of a multitude of American organizations, established in 1997. These partners are mainly comprised of schools, research institutes, governments, and Marine institutes. The main goal of this cooperation is to improve marine technical education, which in turn strengthens the future of maritime operations.

Figure 2: MATE logo

In 2021, *MATE* transferred the responsibility for student activity over to *Marine Advanced Technology Education for Inspiration and Innovations*, otherwise known as *MATE II*. Their main objective is to motivate students' interest in maritime knowledge, mainly by hosting the *MATE* competition every year. Here they challenge students to implement engineering principles and knowledge to solve subsea tasks. UiS Subsea is competing in the EXPLORER class, which is reserved for students with Higher technical educational backgrounds.

Figure 3: MATE II logo

4.5.1 MATE ROV Competition

The information about the competition is retrieved from the competition manual [1].

This year's competition themes are no different from the previous years, where they highlight the importance of the United Nations Decade of Ocean Science for Sustainable Development (2021-2030). Their initiative is to increase ocean knowledge and ensure that society implements this knowledge, thus contributing to the United Nations Sustainability goals. The task this year is to create a ROV, and additionally a scientific float. The themes raised this year are the facilitation and production of clean energy, surveillance, and tracking of the ocean's biological diversity.

Figure 4: MATE Competition logo

4.5.2 Points

Below is a table, demonstrating the segmentation of the available points. Product demonstration is the first part, where UiS Subsea will solve 3 practical tasks within 15 minutes. If this is achieved, additional points are given, 1 per minute and 0.01 per second. Additional points are given for ROVs below 25 kg and good teamwork during the competition. These practical tasks are meant to test the operational characteristics of the ROV. The secondary segment awards points for the technical documentation and how the organization portrays itself. The final points are given based on the safety of the ROV and how the relevant dangers have been adequately analyzed and addressed.

Table 1: Points structure

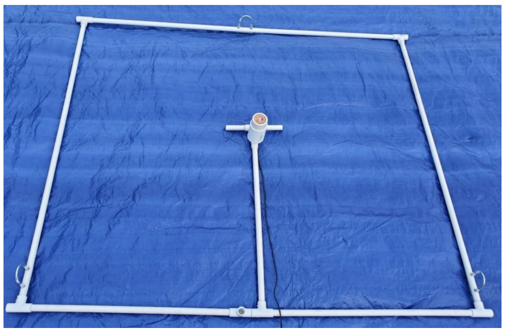
Product Demonstration	
Tasks	300 points
Time bonus	10 points
Weight Restrictions	10 points
Organization efficiency	10 points
Engineering and Communication	
Technical Documentation	100 points
Product presentation	100 points
Marketing	50 points
Company specification sheet	20 points
Company responsibility	20 points
Safety	
Review of safety documentation	20 points
Safety inspection	30 points
Safety job analysis	10 points
Total	**680 points**

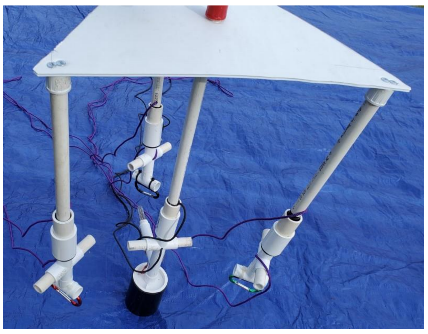
4.5.3 Task 1: (Maritime renewable energy)

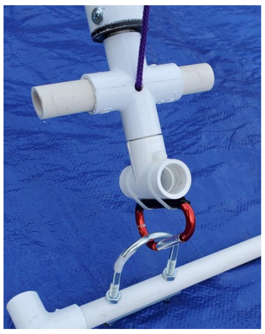
UNs Sustainability goals:

- 7. Clean energy for all
- 12. Responsible consumption and production

The first task is set in a design that simulates an offshore installation of floating solar panels in an established floating wind farm, the task is to remove biofouling and pilot the ROV either autonomously or manually into an underwater docking station


(a) Seabed anchor for installation

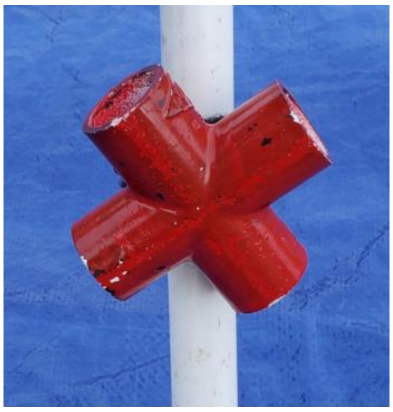

(b) Floating solar panel

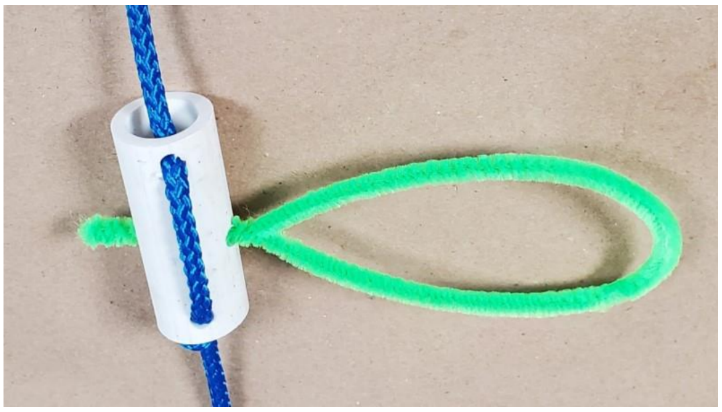

(c) Hook for mooring

1.1: Installation of a collection of floating solar panels

- Manoeuvre the solar panels between three existing wind turbines: 10 points.
- Moor 3 moorings to the solar panels: 15 points.
- Remove the lid from power port entry: 5 points.
- Connect plug from solar panels: 10 points.

1.2: Remove biofouling from the floating wind turbines Biofouling is simulated either with red PVC pipes connected with Velcro or chenille pipe cleaners twisted together.


(a) Biological material on structure


(b) Biological material on rope

- Remove 1-2 units av biofouling: 5 points.
- Remove 3-5 units av biofouling: 10 points.
- Remove 6 units av biofouling: 15 points.

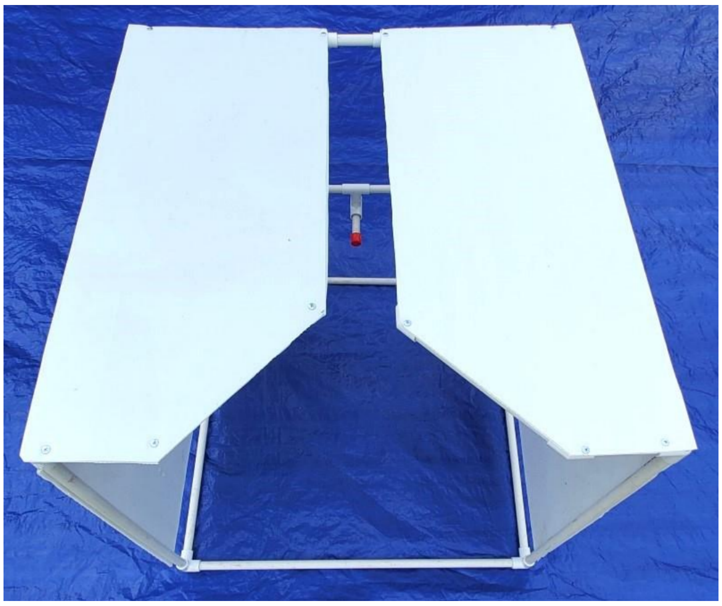


Figure 7: ROV Docking station

.

1.3: Maneuver the ROV into docking station

- Maneuver autonomously into docking station: 15 points.
- Maneuver manually into docking station: 10 points.

4.5.4 Task 2A: Coral reef and blue carbon

UNs Sustainability goals:
# 13 Stop climate change
# 14 Ocean life
The second task is divided into 2 parts: 2A and 2B. Part A represents scientific tasks: scanning a coral reef, identification of organisms by utilizing eDNA, exposure to UV light on sick coral reefs, and inspection and installation of an environmental mooring system to protect seaweed on the seabed.

Figure 8: Main coral with white spots

.

2.1: Measure, model, and identification of disease on coral reef

- Measure diameter on a coral reef: 5 points.
- Measure height on a coral reef: 5 points.
- Measure area of infection: 5 points.
- Make a 3D autonomously: 15 points.
- Make a 3D model in CAD manually: 5 points.

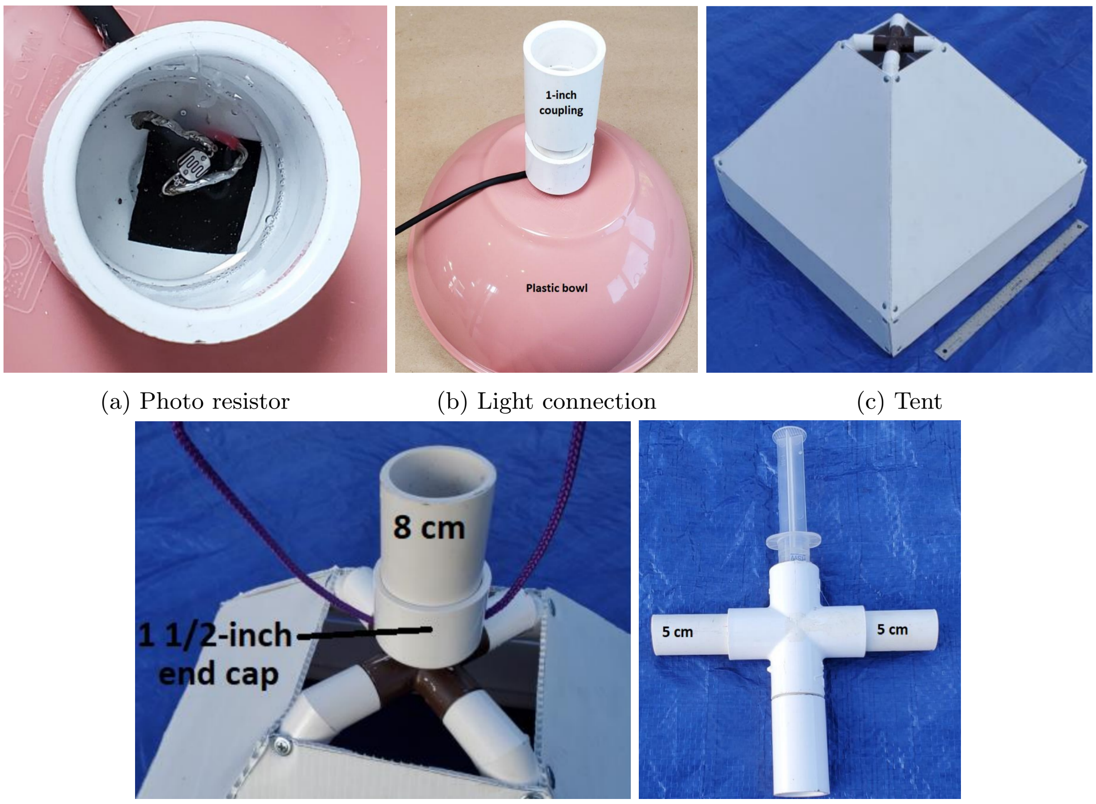
All measurements are to be done within 2cm and can be completed either autonomously or manually with reference objects.

(a) Water bag connection

(b) Connection for sample extraction

2.2: Identify coral reef organisms with eDNA

- Retrieve water sample from the bottle: 10 points.
- Identify fish species based upon 3 samples provided by hosts: 5 points.

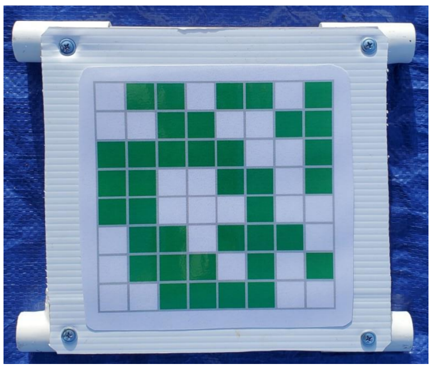


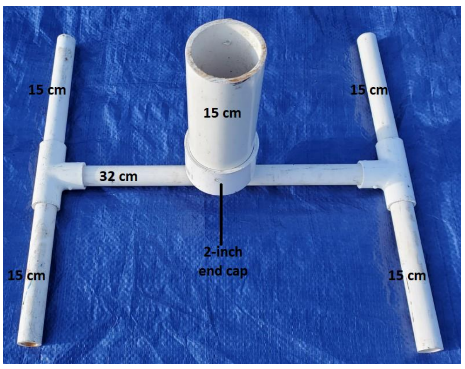
(a) Photo resistor (b) Light connection (c) Tent

(d) Syringe connection (e) Syringe

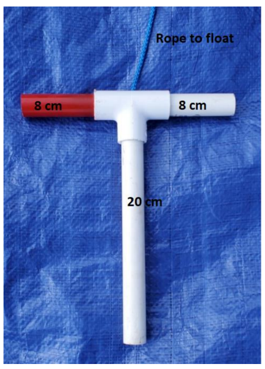
2.3: Administrate Rx? to infected coral

- Position UV-light above infected area: 5 points.
- Activate light and cure: 5 points.
- Place tent above coral reef: 10 points.
- Place syringe in tent opening: 5 points.
- Remove syringe contents inside the tent: 5 points.


(a) Seaweed



(b) Eco-Mooring base



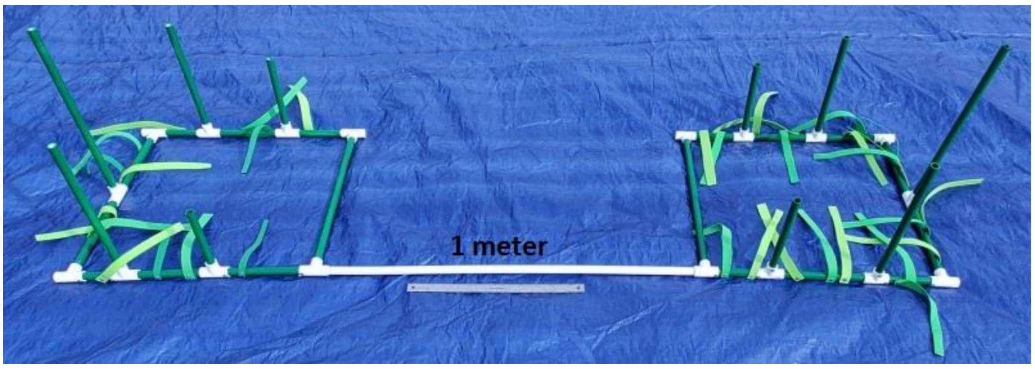
(c) Mooring

2.4: Seaweed habitat protection and surveillance

- Identify if seaweed habitat has been rehabilitated, remained unchanged, or worsened, based upon images: 5 points.
- Install Eco-Mooring system on the seabed, inside a base and rotate the mooring 720°in the base: 10 points.

4.5.5 Task 2B: Lakes and rivers

UNs Sustainability goals:
# 13 Stop climate change
# 14 Ocean life
Task 2B is more inclined to work with freshwater bodies. The tasks are to look for fish and determine if they are an invasive species and release fry where it is safe. There also is rope inspection, clearance of a larger object, following a transect line, counting frogs, and installing a camera on the seabed.



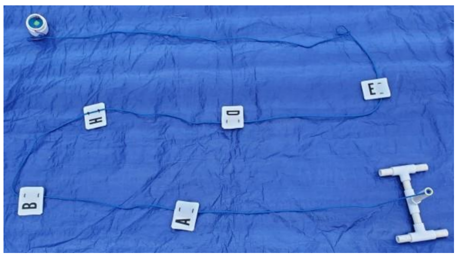
(a) Habitat area

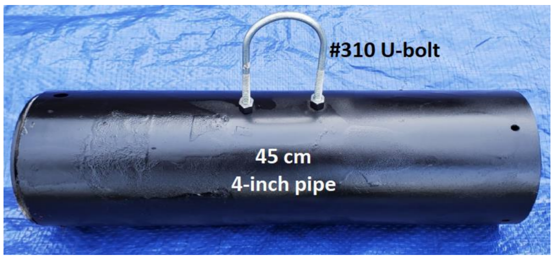
(b) Fry

(c) Current fish species

2.5: Re-introduce endangered species of Northern Redbelly Dace fry

- Survey 2 areas, and identify which is safe to place to release the fry: 10 points.
- Acclimatize fry to a safe area: 5 points.
- Release fry in the safe area: 10 points.


(a) Rope with letters

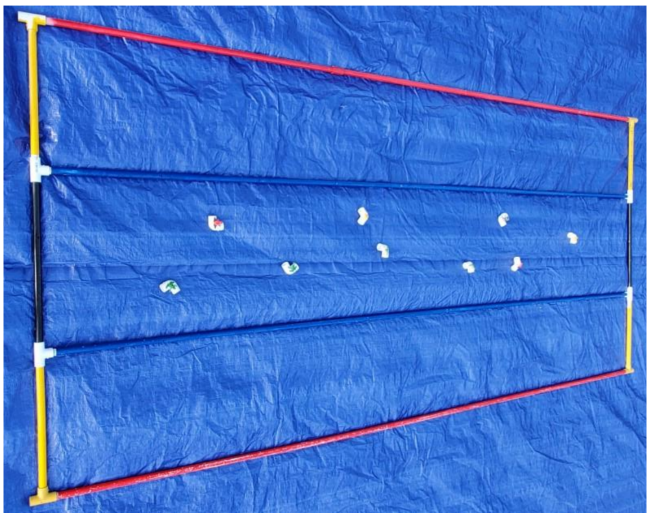


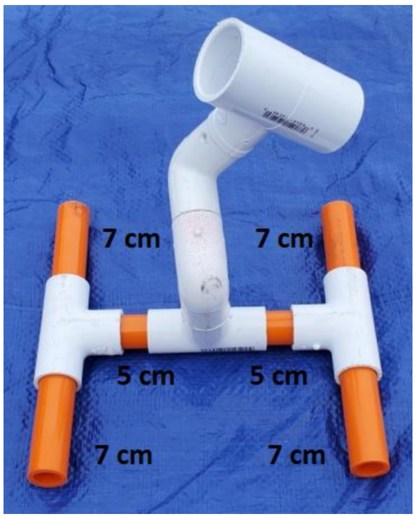
(b) Heavy object which is to be removed

2.6: Ensure the health and safety of the Dillion reserve

- Inspect rope for a buoy and display the 10 letters attached: 10 points.

- Display the documentation of the ROVs lifting capacity: 5 points.
- Lift object a maximum of 120 Newton above water: 10 points.
- Return object to land: 5 points.


(a) The transect area that the ROV will fly over



(b) The camera which is to be installed

2.7: Surveillance of endangered Lake Titicaca frogs

- Fly a transect line and maintain the image within the area: 10 points.
- Count amount of frogs within an area: 5 points.
- Install a camera in the designated area: 5 points.

4.5.6 Task 3: (MATE Floats!)

UNs Sustainability goals:
# 13 Stop climate change
This task represents the construction of a functioning scientific float, which will transmit data when it reaches the ocean surface.[13]

3.1: MATE Floats! 2023

- Design and construct a functioning vertical profile float: 5 points.
- Float communicates with land before submersion: 10 points.
- Vertical profile 1, the float sinks and rises after impact with seabed: 10 points.
- Float transmits to land at the time of completion after the first vertical profile: 10 points.
- Vertical profile 2, the float sinks and rises after impact with seabed: 10 points.
- Float transmits to land at the time of completion after the second vertical profile: 10 points.

4.5.7 Restrictions and demands

The competition has certain physical restrictions with size, weight, the operational environment, and some electrical limitations. The only vehicle to be utilized is an ROV

Environment: The ROV shall be able to operate in fresh, salt, or chloride water, in a temperature span of 15 to 30 °Celsius

Materials: The ROV shall be able to operate at a minimum of 4 meters depth, whilst being under a maximum of 35 kg.

Tether length: The tether has to be long enough to operate within an area 10 meters from the edge and 4 meters deep. The topside control system can be up to 3 meters from the edge of the pool.

Thrusters: The thrusters shall be adequately protected and meet IP-20 standards. The thrusters shall be designed to operate underwater.

Electrical: The organizer provides a power supply for the ROV of 30A and 48 VDC. Conversion to lower voltages has to happen inside the ROV. There shall also be implemented overload protection on 150% of nominal power usage on the ROV.

Float: Batteries utilized shall be of the type: AAA, AA, A, A23, C, D, or 9V alkaline batteries. The float shall be protected with a 7.5A fuse. There must be a pressure relief valve with a diameter of a minimum of 2.5cm.

4.6 About ROV Project

4.6.1 ROV history

The creation of the very first ROV can be credited to Dimitry Rebikoff, with his invention shown below. [44] The aptly named Poodle was made in 1953, complete with a tethered connection and operated with a topside control panel.

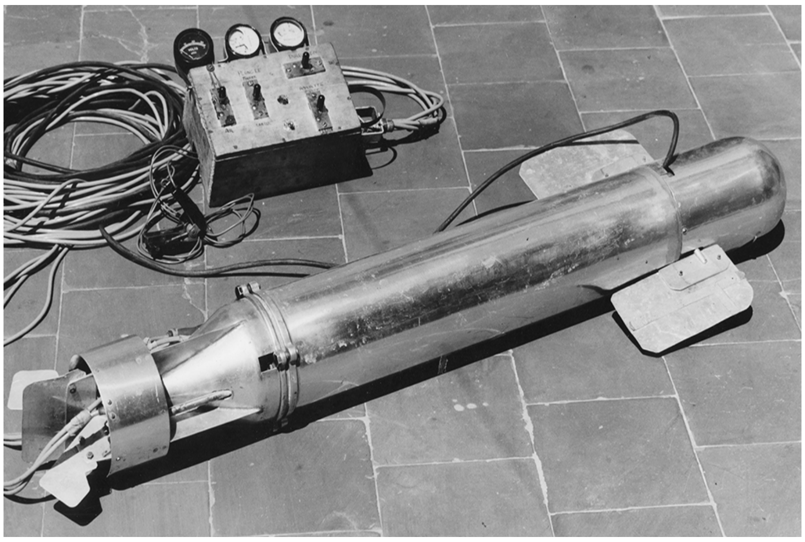


Figure 15: The world's first ROV

The ROV field has come a long way since then and modern ROVs are quite different. In the 1960s the U.S. NAVY utilized ROVs as recovery drones for underwater equipment.[16] Within a couple of decades, there were more ROVs, mostly in the commercial market, each with its own tasks and purpose.[20] There are a couple of common components that are standard on ROVs, being: [44]

1. Thrusters
2. Tether
3. Camera
4. Lights
5. Frame
6. Pilot controls
7. Buoyancy element

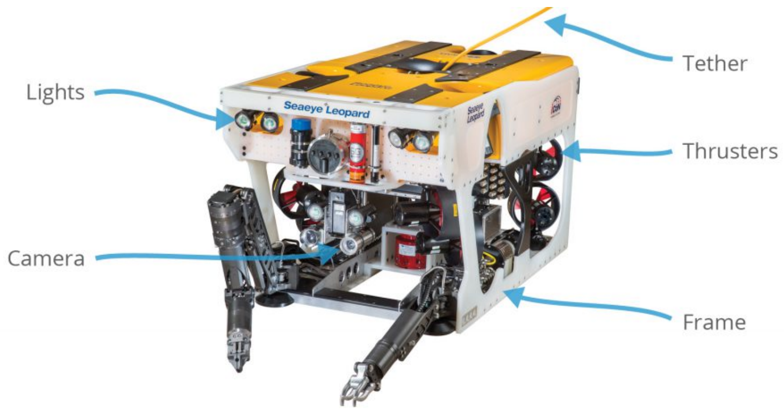


Figure 16: Diagram of common components

Modern ROVs are designed for a given task, some of these can be: observation, high-speed survey, inspection, trenching, burial, intervention, and construction.[35] Some ROVs can be used for a multitude of tasks whilst others are limited in their design. There are seven main classes of ROVs, from I to VII [35]:

From left to right and top to bottom:

I - Pure observation

II - Observation with payload option

III - Work class vehicles

IV - Seabed-Working vehicles

V - Prototype or development vehicles

VI - Autonomous underwater vehicles (AUV)

VII - High-Speed survey vehicles

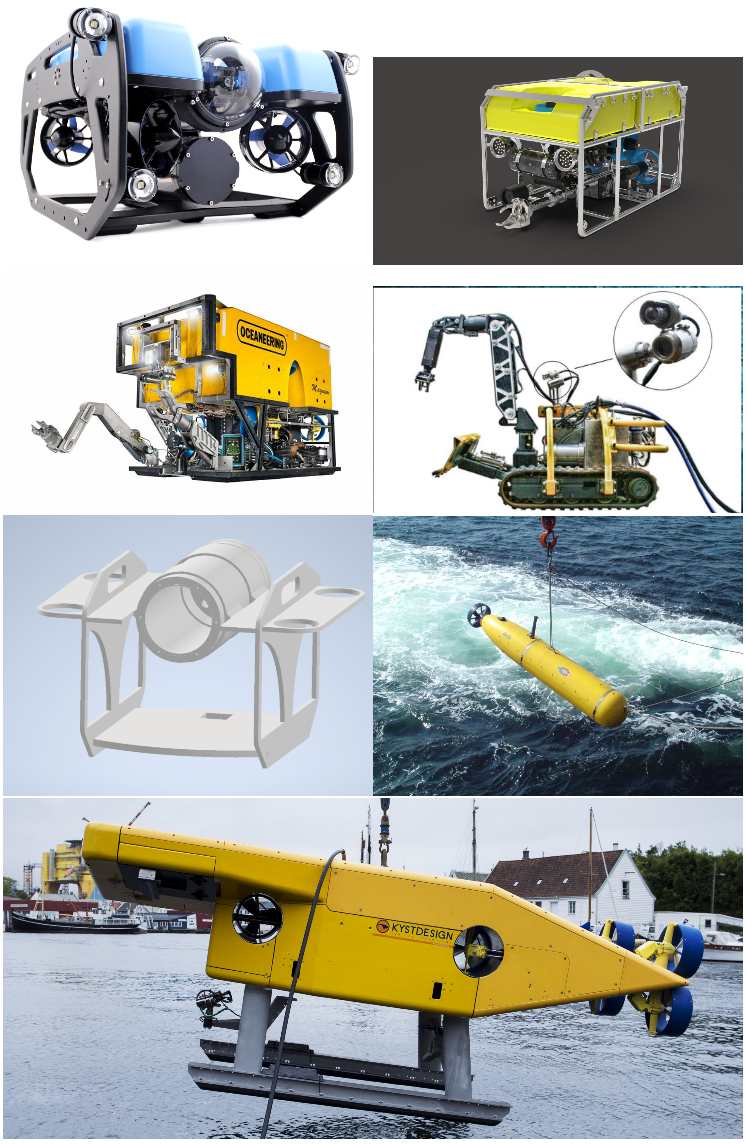


Figure 17: The different ROV classes

Each class has its benefits and limitations. Class I: Pure observation vehicles are physically limited to video observation, however, they are highly maneuverable.[35] Generally, they are small vehicles fitted with video cameras, lights, and thrusters. They cannot undertake any other task without considerable modification.

Class II – Observation ROVs with payload have the same capabilities as a pure observation ROV, but usually with additional functionality, such as a manipulator, color cameras, additional cameras, sonar, and cathodic protection measurement system.

Class III: Work class vehicles are large enough to carry additional sensors and/or manipulators. They have semi-autonomous capabilities, also known as multiplexing capability, which allows heavier equipment to run without being *hardwired* through an umbilical system. Furthermore, they have enough stability and buoyancy to carry additional detachable equipment without loss of functionality. This class is larger than the previously mentioned, with three sub-classes based on power rating:

1. Class III A – Work class vehicles $<$ 100 Horsepower
2. Class III B – Work class vehicles 100 Horsepower to 150 Horsepower
3. Class III C – Work class vehicles $>$ 150 Horsepower

Class IV: Seabed working vehicles are utilized as the name suggests. They maneuver on the seabed by a wheel, belt traction system, thruster propellers, water jet power, or a combination of the mentioned. These vehicles are usually even larger than Class III, with their main purpose being underwater work: dredging, mining, cable and pipeline trenching, excavation, and other construction work.

Class V: Prototype or development vehicles are classified as under development or have not been sufficiently tested. Most special-purpose vehicles or one-off prototypes end up here since they cannot be categorized by any of the previous classes. Both Class VI and VII are in this class according to the Norwegian standard of ROVs, since they are still under development and only a select few companies produce these kinds of vehicles.

4.6.2 Our ROV

The ROV developed and produced this year is a class 2 vehicle, with 6 degrees of freedom and a compact design. The goal is to create a vehicle based upon well thought existing solutions, whilst still being in accordance with UN Sustainability goals.

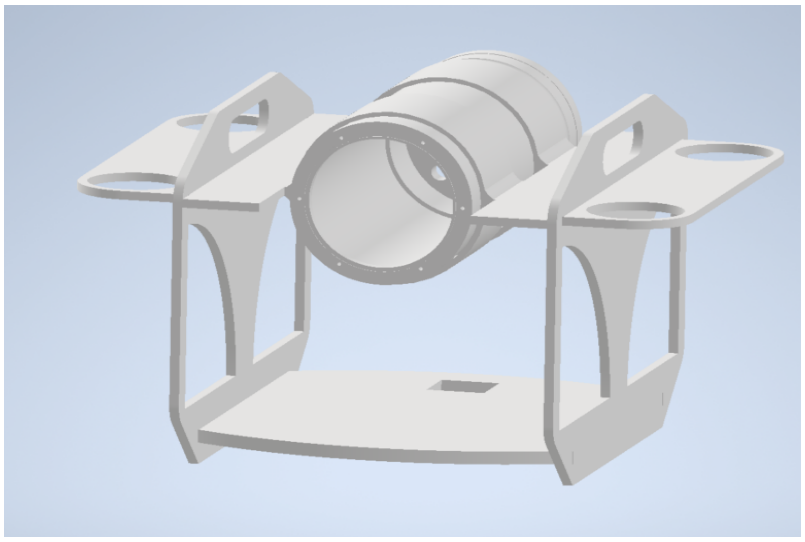


Figure 18: 3D model of ROV

The ROV is operated with a controller and a customized GUI, which communicates via an umbilical cord to the ROV. The main goal of the project is to compete at the MATE ROV competition and thus the vehicle is designed for this purpose. Additionally, the project participants have set a personal goal to be able to operate at 100m depth. The design is modular, something that makes parts easily replaceable and further developed, both as an ROV and AUV (Autonomous Underwater Vehicle)

4.6.3 Float

Scientific floats have been utilized for a long time, ever since Henry Melson Stommel came up with the idea back in 1955.[15] A float is a device separate from the ROV that contains various sensors for acquiring data from being dropped into water, and a balloon that inflates after some time to make it float back up to the surface.[11] A typical float will look like this:

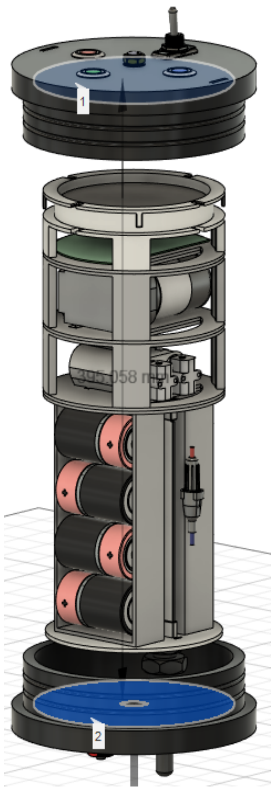


Figure 19: Typical frame of a float.

The main goal of a float is to track and monitor deep drift currents, and the first was manufactured of aluminum with a depth rating of 4500m.[7] Using a buoyancy engine it changes its depth to preset heights. A biogeochemical float uses an array of optical and chemical sensors to gather valuable data at otherwise difficult locations.[12]

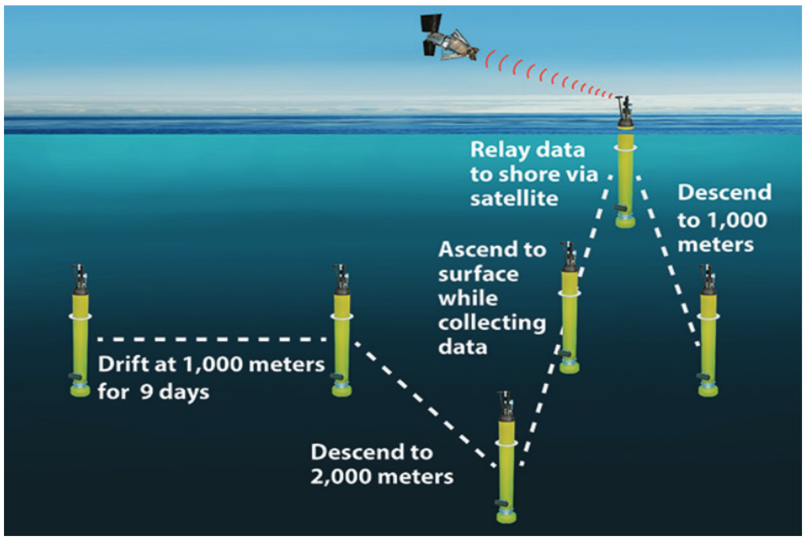

Relay data to shore via satellite
Descend to 1,000 meters
Ascend to surface while collecting data
Drift at 1,000 meters for 9 days
Descend to 2,000 meters

Figure 20: Figure of a floats cycle

Figure 20 Shows a normal float cycle. Firstly, they descend to a depth of 1000m, drifting for 5-10 days while acquiring valuable data.[10] This is again repeated at 2000m and finally ascends to the surface for transmission of the data.

4.7 This Bachelor's Thesis

This part will contain the main goal of this bachelor's thesis, as part of the larger subsea project. The mission is to provide logic for solving the tasks given by MATE. The tasks are as follows:

Transect maneuver and frog count

In this task, the ROV should be autonomously maneuvered across a designated transect. The transect has two blue pipes along the bottom, close to the middle of the track. There are also two red pipes further to the sides, also going along the bottom of the track. The task is to have the ROV always see the blue pipes, while never seeing the red pipes with the camera mounted on the bottom. Completing this gives 10 points. See figure 21 for a visual of the transect.

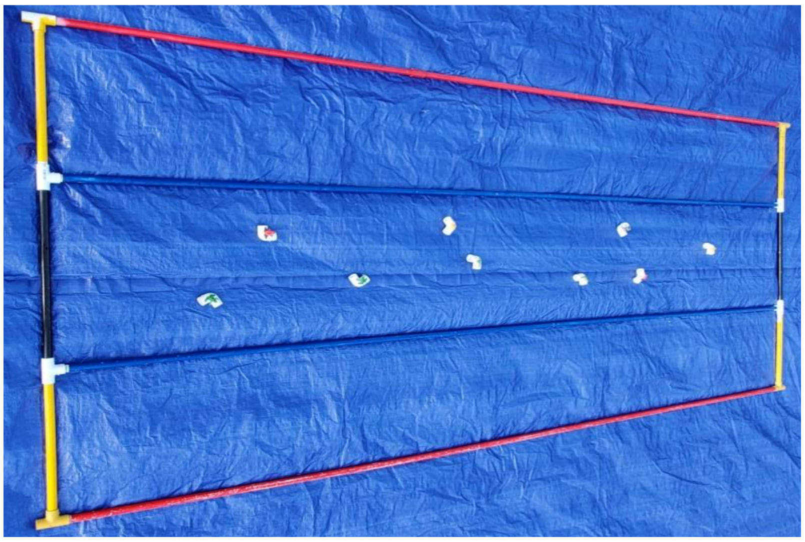


Figure 21: Figure of transect provided by MATE.

In addition to the red and blue pipes, there are also frogs placed along the bottom of the pool. While doing the transect maneuver, the ROV should also count the number of frogs. Getting the correct amount of frogs gives 5 points in the MATE ROV competition.

Autonomous docking

The next task is about docking the ROV autonomously in a designated area. There will be a big red button placed inside a square box. The ROV needs to drive into the docking station and push the button for the task to complete. Doing this autonomously gives 15 points, and manually gives 10 points. See figure 22 for a visual representation of the docking station.

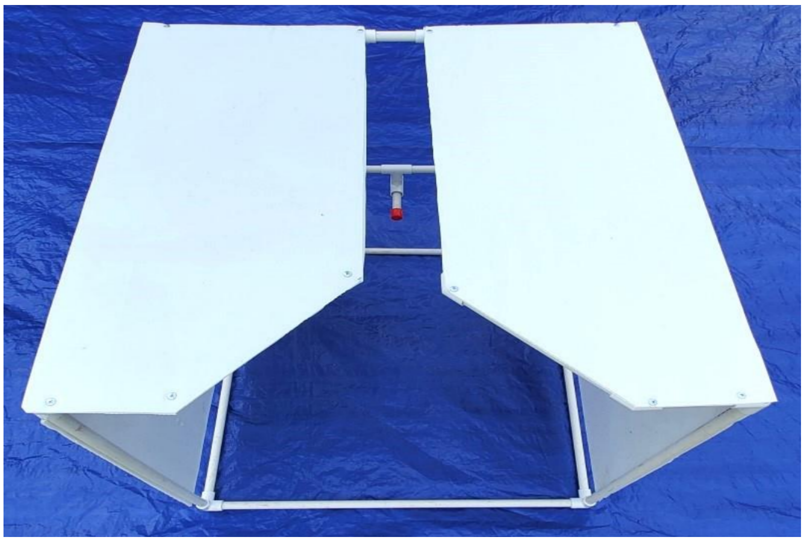


Figure 22: Figure of docking station provided by MATE.

Seagrass monitoring

For the last task, the ROV should take a picture before and after the simulated seagrass. The simulated seagrass is an 8x8 grid with green squares representing seaweed, and white squares representing seafloor. The amount of green squares might be different in the two pictures and the ROV should calculate the difference, if there is any. See figure 23

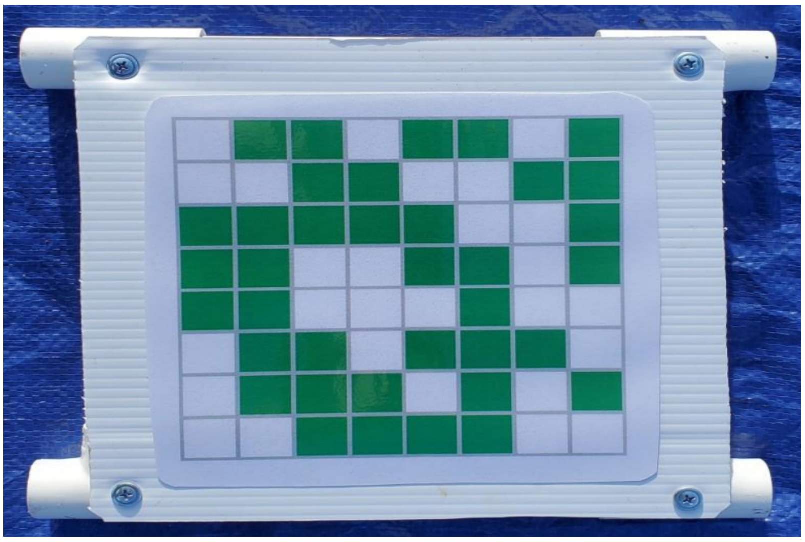


Figure 23: Figure of seagrass provided by MATE.

3D modeling of sick coral

The largest task, worth 30 points in MATE, is about modeling a coral in 3D and taking various measurements of the coral. Specifically the radius and height of the coral, as well as the area of the infected coral. This can also be done manually for fewer points.

Integration with GUI

Lastly, the code must be integrated with the GUI created by the GUI group.

5 Background

This section will provide the necessary information to understand the rest of the document, such as the programming language, frameworks, development methodology, and collaboration.

5.1 Collaboration

Solving specific or smaller problems as part of a large-scale project, we had to collaborate with other groups in order to connect all modules, programs, components, etc. For this bachelor's project as a software development group, we closely collaborated with:

- **GUI**
- **Communications**
- **Sensors**

We had to collaborate with the **GUI** group to implement the programs into the GUI. This was so that a user can run the software we created for the ROV in a user-friendly way. The GUI group was also responsible for sending driving commands down to the ROV. Because of this, we had to work with them on the autonomous driving functionalities when sending driving commands down to the ROV. See figure 24 for visual representation of the GUI.

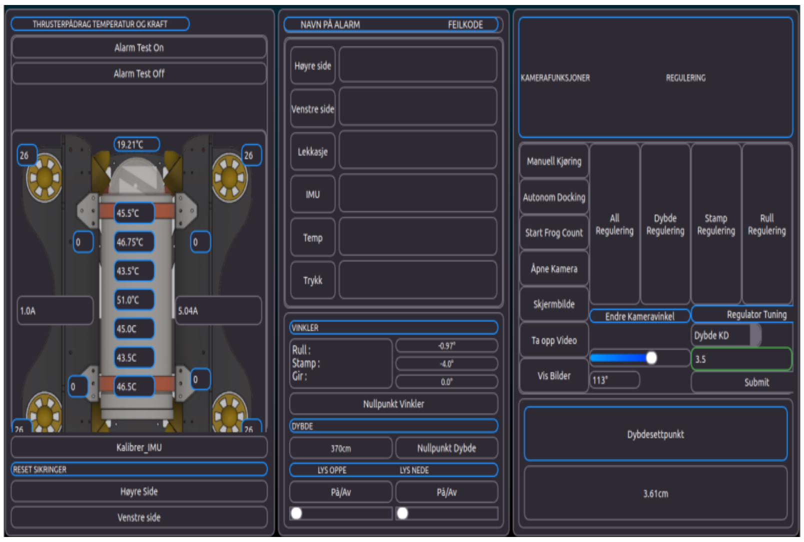


Figure 24: Screenshot of GUI. Notice the dedicated buttons for Autonomous Docking and Start Frog Count.

We worked with the **Communications** group to receive the camera feed as a video stream from the ROV. Together with them, we had to choose what streaming protocol to use, which ports to send the stream through, and what framework/technology to use.

The **Sensor** group was responsible for deciding and implementing sensors, such as sonar. Therefore we had to meet and discuss with them what sonar options were available. Ultimately it was decided not to install a Sonar on the ROV and rather focus on other tasks instead.

To communicate and work with the other groups, we used the UiS Subsea discord server to ask questions, set up meetings, or share material, links, code, etc. When working collaboratively to solve problems and test communication between topside and ROV systems, we would meet at either the Subsea room or electronics lab at the University, where we would connect components and assess their functionality through testing and troubleshooting.

For the collaboration of code development and implementing new code into the existing code base, we used GitHub. GitHub makes it easy to stay up to date with any code changes made by partners and coworkers. This year, we set up a GitHub organization for all the bachelor groups to have their own designated repository. Within our group's repository, we utilized branches to separate different tasks to work on and prevent merge conflicts.

5.2 Development methodology

This section explains what development methodologies we used when developing the programs and the reason why we chose to follow these principles. This project requires code and features to be implemented into a large existing code base. And to be easily maintained and built upon by future bachelor students. Because of this, it is important to apply good principles and habits to the new code implemented. For this project, we chose to follow principles from two well-known development methodologies, TDD and DDD.

5.2.1 Test-driven development

For most of our development process, we used principles from TDD. [41] This is a software development process where each feature or functionality is tested before the final implementation. The process involves writing tests before the code, then writing the code to pass the tests, and then refactoring it into a clean and maintainable final product. While we did not write the tests before the code itself, we wrote the tests while we wrote the code.

The process looked like this:

1. Problem
2. Idea for solution
3. Try writing code idea
4. Write several different tests
5. Continue developing and improving the code until it passes all tests in a clean and efficient way

Developing like this made it secure for us to tweak or alter different parts of the code, without getting unforeseen bugs or losing some functionality. It also made it easier to test performance, run time, and accuracy. In general, it made it easier for us to test and tweak the code in order to improve it.

The reasoning behind following these principles is that the ROV we intended to write software for was not built yet, and therefore we could not easily test our implementations. By making extensive and challenging tests for the code while developing, we could better test the functionality and also improve it as we were developing. Another reason for making many different tests is because of water. The ROV is going to run and perform tasks in a pool or the ocean. The water is constantly moving, leading to variations in the light and lots of noise in the camera vision. It is important that the code is stable and robust, and able to handle different lighting, contrasts, and angles. The ROV is also moving, relative to the water and its surroundings, creating further variations. Ensuring the ROV is able to perform under a variety of conditions, requires a variety of challenging tests.

Pros of test-driven development:

- **Early detection of bugs.** Saving time and effort if a bug or small problem occurs.
- **Improved code quality.** TDD encourages the development of simple and clean code. The process of writing tests first helps to define the requirements and ensures that the code meets those requirements.
- **Easier testing of code.** Makes it easy to test multiple things at once, saving time for the developer.
- **Better documentation.** Tests not only serve the purpose of making sure a block of code performs correctly, but also serve as a type of documentation. It explains how a block of code is supposed to work, inputs, outputs, etc.

Cons of test-driven development:

- **Time consuming.** Writing tests requires planning and preparation. Depending on the tasks and thoroughness of the tests this could prove to consume more time than it saves.

- **Overhead of maintaining tests.** Tests need to be maintained and updated as the code evolves. This can become a burden if the tests are not written in a clear and maintainable way.

After valuing the pros and cons of TDD we decided that the pros outweighed the cons, and would lead to an overall better product. More stable, more secure, easier to test, and easier to maintain.

5.2.2 Domain-driven design

For this project, it is important to write clean, understandable code that is easy to improve or build upon, but also easy to maintain. The reason for this is to create a clean environment for us to work within and also for future Subsea bachelor students to understand and build upon. To achieve this goal we used some of the principles we learned in the course DAT240 (Advanced Programming) about DDD [**asad_domain-driven_2020**]. This development methodology is a software development approach that focuses on modeling a complex software system and aligning the software design with the specific tasks it is meant to serve.

Pros of domain driven design:

- **Increased flexibility.** DDD encourages modular and flexible design, allowing the software to evolve and change as there is a need for new features.
- **Better alignment with the goals.** By focusing on the core problems and objectives, DDD helps ensure that the software design is aligned with the tasks it is meant to serve.
- **Improved communication.** DDD helps improve communication between domain experts and software developers by providing a common language and framework for discussing complex domain models. In this case, it would be communication between our bachelor group and the rest of the Subsea team.
- **Better understanding of complex domains.** DDD provides a structured approach to modeling complex domains, helping software developers gain a deeper understanding of the problem space and the relationships between the different concepts and objects within it.

Cons of domain driven design:

- **Increased complexity.** DDD can add complexity to the software design process, particularly for teams that are not familiar with the DDD approach.
- **Higher initial investment.** DDD needs more planning in the early phase of the development, to make sure the software is properly and efficiently structured according to the business domain.

- **Resistance of change.** Once a context has been defined it can be difficult to undo or change the boundaries of the context.
- **Risk of over-engineering.** DDD is useful for large programs that need a proper structure to be developed. For a small program, it may add unnecessary complexity, leading to overall less efficiency in the project.
- **Lack of Standardization.** There is no standard way of implementing DDD it is very specific to the individual program, which could lead to confusion and misunderstandings among different teams.

Typically DDD is applied to a business context where the key idea is that the software design should be driven by the underlying business domain, rather than being based on generic technical solutions. Though our project is not large enough to justify the complete application of DDD, there are still valuable concepts that can be applied to this project. These are the concepts/ principles we focused on while developing the software:

- **Dividing large parts into smaller ones.** Creating a more clear space to work within, making it easier to implement and easier to maintain.
- **Modular and flexible design.** Avoid direct relations and coupling, making it easy to swap or change parts. Providing overall flexibility and stability to the program.
- **Ubiquitous language.** Using a shared vocabulary that can be understood by all participants in the industry.

5.3 Programming language

This section will explain why **Python** was chosen as the main programming language for the project, as well as discuss the pros and cons of using this language.

Our task in this bachelor's is to use image processing to solve computer vision tasks. Python is one of the most used and well-documented programming languages in the world and has a lot of existing libraries and support for image processing, such as OpenCV (open-source computer vision). [43] This is useful since image processing is highly taxing on the computer and requires complex matrix operations to be done, which libraries like OpenCV do efficiently.

Because of Python's popularity and large user base, many common problems within image processing and computer vision have been solved and well-documented online. Since we did not have any background in image processing when we started this project, it was important to have lots of good material to study when solving our specific tasks. [23]

Pros of using python:

- **Ease of Use**: Python is easy to learn, write and read, thanks to its simple syntax and structure.
- **Fast development**: Because of its simple syntax, writing code is fast reducing the overall development time of a project.
- **Large Library**: Python comes with a large library that includes many useful modules and functions.
- **Open-Source**: Python is an open-source language, meaning that it is free to use and distribute, which is one of the reasons why it is so popular.
- **Cross-platform compatibility**: Python can run on different operating systems like Windows, Linux, and MacOS without any changes making it flexible for different users.
- **Versatility**: Python is a versatile language that can be used for web development, data analysis, machine learning, and scientific computing.

Cons of using Python:

- **Interpreted language.** Python is an interpreted language, which means it has a slower execution speed than compiled languages such as C++, Java, etc.
- **Global Interpreter Lock.** Python has a global interpreter lock (GIL), which can lead to performance issues when working with multi-threaded code. This is caused by the GIL preventing Python from exe-

cuting code in parallel.[**noauthor_globalinterpreterlock_nodate**] The purpose of this is to prevent race conditions and ensure thread safety.

- **Memory Management.** Python's automatic memory management can be a disadvantage in certain situations, leading to performance issues and unexpected behavior. TODO [**empty citation**]
- **Dynamic Typing.** While it is very comfortable to work with, Python does not ask for the types of variables and objects in the code and will spend time calculating them during runtime. This can delay the detection of errors and bugs.

Overall, after weighing the pros and cons of using the language, it was clear that Python was a good fit for this project.

5.4 Frameworks

This section contains a collection of frameworks that were utilized in the project.

5.4.1 OpenCV

OpenCV is a well-known and well-documented computer vision library designed for image and video processing. OpenCV is being used in this project for several reasons, including its reputation and extensive documentation [28].

- **Ease of use.** OpenCV provides a comprehensive set of functions and utilities that makes it easier to perform a wide range of image and video processing tasks.
- **Large Community.** OpenCV has a large and active community that contributes to its development and provides support to users.
- Compatibility: OpenCV is compatible with multiple operating systems and programming languages, including Python, which is the language we are using for this project.
- **Performance.** OpenCV is optimized for real-time processing, making it suitable for use in applications where speed is critical. This is important for this bachelor, given the image and video processing tasks are required to be done in real-time from the video stream coming from the ROV, as well as simultaneously being shown on the GUI.
- **Wide range of functionalities.** OpenCV has a wide range of functionalities, the specific ones used for this bachelor's thesis. This will be discussed later in this paper.

5.4.2 Notable OpenCV Functions

OpenCV offers a large variety of useful functions that are frequently used for image and video processing tasks. This section will highlight the ones that are key for this project. An in-depth description of these can be found in OpenCV's official documentation, [25].

- **findContours(...)**, is a crucial function in openCV. A contour can be explained simply as a curve joining all the continuous points (along the boundary). This means that the contour is a sequence of points that form a curve that represents the shape of an object or shape. These contours are very useful for detecting objects and analyzing shapes. Shown in figure 25

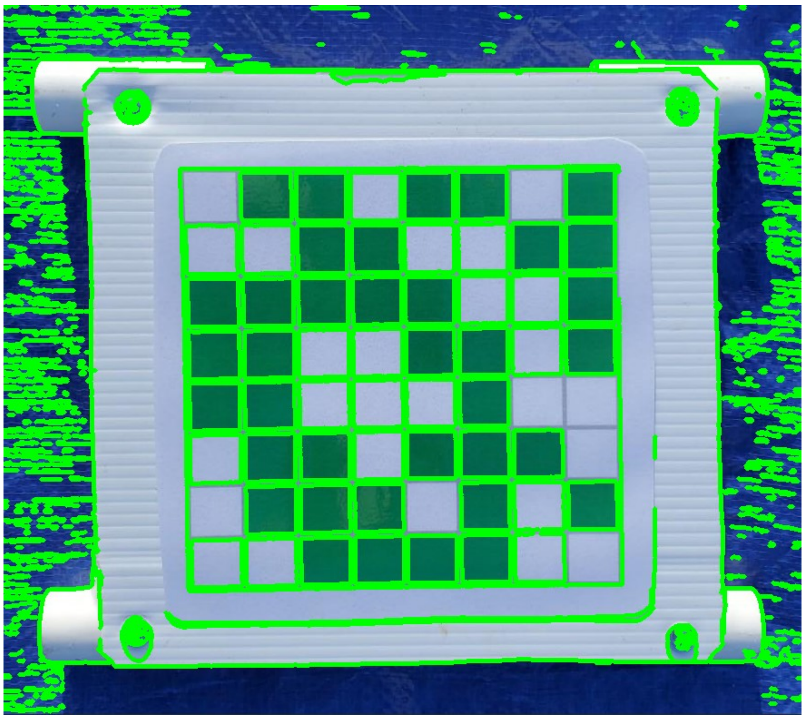


Figure 25: Contours highlighted in green.

- **cvtColor(...)**, This function's primary task is to turn images into grayscale. Shown in figure 26

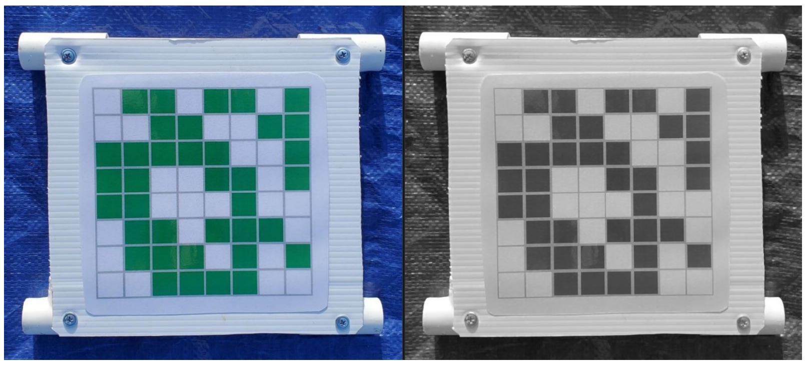


Figure 26: Color conversion from RGB to grayscale.

- **GaussianBlur(...)**, blurs the image and gives a more powerful blur based on input parameters. Frequently used as a technique to reduce noise in an image. Shown in figure 27

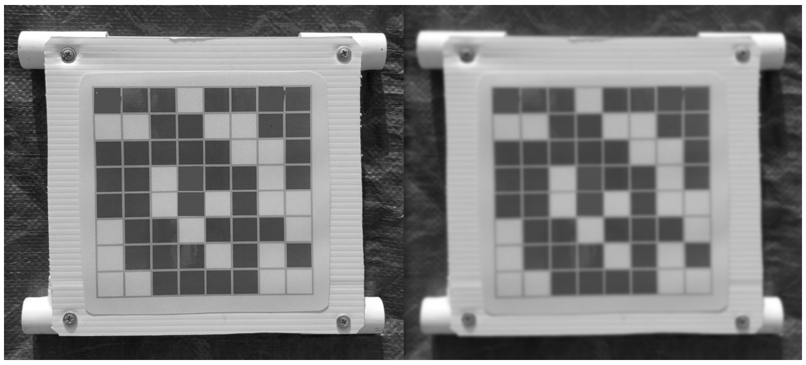


Figure 27: Grayscale to blurred

- **Dilate(...)**, dilates image based on input parameters, useful for making small details more pronounced, such as lines between squares. In other words, enhancing or enlarging contrasts in the image. Shown in figure 28

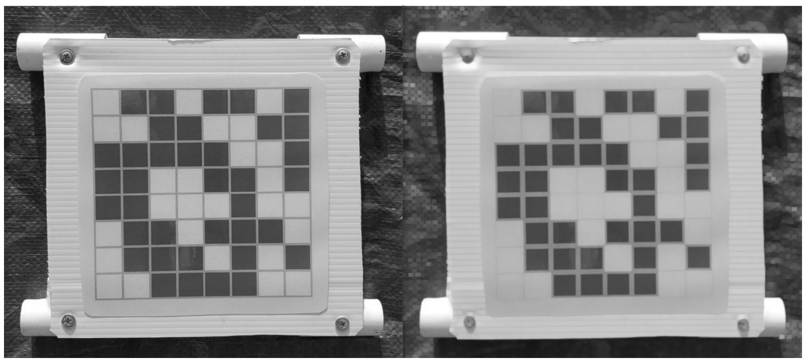


Figure 28: grayscale to dilated

- **threshold(...)**, turns all pixels in the image black if the pixel grayscale value is above a threshold, and white if it's below the threshold. Contains several tweakable options based on the thresholding method. More details can be read on OpenCV's site. Shown in figure 31

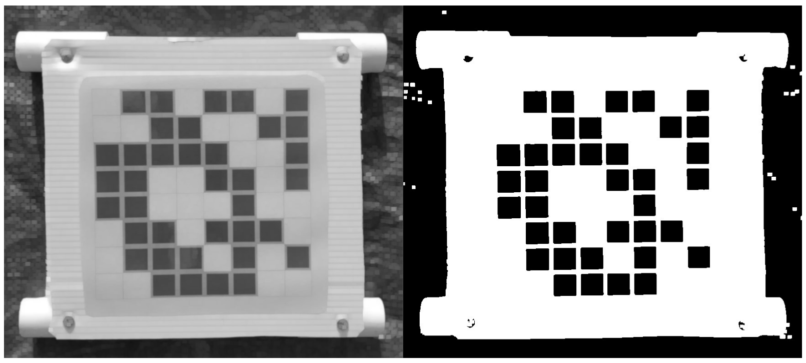


Figure 29: Binary threshold of dilated image, maxval = 120

- **inRange(...)** Used on an image to isolate only pixels in a certain BGR range. Isolated pixels are turned to white, and everything else turns black. Shown in figure 30

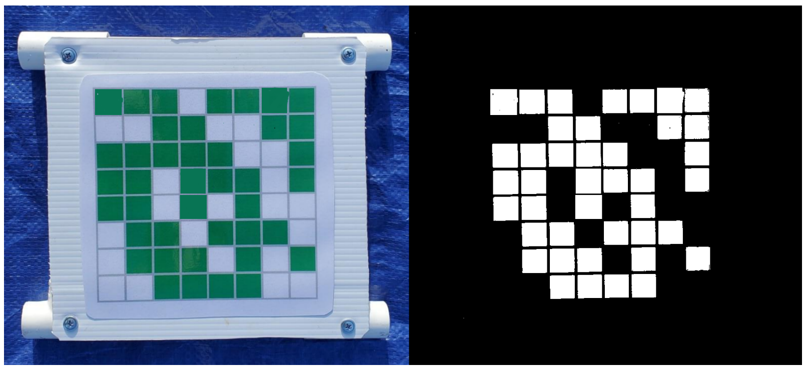


Figure 30: Thresholding using inRange() function. Isolates a certain color given as input parameter.

- **Canny(...)** Edge detection algorithm, more details at OpenCV [24]

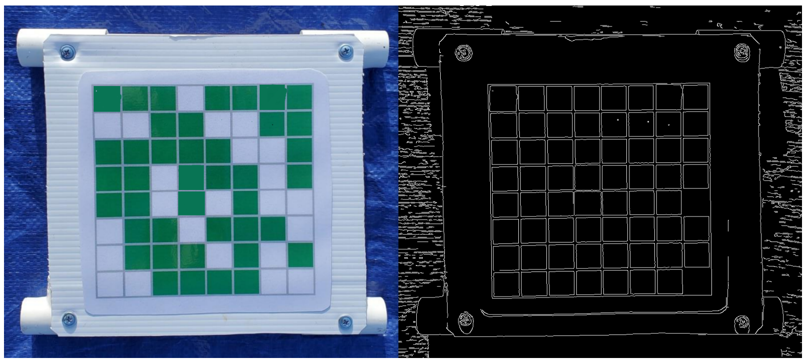


Figure 31: Visual representation of Canny edge detection, used to find edges and contours.

5.4.3 Python Pillow

Pillow is a part of the Python Imaging Library (PIL) and is considered a reliable library for several reasons [31]:

- **Compatibility.** Pillow is compatible with a wide range of image file formats, including JPEG, PNG, etc, making it a versatile choice for image processing tasks.
- **Easy to use.** The library contains basic image processing functionality, such as resizing, rotation, coloru space conversions etc. Additionally making it easy to load and store images.

5.4.4 GStreamer

To send the camera feed from the ROV to the topside system we decided to use **GStreamer**. Gstreamer is a widely used streaming framework for creating streaming applications. The way it does this, is by creating pipelines, encoding video data, and sending it over a network. It is frequently used in the industry for image processing and computer vision. Here are the main reasons why we decided to use Gstreamer for this project [3]:

- **Integrated in OpenCV**: GStreamer is compatible with OpenCV, it only needs to be downloaded and configured within OpenCV. Because of the compatibility with OpenCV, it makes it efficient and easy to get camera feed from the ROV and into the program. [27]
- **Integrated by Nvidia for Jetson**: Nvidia has integrated the Gstreamer framework on the Jetson mini-PC for image processing. Since we are using a Jetson as the mini-PC on the ROV, Gstreamer makes it efficient and easy to encode and decode the camera feed on the ROV before sending the data topside. [2]
- **Large Community**: Because of its popularity and common use in the image processing and computer vision industry, there is a lot of great documentation and material online.

Last year, they processed the image on the mini-PC on the ROV and sent the processed image topside using OpenCV's inbuilt streaming and video sharing functions. This suboptimal solution resulted in lots of delays and high latency. Image processing in itself requires lots of processing power. Because of this, the mini-PC throttled due to overheating and struggled with processing the images from the camera feed and sending it to the topside system.

Due to last year's problems with sending the camera feed from the ROV topside, we decided to take a different approach to the problem. Considering the pros mentioned above we decided that Nvidia Jetson paired with Gstreamer would be a better solution.

5.4.5 Python Unittest

For writing tests of code, we decided to use the integrated testing library standard to Python, called **Unittest** [42]. The benefits of this library are that it already comes preinstalled with Python, gives good feedback when a test passes or fails, and is easy to use. Though it is not particularly fast or efficient, this is not required by our programs since they are not large enough and performance impacts the time to test. Both unit tests and integration tests can be written using Unittest, which we will showcase later in this paper.

5.4.6 Multi processing

Python, whilst not having real parallel processing, combats this problem by using the multiprocessing module. Multiprocessing is a technique for achieving parallelism in Python [22]. This means that several processes can be run simultaneously without blocking each other. Python achieves this by running a separate interpreter process from the rest of the code, allowing concurrent code to run. The memory will not be shared between processes unless you make use of shared memory. This will be very useful for running all the different cameras at the same time, without the code stopping and waiting for the other to run.

The other option was the multithreading technique in Python, which tries to achieve parallelism by creating several threads within a singular process that can run code. This will have shared memory so they can access the same data and variables, but it can be a challenge when it comes to coordinating how the resources should be used. [38]

6 Results

This section will briefly show how we solved the tasks we were given. Further discussion and details behind the solutions are given in the discussion chapter.

6.1 Camera feed

The camera configuration on the ROV contains four cameras. Two cameras as a stereo camera setup in front. One camera pointing downwards and one camera on the manipulator. This configuration is made to give the user an overview of the surroundings of the ROV as well as provide information to the image-processing group.

As there is more than one camera on the ROV, multi-cast is utilized with Gstreamer to send the data topside. This was done via a physical network cable using UDP transport protocol. Multiprocessing code is used to handle all the streams. For this, the program uses Pythons multiprocessing module which allows the program to run a different process on each CPU core.

The video data is received by OpenCV through GStreamer, which processes and encodes the stream using a pipeline. These pipelines can be accessed with a GStreamer feed via OpenCV. For this to work a network cable has to be plugged into the ROV mini-PC, and it has to start the sending process. This is handled by the Communication group in UiS Subsea. The video stream sent this way is limited to 30FPS and shown in 1080p. For the image-processing task, this quality is more than adequate. In order to open the left stereo camera, the video stream is accessed using GStreamer through the following function:

```
GST_FEED_STEREO_L = "-v udpsrc multicast-group=224.1.1.1 auto-multicast=true port=5000 !
    application/x-rtp, media=video, clock-rate=90000, encoding-name=H264, payload=96 !
    rtph264depay ! h264parse ! decodebin ! videoconvert ! appsink sync=false"

cv2.VideoCapture(gst_feed, cv2.CAP_GSTREAMER)
```

Listing 1: Opening Gstreamer feed on port 5000.

This pipeline is going to be used by the other cameras as well, only changing the port value. See table 2 for an overview of all the streaming channels.

Port num.	Camera pos.	GStreamer link
5000	Stereo Left	"-v udpsrc multicast-group=224.1.1.1 auto-multicast=true port=5000 ! application/x-rtp, media=video, clock-rate=90000, encoding-name=H264, payload=96 ! rtph264depay ! h264parse ! decodebin ! videoconvert ! appsink sync=false"
5001	Stereo Right	"-v udpsrc multicast-group=224.1.1.1 auto-multicast=true port=5001 ! application/x-rtp, media=video, clock-rate=90000, encoding-name=H264, payload=96 ! rtph264depay ! h264parse ! decodebin ! videoconvert ! appsink sync=false"
5002	Down	"-v udpsrc multicast-group=224.1.1.1 auto-multicast=true port=5002 ! application/x-rtp, media=video, clock-rate=90000, encoding-name=H264, payload=96 ! rtph264depay ! h264parse ! decodebin ! videoconvert ! appsink sync=false"
5003	Manipulator	"-v udpsrc multicast-group=224.1.1.1 auto-multicast=true port=5003 ! application/x-rtp, media=video, clock-rate=90000, encoding-name=H264, payload=96 ! rtph264depay ! h264parse ! decodebin ! videoconvert ! appsink sync=false"

Table 2: This table shows which ports belong to what camera on the ROV. And what the GStreamer link is for the different ports.

The Camera class looks like this:

```
class Camera:
    def __init__(self, name, gst_feed = None):
        self.name = name # Name of camera
        self.gst = gst_feed # Gstreamer feed

    def get_frame(self):
        success, frame = self.cam.read()
        if not success:
            print("Error reading frame")
            return
        return frame

    def open_cam(self):
        self.cam = cv2.VideoCapture(self.gst, cv2.CAP_GSTREAMER)
            if not self.isOpened:
                print("Error opening camera")
                return False
            print(f"{self.name} Camera opened")
            return True

    def release_cam(self):
        self.cam.release()
```

Listing 2: Camera class

This class in listing 1 sets up methods for the following: opening a camera, extracting frames, and closing the camera. A manager class will utilize these methods.

A camera manager class is created in order to manage the camera objects. It is tasked with opening the cameras, running the methods from the camera class, as well as saving all cameras and frames in one location. Different methods are set up for starting up each camera object, using the code in listing 1 and closing all cameras when necessary. This class looks as follows:

```
class CameraManger:
    def __init__(self):
        self.cam_Down = None
        self.frame_Down = None

    def start_cam_Down(self):
        self.cam_Down = Camera("Down", GST_FEED_DOWN)
        success = self.cam_Down.open_cam()

    def get_frame_Down(self):
        self.frame_Down = self.cam_Down.get_frame()

    def close_Down(self):
        self.cam_Down.release()
```

Listing 3: Camera manager class, this is only showing code for one specific camera to reduce size

The camera manager class is populated with all the other camera and frame properties not shown above to save space. Other cameras used follow the naming scheme listed in table 2. Names of the methods follow the pattern for start_cam, get_frame, and close using the following names: StereoL, StereoR, Down, and Manip. This makes the class a central area for the Execution class which will be used to connect the image processing classes to the GUI. It is essential for the image-processing functions to get in a frame for performing their respective tasks.

This class also has methods for taking a picture and recording a video. These are important features for capturing an important frame, or when a video is needed for testing. The code for taking a picture utilizes openCV's image write function as follows:

```python
class CameraManager:
    def save_image
        for cam in self.active_cameras):
                frame = cam.get_frame()
                cv2.imwrite(f"path/to/image/"{cam.name}{datetime.datetime.now()}.jpg", frame)
```

Listing 4: method for saving a picture frame

This method adds a list called active_cameras that gets appended with a Camera object whenever it was started with code from listing 3. Running the method will save the current frame from each active camera. It uses the date and time at the moment and the name of the camera to differentiate what is being written.

6.2 Execution Class

This project has many different classes that solve the different tasks, but nothing connecting them to the overarching GUI. That will be the primary purpose of the execution class, having a common class to unite all the other classes. This approach is modular in design and allows us to change each functional class as required without affecting the executing class or GUI. With this method, the needed functions are easy to bind to any GUI buttons or keybindings.

The first thing that was agreed upon between the computer science groups and the electronic groups was the structure of the driving commands that were to be sent down. The electronic groups responsible for applying the driving commands to the ROV decided that it needed four degrees of freedom for the movement of the ROV. They did also require four more values for other tasks that do not relate to movement and therefore will be set to zero. The values that will be sent will be:

- x = amount of movement in the x-axis from 0 to 100
- y = amount of movement in the y-axis from 0 to 100
- z = amount of movement in the z-axis from 0 to 100
- r = amount of rotation from -100 to 100, where negative numbers are counter-clockwise rotations and positive numbers are clockwise rotations.

The driving packet will then look like: [x,y,z,r,0,0,0,0]

The primary classes we need to run from the Execution class are as follows; Autonomous Docking Class, Autonomous Transect Class, Seagrass Monitor Class, and the Camera class. These classes are imported from their respective files and initialized within the init method of the Execution class. All of the autonomous classes have run methods that only take in a frame, while the classes handle the rest of the logic. The frames are acquired from the code in section 6.1.

In the execution class, we initialize the camera manager class. From there the cameras can be started and start extracting frames which will be funneled into the execution class. These are then used for the core functions such as autonomous docking. Here is how it looks in the Execution class init method:

```python
class ExecutionClass:
    def __init__(self, driving_queue, manual_flag):
        self.AutonomousTransect = AutonomousTransect()
        self.Docking = AutonomousDocking()
        self.Seagrass = SeagrassMonitor()
        self.CameraManager = CameraManager()

        self.done = False
        self.manual_flag = manual_flag
        self.driving_queue = driving_queue
```

Listing 5: Execution Class init method.

The class will initialize the classes required for different tasks as well as the camera manager. This makes it easy to extract frames from the required camera as talked about in section 6.1. The driving_queue is a multiprocessing queue that is created in the main file that runs the GUI made from the other group. This means that the queue can share data between processes and can handle inputs and outputs from several locations simultaneously. It also includes a done flag, and a manual flag, which will be touched on later in this section. For using autonomous docking it would look like this:

```python
class ExecutionClass:
    def docking(self):
        self.done = False
        self.CameraManager.start_stereo_cam_L()
        self.CameraManager.start_down_cam()
        while not self.done and self.manual_flag.value == 0:
            self.CameraManager.get_frame_StereoL()
            self.CameraManager.get_frame_Down()
            driving_data_packet = self.Docking.run(self.frame_stereoL, self.down_frame)
            self.send_data_to_rov(driving_data_packet)
        else:
            self.stop_everything()
```

Listing 6: Docking method in execution class.

This code in listing 6 will run until done or manual control is activated. This means that it will loop until the done flag is set to True, which happens whenever manual driving mode is entered, or the autonomous task is finished. When manual driving mode is triggered the manual mode flag will turn True. This flag is a value datatype from Python's multiprocessing module. This property allows several processes to access the value and change it when required. A change to the manual flag happens whenever manual mode is entered through a button or the controller is used. Any movement on the controller will turn on manual mode.

The required cameras that are included are as follows: the left stereo cam and down-facing camera, and the manipulator camera. It then loops while extracting frames from the camera and sends it into a Docking.run function. That returns the driving packet that has been decided by the docking class. Driving packets will be sent down to the ROV through a multiprocessing queue via the send_data_to_rov method. All this method does is put the packet into the driving_queue in the Execution class. Information about the functional classes and how data packets are decided is shown further on, but here is an example of the Docking class:

```
   class AutonomousDocking:
    def __init__(self):
        self.driving_data = [0, 0, 0, 0, 0, 0, 0, 0]
        self.frame = None
        self.down_frame = None
        self.angle_good = False

    def run(self, front_frame, down_frame):
        self.frame = front_frame
        self.down_frame = down_frame
        self.update()
        data = self.get_driving_data()
        return data
```

Listing 7: Autonomous Docking run method.

The run method in docking takes in the two frames given by the function in the Execution class and uses it in the update method of the class. The content of this method is discussed in the Autonomous Docking section later on. This will decide how the ROV should move and send a corresponding data packet that will be returned to the Execution class which can then send it to the ROV.

Recording a video with OpenCV is similar to the method for taking pictures like in listing 4, but requires an openCV video writer to be set up. The video writer creates an empty video file that can be populated with frames using the write method in it. See:

```
class CameraManager:
    def record(self):
        if recording == False:
            self.videoResult = cv2.VideoWriter("path/to/video.avi", encoding_method,
    frame_rate_limit, dimensions)

            recording = True

        if recording == True:
            cam = camera
            frame = camera.get_frame()
            self.videoResult.write(frame)
        else:
            self.videoResult.release()
```

Listing 8: Method for recording a video

This method is used in the record method in the Execution class that runs in a loop, so it will be a continuous video. It will only set up the video Result once, because it checks if it is recording, and then put that value to true. See:

```
class Execution:
    def record_video():
        self.CameraManager.recording = False
        while not done:
            self.CameraManager.record()
        else:
            self.CameraManager.recording = False
```

Listing 9: Activating recording, Execution class

The recording is then stopped whenever we enter manual mode, or the record method in the Execution class is run again as the first thing it does is set the recording flag to false.

Similarly to the autonomous docking method, the classes for the other tasks also have run methods, allowing the number of parameters required to be kept to a minimum and making the program more modular.

6.3 Autonomous Docking

This section presents our solution for Autonomous Docking. We solved the autonomous docking task by using computer vision and motion regulation.

See listing 10 for a pseudocode representation of the whole AutonomousDocking program.

```
class AutonomousDocking:
    def __init__(self):
        self.driving_data = [0, 0, 0, 0, 0, 0, 0, 0]
        self.frame = None
        self.down_frame = None
        self.draw_grouts = False
        self.draw_grout_boxes = True
        self.angle_good = False

    def run(self, front_frame, down_frame)

    def update(self)

    def get_driving_data(self)

    def find_red(self)

    def regulate_position(self, displacement_x, displacement_y)

    def find_grouts(self)

    def find_relative_angle(self)

    def rotation_commands(self)
```

Listing 10: Pseudocode representation of the AutonomousDocking class. Initialized variables and methods are included.

These are the steps the program goes through during the docking process:

1. Get frames from the camera feed.
2. Check the ROV's alignment relative to the docking station and rotate it if needed.
3. Check the ROV's position relative to the red center point (in the docking station) and readjust if needed.
4. Send driving commands down to the ROV.

6.3.1 Step 1. Get frame

The first step in the docking process is to get the needed frames from the camera feed. For this particular task, we need two frames. A frame from one of the forward-facing cameras, and a frame from the downwards-facing camera. The frame from the front will be used to detect the red center point, see section 6.3.3 for this process. The down frame will be used to check the ROV's alignment relative to the docking station, see section 6.3.2 for this process.

Frames are fed from the Camera class using the Execution Class docking method, see listing 11.

```
def docking(self):
        self.done = False
        self.Camera.start_stereo_cam_L()
        self.Camera.start_down_cam()
        while not self.done and self.manual_flag.value == 0:
            self.update_stereo_L()
            self.update_down()
```

```
8                docking_frame, frame_under, driving_data_packet =
9                self.Docking.run(self.frame_stereoL, self.frame_down)
10               self.show(docking_frame, "Docking")
11               self.show(frame_under, "Frame Under")
12               self.send_data_to_rov(driving_data_packet)
13               QApplication.processEvents()
14               # self.show(frame_under, "Frame Under")
15           else:
16               self.stop_everything()
```

Listing 11: Method for handling autonomous docking within the Execution class.

Listing 11 shows how the Camera class is used to feed frames into the AutonomousDocking class, through the docking() method from the Execution class. See figure 32 for a rough flowchart of the dependencies.

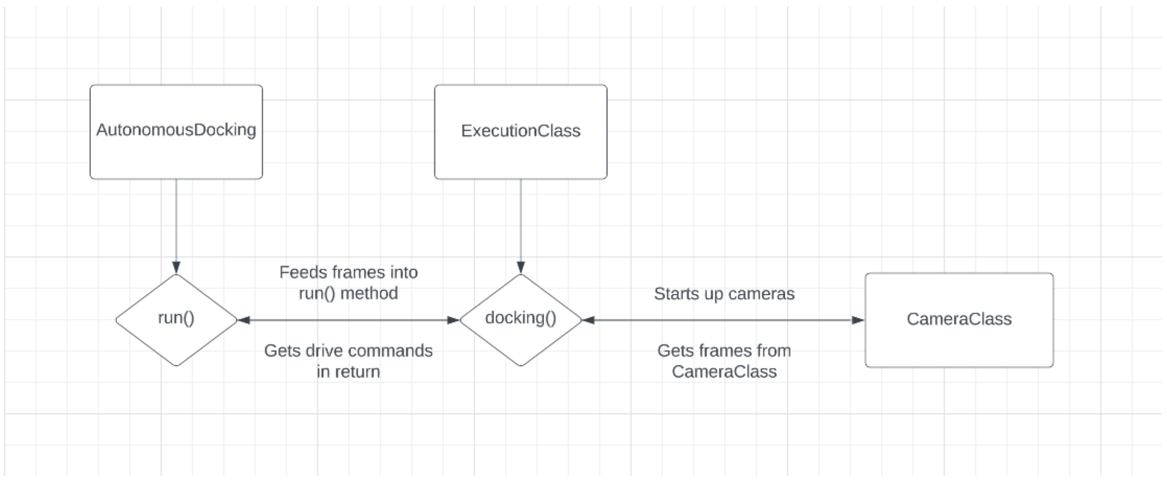

AutonomousDocking
ExecutionClass
run()
Feeds frames into run() method
Gets drive commands in return
docking()
Starts up cameras
Gets frames from CameraClass
CameraClass

Figure 32: ExecutionClass being the main-class starting cameras and getting frames. Feeding the frames into the AutonomousDocking run() method and getting drive commands in return.

As shown in listing 11, in lines 3 and 4, the left stereo camera and the down camera are started, there is no particular reason for why the left stereo camera is used instead of the right one. In lines 6 and 7 the frames from the camera-feeds are updated. These frames are then used as input parameters for the AutonomousDocking.run() method. See listing 12 for the run method.

```
class AutonomousDocking:

    def run(self, front_frame, down_frame):
            self.frame = front_frame
            self.down_frame = down_frame
            self.update()
            data = self.get_driving_data()
            return self.frame, self.down_frame, data
```

Listing 12: Main method for running the AutonomousDocking class.

The AutonomousDocking.run() method follows the steps listed in 6.3. The next steps explain how computer vision is used to observe the surroundings of the ROV and how it makes decisions based on what it observes.

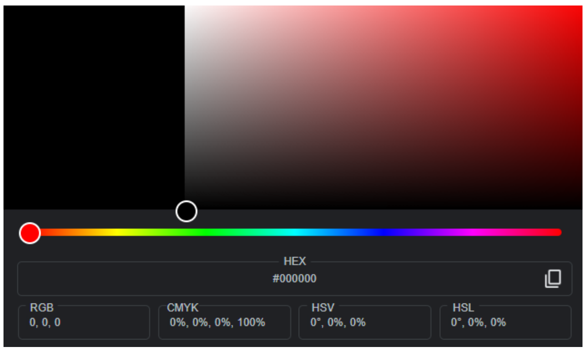
6.3.2 Step 2. Check alignment relative to docking station and rotate

After retrieving frames from the camera feed, the program has to check its alignment relative to the docking station, see more detailed explanation in section 8.3.4. This is done by looking at the grouts on the seafloor and calculating the relative angle between the direction of the grouts and the ROV. To do so, computer vision techniques are used to detect the grouts. The find_grouts() and find_relative_angle() methods within the AutonomousDocking class are responsible for handling this task.

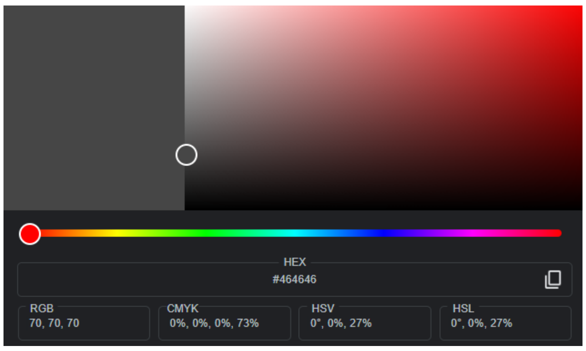
```
1 def find_grouts(self):
2         lower_bound, upper_bound = (0, 0, 0), (70, 70, 70)
3         grouts = cv2.inRange(self.down_frame, lower_bound, upper_bound)
4         grouts_dilated = cv2.dilate(grouts, None, iterations=10)
5         canny = cv2.Canny(grouts_dilated, 100, 200)
6         blurred = cv2.GaussianBlur(canny, (11, 13), 0)
7         grout_contours, _ = cv2.findContours(blurred, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
8 
9         return grout_contours
```

Listing 13: Computer vision method to find grout contours using OpenCV.

Listing 13 is the method used to find grouts. In line 3, the cv2.inRange() function creates a mask where only colors within said range are shown, essentially isolating the grouts from the rest of the image. The lower_bound tuple is the BGR value of pure black and the upper_bound tuple is the BGR value of a slightly dark shade of gray. See figure 33 for a visual representation of the color range.


HEX #000000
RGB 0, 0, 0
CMYK 0%, 0%, 0%, 100%
HSV 0°, 0%, 0%
HSL 0°, 0%, 0%

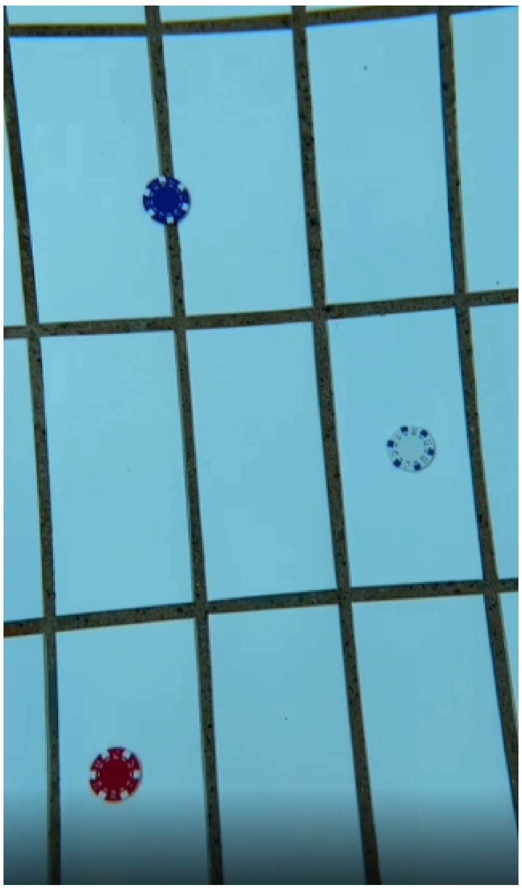
(a) Lower_bound = (0, 0, 0) = pure black.


HEX #464646
RGB 70, 70, 70
CMYK 0%, 0%, 0%, 73%
HSV 0°, 0%, 27%
HSL 0°, 0%, 27%

(b) Upper_bound = (70, 70, 70) = dark gray.

Figure 33: The cv2.inRange() method creates a mask ranging between these two RGB values.

Lines 4-6 in listing 13 are filtration techniques applied to the grouts mask, to remove noise and impurities in the frame, making it easier and cleaner to find contours. The find_grouts() method returns grout_contours, which is a list of all the contours it was able to detect. This contours list is received by the find_relative_angle() method. See figure 34 for a visual representation of what the contours look like.


(a) Frame of poolfloor used in find_grouts().

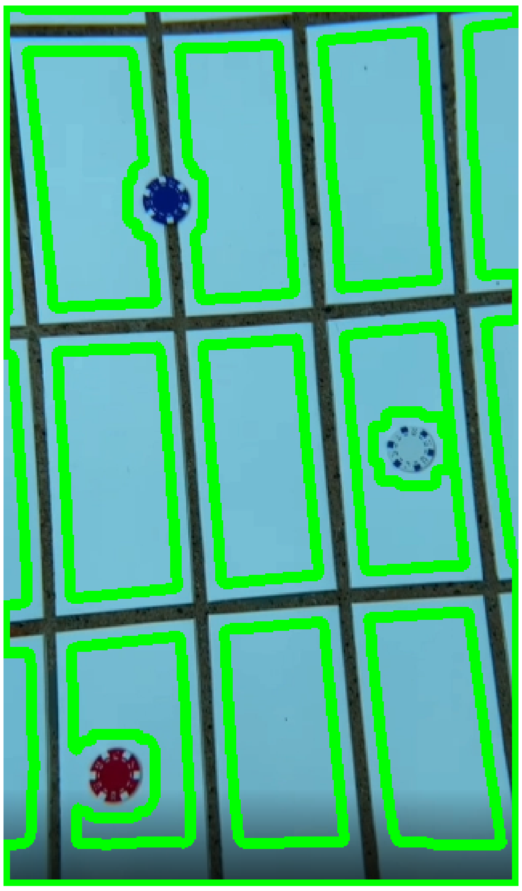

(b) Output of find_grouts() method. Contours are outlined in green.

Figure 34: (a) is input of find_grouts() method. (b) is visual representation of output of find_grouts() method.

The grout_contours list returned from find_grouts() is then used by the find_relative_angle() method. This method essentially loops through a list of grout_contours, filtering out inappropriate contours and calculating the average angle of all the appropriate contours. Here is a snippet of the most important part of the find_relative_angle() method 14.

```
for contour in contours:
        rect = cv2.minAreaRect(contour)
        (x, y), (w, h), angle = rect
```

Listing 14: Loops through the grouts list. Fits a rectangle on top of the contour.

In listing 14, (x, y) are the pixel coordinate of the top left corner of the rectangle. (w, h) are the width and height of the rectangle. angle describes how much the rectangle is rotated relative to the vertical. With these values: (x, y), (w, h), and angle, the program can filter out inappropriate contours, such as very small contours/

spots that may come with noise in the frame, as well as misformed contours and or other unwanted contours. We filter the contours based on their total area relative to the whole frame. See listing 15 for code snippet of the filtering process. See figure 35 for the filtered contours.

```
MAX_AREA = (frame_height * frame_width) * 0.30
MIN_AREA = (frame_height * frame_width) * 0.05
if (area > MAX_AREA) or (area < MIN_AREA):
    continue
```

Listing 15: Codesnippet for filtering contours.

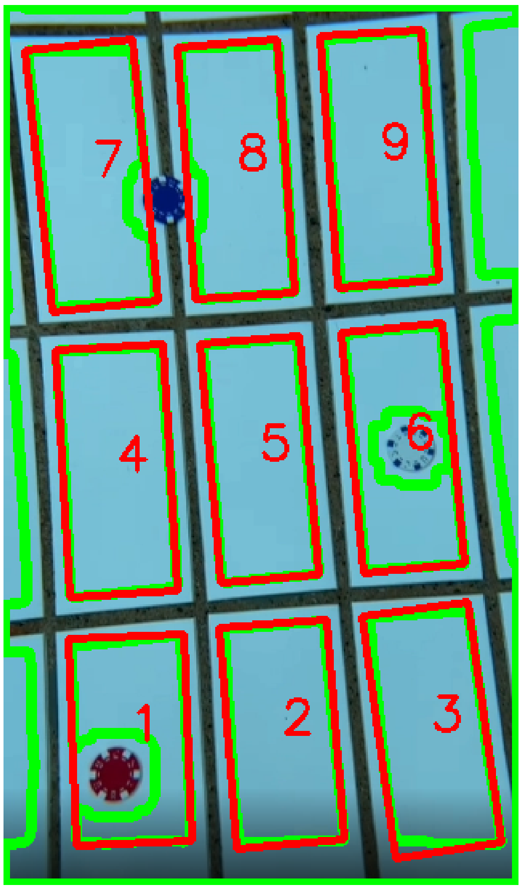


Figure 35: Visual represenatation of the filtered grout contours. The filtered contours are outlined in red.

Using all the filtered grout contours, the average angle can be calculated. This serves as the estimated angle of the ROV relative to the pool floor (relative to the docking station). The find_relative_angle() method returns the average angle of all the grout contours as a float. With this information, the program can evaluate whether it needs to rotate, relative to the docking station, or look for a red center point and adjust its position.

The program allows for some margin of error between the angle of the ROV relative to the docking station (pool-floor grouts). If the ROV is within -2 and 2 degrees, it does not rotate. This prevents the program from getting stuck if it never reaches a perfect angle. If the angle is outside of said range it will update the **self.driving_data** variable within the class with a data packet containing a rotation value.

The magnitude of rotation sent in the packet depends on how rotated the ROV is. If it is rotated 30 degrees, it sends a data packet with a rotation value of 30. It is a low value but we want the ROV to maneuver slowly to prevent drift or rotate past its wanted alignment position. This is what the packet would look like:

```python
#driving data packet: [x, y, z, r, 0, 0, 0, 0]
self.driving_data = [0, 0, 0, 30, 0, 0, 0, 0]
```

Listing 16: Values in the packets are: x value is forward axis. the y value is left to the right axis. The z value is up and down with respect to underwater depth. r value is the value of rotation on the horizontal plane. The values in the data packet range between -100 and 100.

6.3.3 Step 3. Check position relative to the center point and reposition

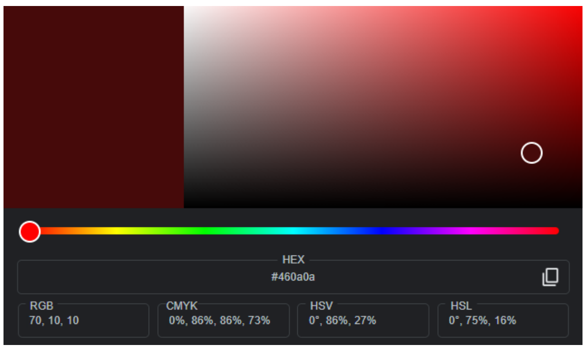
After the ROV has rotated into correct alignment, the program will look for the red centerpoint in the dockingstation. This is done with the **find_red()** method in the autonomous docking class.

The **find_red()** method finds the red CenterPoint in the frame. Since the red center point is a PVC-pipe cap, it is circular. Therefore the program returns the center pixel coordinate and the radius of the red cap. See figure 7 for an illustration of the docking station. See listing 17 for code to isolate and find red. The **find_red()** method also uses techniques to remove noise for better isolation of the red. It uses both blurring and dilation.

```python
1 def find_red(self):
2         lower_bound, upper_bound = (10, 10, 70), (70, 40, 255)
3         blurred = cv2.GaussianBlur(self.frame, (11, 13), 0)
4         mask1 = cv2.inRange(blurred, lower_bound, upper_bound)
5         dilated = cv2.dilate(mask1, None, iterations=6)
6         red_isolated = cv2.bitwise_and(self.frame, self.frame, mask=dilated)
```

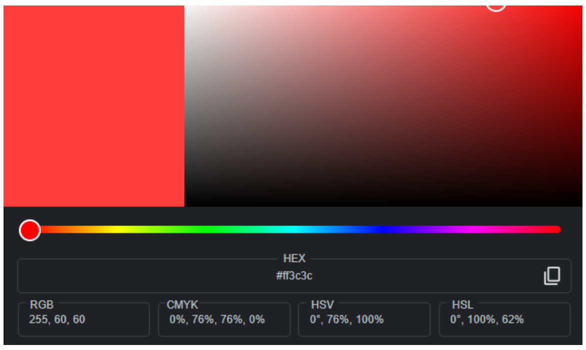
Listing 17: Codesnippet from the method used to find the red centerpoint in the docking station.

In Listing 17, line 4, the program uses the cv2.inRange() function to find all red pixels in the frame. We give it an upper and lower range to find the red between. The lower_bound tuple is the BGR value for a dark red color while the upper_bound tuple is the BGR value for a lighter brighter red color. In other words, the program is looking for a pixel with low blue, green values, and a high red value. Combined this can be many different shades of red, making the program more flexible. See figure 36 for visuals of the two RGB values.



HEX #460a0a
RGB 70, 10, 10
CMYK 0%, 86%, 86%, 73%
HSV 0°, 86%, 27%
HSL 0°, 75%, 16%

(a) Lower_bound = (10, 10, 70) BGR = dark red.



HEX #ff3c3c
RGB 255, 60, 60
CMYK 0%, 76%, 76%, 0%
HSV 0°, 76%, 100%
HSL 0°, 100%, 62%

(b) Upper_bound = (60, 60, 255) BGR = bright red.

Figure 36: The cv2.inRange() method creates a mask ranging between these two RGB values.

After the red color is isolated, the program finds contours. Then it loops through the contours list, picking the largest contour in size as the one representing the center point in the docking station.

See listing 18 for the rest of the code in the find_red() method.

```
1  canny = cv2.Canny(red_isolated, 100, 200)
2  contours, _ = cv2.findContours(canny, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
3  red_center = (0, 0), 0
4  for c in contours:
5      (x, y), radius = cv2.minEnclosingCircle(c)
6      if radius > red_center[1]:
7          red_center = (x, y), radius
8  center_point = (int(red_center[0][0]), int(red_center[0][1]))
9  radius = int(red_center[1])
10 cv2.circle(frame, center_point, radius, (0, 255, 0), 2)
11 return center_point, radius
```

Listing 18: The rest of the code in the find_red() method. Returns the center point and radius of the largest contour.

In listing 18, in line 5, the program uses cv2.minEnclosingCircle() to fit as small of a circle as possible but still enclose the contour. This function returns the center pixel coordinates and the radius of the circle. Since the red center point is also circular, the minimum enclosing circle is a good fit. See figure 37 for a visual representation of the return values from the find_red() method.

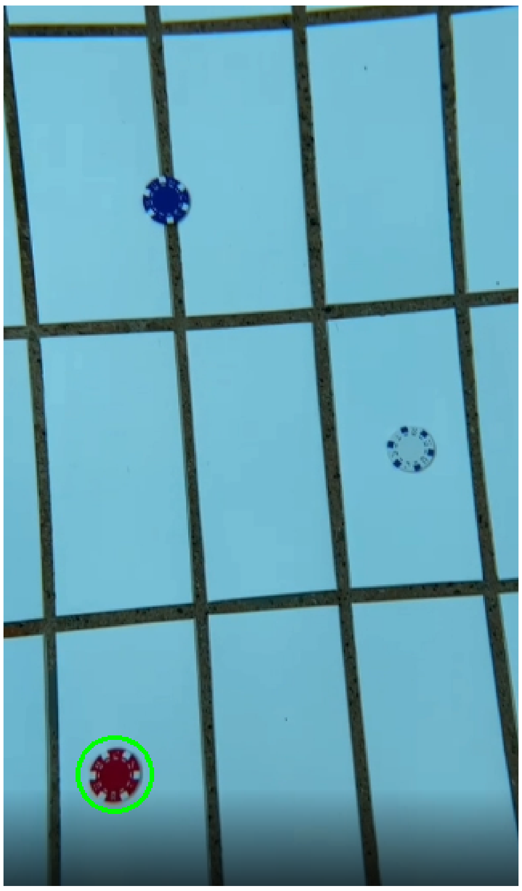


Figure 37: Visual representation of the return values of the find_red() method drawn as a green circle.

After the program finds the center pixel coordinate and radius of the red circle, the program calculates the percent difference in position relative to the center of the frame. The difference in y and z pixel position is used by the **self.regulate_position()** method. This method updates self.driving_data based on the displacement of the ROV relative to the red center-point in percent. See figure 38 for the calculation of the center difference. See listing 19 for the code for regulate_position().

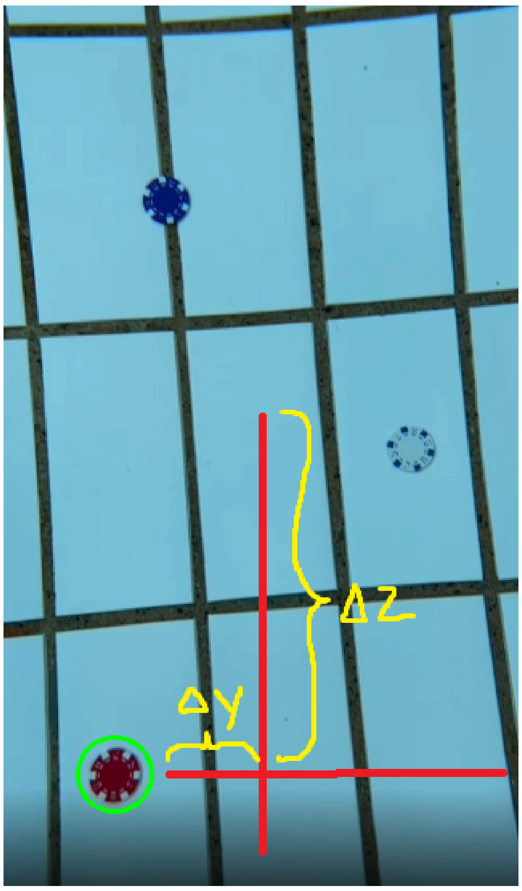


Figure 38: The red center point is in the y, z plane. The ROV needs to move in the direction of delta y and delta z in order to have the center point in front of itself. Delta y is the difference between the center of red and the center of frame on the y-axis. Delta z is the difference in center of red and center of frame on the z-axis.

```python
def regulate_position(self, displacement_y, displacement_z):
    if displacement_y > 2:
        #drive_command = "GO LEFT"
        self.driving_data = [0, displacement_y, 0, 0, 0, 0, 0, 0]

    elif displacement_y < -2:
        #drive_command = "GO RIGHT"
        self.driving_data = [0, displacement_y, 0, 0, 0, 0, 0, 0]

    elif displacement_z > 2:
        #drive_command = "GO DOWN"
        self.driving_data = [displacement_z, 0, 0, 0, 0, 0, 0, 0]

    elif displacement_z < -2:
        #drive_command = "GO UP"
        self.driving_data = [displacement_z, 0, 0, 0, 0, 0, 0, 0]
    else:
        # drive_command = "GO FORWARD"
        self.driving_data = [10, 0, 0, 0, 0, 0, 0, 0]
```

Listing 19: Code for the method that changes position of the ROV to get the red centerpoint in front of the ROV.

The **self.regulate_position()** method first positions the ROV correctly on the y-axis. After it positions the ROV correctly on the z-axis. It updates the self.driving_data with thrust equal to the percent displacement of the ROV. This way the ROV moves slower and slower towards the correct position, preventing it from drifting past the correct position. If the ROV is in the correct position, meaning that the red center point is in front of the ROV. The program will update self.driving_data to move the ROV forward, in the positive direction on the x-axis. This forward motion remains constant at a slow pace to prevent the ROV from drifting and giving the program time to check all the alignments and positions.

6.3.4 Step 4. Send driving commands

Sending the driving commands down to the ROV is done by the execution class. The execution class runs a while loop constantly calling the AutonomousDocking.run() method which returns self.frame, self.down_frame, and self.driving_data. The execution class has a multiprocessing queue that is shared with the main file running all the topside code for the ROV, **main.py**. The execution class puts driving data into the queue in every iteration of the while loop running the AutonomousDocking.run() method. From main.py, the **send_data_to_ROV()** method, grabs data packets from the queue and sends them down to the ROV.

We tested this multiple times on land during dry tests of the electrical system, and it worked as intended. The program got frames from the camerafeed, processed them, sent the correct driving data down to the ROV and the ROV's thrusters spun.

6.4 Autonomous Transect Maneuver and Frog Count

This section presents our solution for driving the ROV autonomously across a transect and counting frogs beneath. One of the tasks from MATE is to maneuver the ROV along a transect (a channel between two dark blue PVC pipes) and count the number of frogs in the transect using image processing and computer vision. See Figure 39.

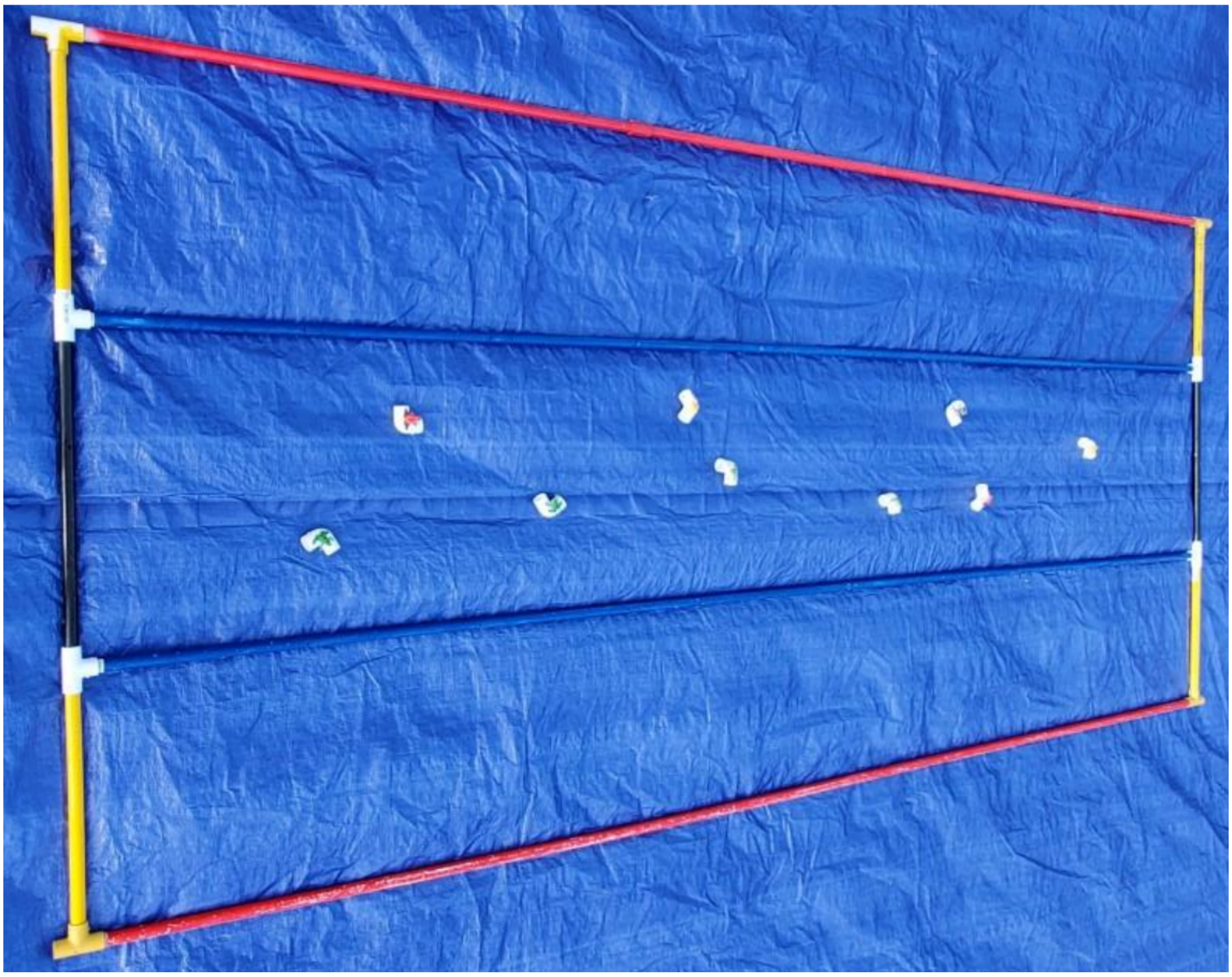


Figure 39: Image of frogs randomly spread out within the transect area.

6.4.1 Counting Frogs

For the task of counting frogs, the ROV uses two methods. One which detects all frogs except for red ones, shown in listing 20, and another which detects all frogs except for dark ones, shown in listing 21. These functions use similar processes for detecting frogs. They find two different thresholds, one which highlights only the grouts, and another which highlights both the grouts and the frogs. The programs then uses the subtract() function from OpenCV to highlight only the frogs.

```
def frogDetectionNoRed(self, image): # Does not detect red frogs
    thresh1 = cv2.threshold(image[:,:,0], 70, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C)[1]
    thresh2 = cv2.threshold(image[:,:,0], 200, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C)[1]
    difference = cv2.subtract(thresh2, thresh1)

    blur_difference = cv2.GaussianBlur(difference, (41, 41), 0)
    new_thresh = cv2.threshold(blur_difference, 0, 255, cv2.THRESH_OTSU)[1]

    contours, _ = cv2.findContours(new_thresh.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    frogRectangles = self.contourFiltration(contours)
    return frogRectangles
```

Listing 20: Frog detection method. Does not detect red frogs.

While the function which detects red frogs looks at the redscale version of the frame, the other function relies on a grayscale image instead.

```
def frogDetectionNoGrout(self, image): # Does not detect frogs in dark tile grouts or frogs which match the pool floor color
    hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS_FULL)
    gray = cv2.cvtColor(hls, cv2.COLOR_RGB2GRAY)

    thresh1 = cv2.threshold(gray, 90, 255, cv2.THRESH_OTSU)[1]
    thresh2 = cv2.threshold(gray, 120, 255, cv2.THRESH_TRIANGLE)[1]
    difference = cv2.subtract(thresh1, thresh2)

    diff_blur = cv2.GaussianBlur(difference, (71, 71), 0)
    dilate_blur = cv2.dilate(diff_blur, None, iterations=6)
    newThreshold = cv2.threshold(dilate_blur, 0, 255, cv2.THRESH_OTSU)[1]

    contours, _ = cv2.findContours(newThreshold.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE
)
    frogRectangles = self.contourFiltration(contours)
    return frogRectangles
```

Listing 21: Frog detections methods

For a visual representation of how the frog detection methods work, see figures 40 and 41

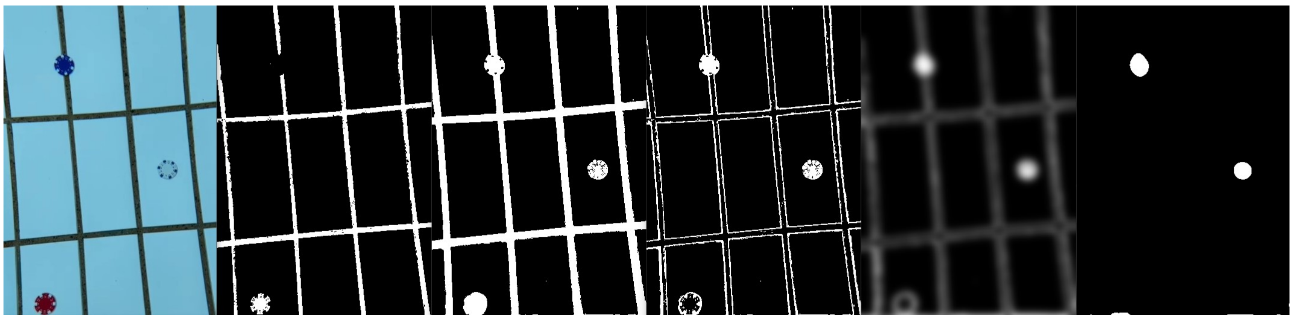


Figure 40: Result of function which does not detect red frogs. Images from left to right represent: original, threshold1, threshold2, difference, blur, final threshold

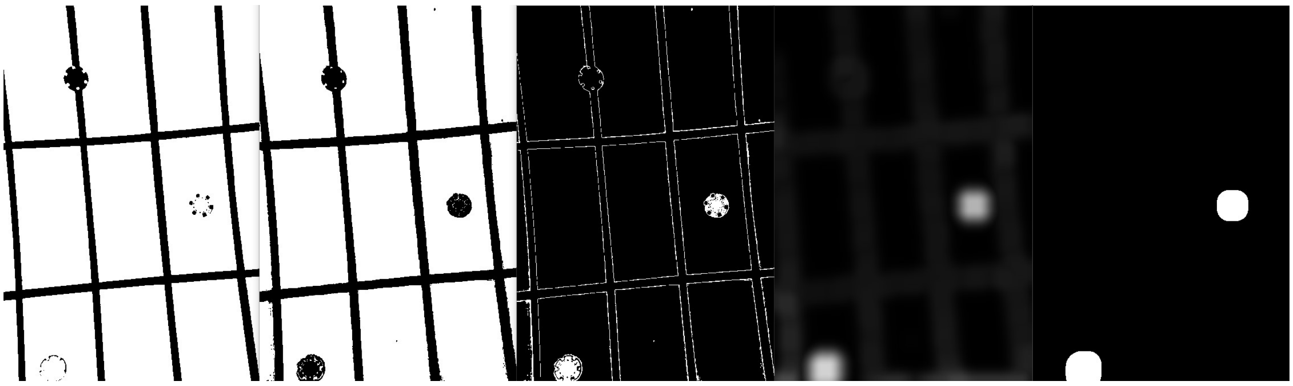


Figure 41: Result of function which does not detect dark frogs. Images from left to right represent: threshold1, threshold2, difference, blur + dilate, final threshold

While none of the functions manage to detect all three coins (frogs) themselves, together they detect them all.

After finding the contours of the images produced by the functions above, the method in listing 22 was used to filter out noise and faulty contours. It does this by removing contours that are too small, too large, or too misformed. Which contours are misformed is determined by creating a polygonal approximation of the contour using OpenCV's approxPolyDP(). The contour is removed if the polygon has too many or too few sides. A rectangle is also fitted above the contour using OpenCV's boundingRect(), if this rectangle is more rectangular than square, the underlying contour is removed. The idea is that frogs should appear more square than rectangular to the camera.

```python
def contourFiltration(self, contours, epsilonValue = 0.03, noise_threshhold_lower = 40,
    noise_threshhold_upper = 300):
    frog_rectangles = []
    for contour in contours:
        epsilon = epsilonValue * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        if 10 > len(approx) > 4:
            x, y, w, h = cv2.boundingRect(approx) # Rectangle, rotated
            if 0.7 < w/h < 1.3:
                if  w > noise_threshhold_lower < h and w < noise_threshhold_upper > h:
                    frog_rectangles.append((x,y,w,h))
                    cv2.rectangle(self.frame, (x, y), (x+w, y+h), (255, 0, 0), 3)
    return frog_rectangles
```

Listing 22: Contour filtration method

To make sure frogs (rectangles) detected by the methods in listing 20 and 21 are not counted twice, the method in listing 23 is used to remove overlapping rectangles. This method is also used later to check which frogs overlap with the image from the previous frame of the video stream. This is done by checking if either rectangle is to the left of the other one, or if either rectangle is above the other one. If either of those is true the rectangles do not overlap, if not they do overlap. This method was inspired by an online article, see citation [9].

```python
def rectangleOverlapFilter(self, rectangles): # Filters out rectangles that overlap
    overlappedRectsPairs = []
    if len(rectangles) >= 2:
        n = 0
        for rectangle in rectangles:
            n += 1
            for index in range(n, len(rectangles)):
                x,y,w,h = rectangle
                x2,y2,w2,h2 = rectangles[index]

                if x + w < x2 or x2 + w2 < x:
                    continue

                elif y + h < y2 or y2 + h2 < y:
                    continue

                else:
                    rectangles.pop(index)
                    break

    return rectangles
```

Listing 23: Method for finding overlapping rectangles

The program also needs to keep track of frogs across several frames. The code in listing 25 does this. It gives each frog a value called detectionCounter, which tracks how many frames in a row a frog is counted. When a new frame is analyzed, it is run through an altered version of the rectangle overlap filter function, shown in listing 24. Instead of removing overlapping rectangles it instead returns a list of tuples containing indexes of overlapping rectangles. The program then goes through each tuple, containing the index of both old and new rectangles, and gives the new rectangle the detectionCounter value of the old one, incremented by 1. If a frog passes a detection counter threshold, it is counted as a frog and the frog detection counter is incremented.

```python
    def rectangleOverlapFilterNoRemove(self, rectangles):
    overlappedRectsPairs = []
    if len(rectangles) >= 2:
        n = 0
        for rectangle in rectangles:
            n += 1
            for index in range(n, len(rectangles)):
                x,y,w,h = rectangle
                x2,y2,w2,h2 = rectangles[index]

                if x + w < x2 or x2 + w2 < x:
                    continue

                elif y + h < y2 or y2 + h2 < y:
                    continue

                else:
                    overlappedRectsPairs.append((n, index))
                    break

    return overlappedRectsPairs
```

Listing 24: Method for finding overlapping rectangles

```
def checkPreviousFrogsAdvanced(self):
    allFrogs = self.currFrogs + self.prevFrogs
    theRectangles = []
    for frog in allFrogs:
        theRectangles.append(frog.rectangle)
    OverlapPairs = self.rectangleOverlapFilterNoRemove(theRectangles.copy())
    for OverlapPair in OverlapPairs:
        # OverlapPair = touple of two indexes, of rectangles that overlap
        if allFrogs[OverlapPair[0]].detectionCounter <= allFrogs[OverlapPair[1]].
    detectionCounter:

            allFrogs[OverlapPair[0]].detectionCounter = allFrogs[OverlapPair[1]].
    detectionCounter + 1

    self.prevFrogs = self.currFrogs
    for frog in self.prevFrogs:
        if frog.detectionCounter == self.detection_counter_threshold:
            self.frog_counter += 1
    self.currFrogs = []
```

Listing 25: Method for tracking frogs across several frames

Lastly, the program has a function that starts all the previous processes and puts them together, see listing 26.

```
    def update(self, image, drawImage = False):
        self.frame = image

        # Calling the frog detection functions
        rectanglesNoRed = self.frogDetectionNoRed(image)
        rectanglesNoGrout = self.frogDetectionNoGrout(image)
        allRectangles = rectanglesNoRed + rectanglesNoGrout
        filteredRectangles = self.rectangleOverlapFilter(allRectangles)

        if drawImage:
            for rect in filteredRectangles:
                x,y,w,h = rect
                cv2.rectangle(image, (x,y), (x+w, y+h), (0, 255, 0), 2)
            cv2.imshow("Image", image)

        # Converts the rectangles into Frog objects, used for Advanced frog counting algorithm
        frogRectangles = []
        for rectangle in filteredRectangles:
            frogRectangles.append(Frog(rectangle))

        self.currFrogs = frogRectangles

        if self.check_previous_algo == "BASIC":
            self.checkPreviousFrogsBasic()

        if self.check_previous_algo == "ADVANCED":
            self.checkPreviousFrogsAdvanced()

        return self.frog_counter
```

Listing 26: Method for starting other methods and connecting them correctly

6.4.2 Transect Maneuver

In addition to counting frogs, the ROV must also maneuver over a designated transect. To do this, the ROV analyzes and corrects both its position in the horizontal plane and its angle. The angle is calculated and adjusted by first locating the blue pipes at the bottom of the pool. The pipes are found by utilizing OpenCV's inRange() function to find dark blue contours, see listing 27

```
def find_dark_blue_contours(self):
    low_blue_range = (0, 0, 0) #b, g, r
    high_blue_range = (255, 60, 60)

    transect_pipe_mask = cv2.inRange(self.frame, low_blue_range, high_blue_range)
    pipe_contours, _ = cv2.findContours(transect_pipe_mask, cv2.RETR_TREE, cv2.
    CHAIN_APPROX_SIMPLE)

    return pipe_contours
```

Listing 27: Method for finding dark blue contours. Uses frame in class and returns contours.

After getting the dark blue contours, the pipes are located using the code in listing 28. The pipes are isolated by fitting a rectangle around the contours and only using contours that pass several requirements. The fitted rectangle must not be too square, the area of the contour must be above a threshold, and either the height or width of the fitted rectangle should go across the entire width or height of the frame.

```
def find_pipes(self):
```

```python
    contours = self.find_dark_blue_contours()
    pipes = []

    for contour in contours:
        rect = cv2.minAreaRect(contour)
        (x, y), (w, h), angle = rect

        box = cv2.boxPoints(rect)
        box = np.intp(box)
        if w > 1 and h > 1:
            if (w / h < 0.25 or w / h > 2) and cv2.contourArea(contour) > 300 and (w > self.frame.shape[0] - 200 or h > self.frame.shape[0] - 200 or w > self.frame.shape[1] - 200 or h > self.frame.shape[1] - 200):
                straightRect = cv2.boundingRect(contour)
                x, y, w, h = straightRect
                cv2.rectangle(self.frame, (x, y), (x + w, y + h), (255, 100, 0), 2)
                cv2.drawContours(self.frame, [box], 0, (0, 0, 255), 3)

                pipes.append(rect)
    if len(pipes) == 2:
        return pipes

    else:
        # Faulty number of pipes
        return "SKIP"
```

Listing 28: Method for locating blue pipes at bottom of pool. Uses frame in class and returns the pipes as OpenCV rectangles.

After finding the pipes, the ROV calculates the angle of the pipes and adjusts accordingly if needed. See listing 29

```python
def stabilize_angle(self):

    pipes = self.find_pipes()
    if pipes == "SKIP":
        return

    transect_angle = self.get_angle_between_pipes(pipes[0], pipes[1])

    if transect_angle < -2:
        # print("Turn left")
        self.driving_data = [0, 0, 0, -10, 0, 0, 0, 0]

    elif transect_angle > 2:
        self.driving_data = [0, 0, 0, 10, 0, 0, 0, 0]
        # print("Turn right")

    else:
        self.canStabilize = True
```

Listing 29: Method which calculates angle of pipes and adjusts ROV

After correcting the angle, the ROV will try to correct its alignment between the pipes. This code only runs if the angle is seen as good by the ROV, and the canStabilize boolean is set to True. To correct alignment the ROV first finds the blue pipes as it did when stabilizing its angle. Afterward, the distance from each pipe to the edge of the frame is calculated, and if the difference in distance is too high the ROV will move to correct itself. The code for this can be seen in listing 30

```python
def stabilize_alignment(self):
    if self.canStabilize:
        pipes = self.find_pipes()
        if pipes == "SKIP":
            print("SKIPPING FRAME")
            return

        # Find leftmost pipe:
        if pipes[0][0] < pipes[1][0]:
            leftPipe = pipes[0]
            rightPipe = pipes[1]
        else:
            leftPipe = pipes[1]
            rightPipe = pipes[0]

        distanceFromLeftPipe = leftPipe[0][0]
        distanceFromRightPipe = self.frame.shape[1] - rightPipe[0][0]
        ratio = distanceFromLeftPipe / distanceFromRightPipe # ratio = 1 means perfect

        if 0.95 > ratio:
            self.driving_data = [-10, 0, 0, 0, 0, 0, 0, 0]
            # Move to left

        elif 1.05 < ratio:
            # Move to right
            self.driving_data = [10, 0, 0, 0, 0, 0, 0, 0]

        else:
            # Go forward
            self.driving_data = [0, 10, 0, 0, 0, 0, 0, 0]
        self.canStabilize = False
        return
    else:
        # Angle bad, can't stabilize yet
        return
```

Listing 30: Method which corrects alignment between the blue pipes.

6.5 Sea grass Monitoring

To analyze the amount of growth/decay in seagrass OpenCV is used to detect the amount of green squares. This is done by the code in listing 31, which isolates the green squares by using a binary thresholding method, coupled with contour filtration. The contour filtration uses OpenCV's approxPolyDP to fit a polygon around the contour. If the polygon does not have 4 sides it is not counted, as it is not a square. Next the code creates a rectangle over the contour using OpenCV's boundingRect(). If the relationship between the width and height of the rectangle is not close to 1, the rectangle is more rectangular than square and is not counted as a result.

```python
def detect_squares(self, frame):
    squares = 0

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (5, 5), 0)
    dilated = cv2.dilate(blur, None, iterations=3)
    _, thresh = cv2.threshold(dilated, 127, 255, cv2.THRESH_BINARY)

    contours, _ = cv2.findContours(thresh, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

    # Contour filtering
    for contour in contours:
        epsilon = 0.03*cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        cv2.drawContours(dilated, [approx], 0, (0), 3)

        if len(approx) == 4: # 4 sides means a square
            i, j = approx[0][0]
            # x,y top left corner. w,h width and height
            x, y, w, h = cv2.boundingRect(contour)
            ratio = float(w)/h
            # how long a square side needs to be in order to be counted, to remove noise
            noise_threshhold = 20
            # ratio between 0.9 and 1.1 means a square
            if  0.9 <= ratio <= 1.1 and w > noise_threshhold < h:
                squares += 1
    return squares
```

Listing 31: Logic for detecting green squares

Result of the function can be seen in figures 42 and 43.

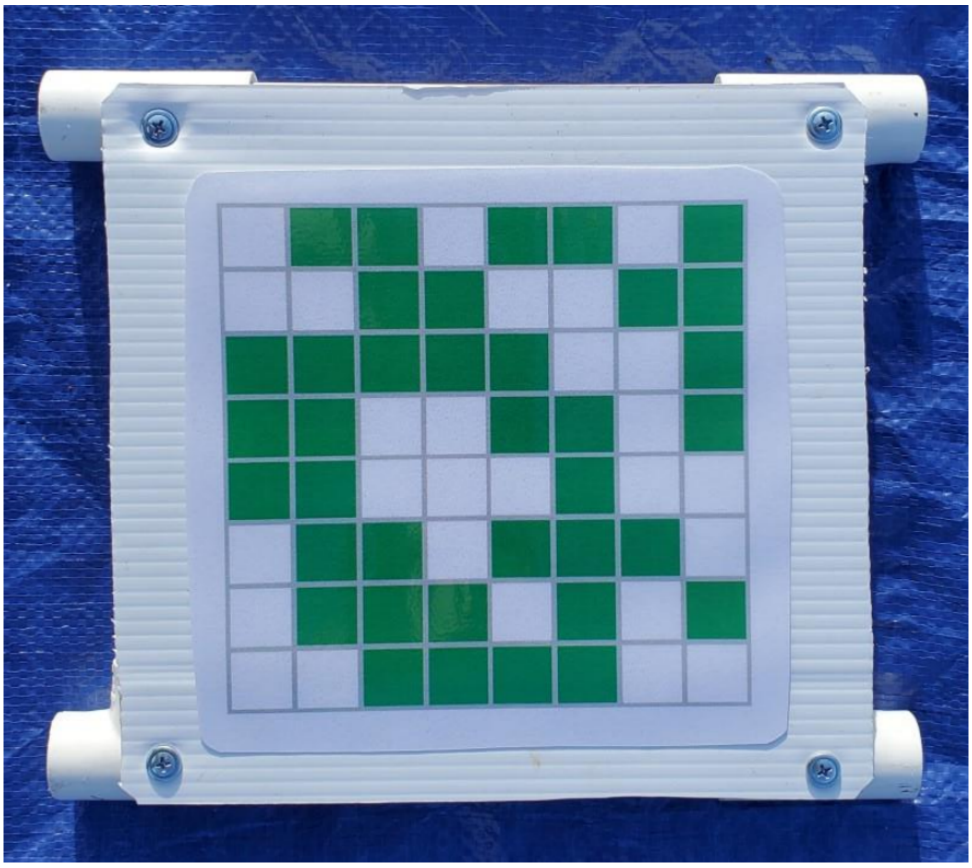


Figure 42: Example image of simulated sea grass.

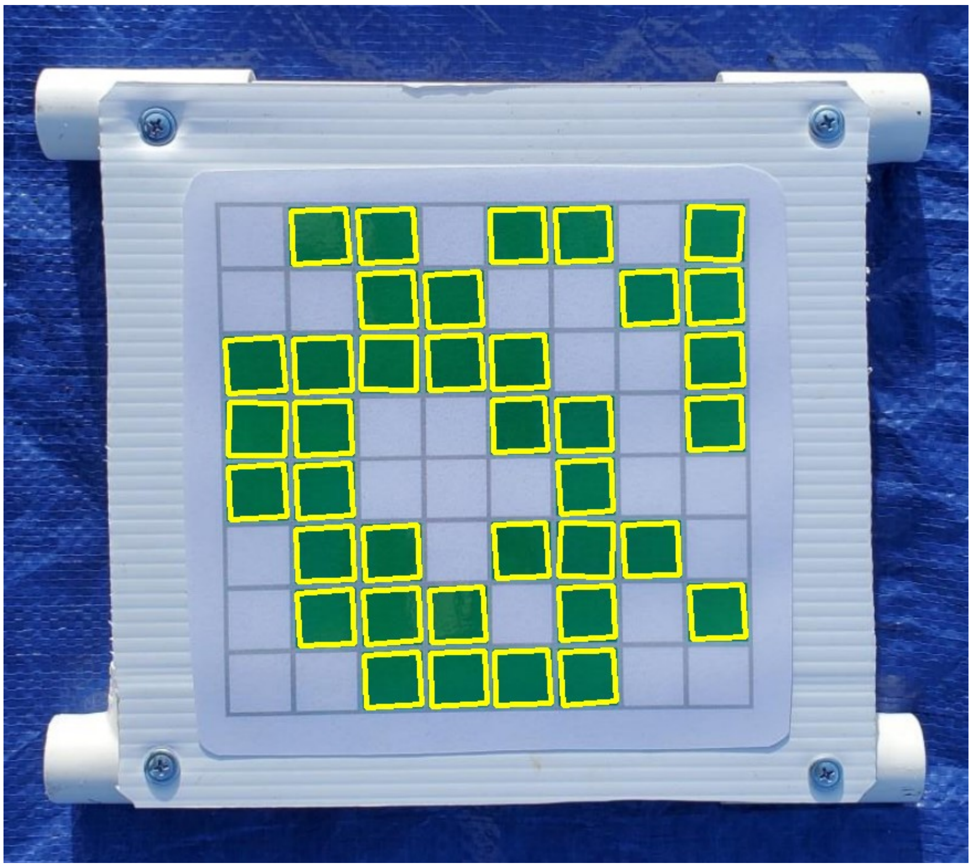


Figure 43: Visualization of counted squares

Lastly the percentage change in seagrass is calculated, shown in listing 32

```
def calculate_seagrass(self, squares_before, squares_after):
    percentage_difference = (squares_after / squares_before) * 100
    return grass_difference
```

Listing 32: Code for calculating percentage difference in seagrass.

7 Economical Overview & Environmental Impact

This section presents our group's budget and what money we spent during the project. Additionally, the economical requirements for the topside system.

7.1 Economical Overview

In Subsea, each group is given a financial budget to cover the cost of production and components. As our group is focused on developing software for the ROV, we were given a relatively low budget compared to the other groups, amounting to 2500kr. We ended up not using any of this as all the components needed by our group were provided by the Subsea organization. Still, our group had influence in a couple of decisions regarding the choice of components. These include:

- Topside system
- Sensor configuration
- Camera configuration

See table 3 for an overview of the different components that are used by our group for the topside system:

Component	Price
ASUS Mini PC PN52 BBR758HD	6 582kr
Kingston SSD 512GB	909kr
Kingston DDR4 16GB	649kr
AOC 24" Monitor	1200kr
Keyboard	299kr
Mouse	199kr
Speaker	299kr
Total	10 137kr

Table 3: Economical overview of the table

Additionally to the topside system, we were involved in configuring the sensors and camera setup. For the sensors, there was a discussion regarding having a sonar on the ROV. This was ultimately dropped, saving some money for the total budget of the ROV. Our group was also involved in the camera configuration. After meeting with the communications group, who were responsible for implementing the cameras on the ROV, we landed on the camera configuration previously mentioned in section 6.1. Though these costs are not directly tied to our group, they impacted the total budget of Subsea for the ROV.

Which cameras to buy were left largely up to us, and we decided on an IMX219-83 stereo camera for the front, as this provides high enough resolution for detailed image analysis and some advanced image analysis techniques utilizing the dual cameras. For the cameras located on the manipulator and beneath the ROV, we decided on OV9281-160 Mono Cameras. These provide lower-quality images than the stereo camera but are good enough for their respective tasks. Together with the GUI group we also decided to buy a mini-PC to run all the programs and code. This decision was made after careful research into what was needed for the GUI and Image processing. As there was little need for a powerful GPU, it was not included in the build. This helped keep the cost low.

7.2 Environmental impact

This year's MATE ROV challenge is centered around sustainability and ocean health. In order to further contribute towards this goal we have focused on sustainable options throughout the project. During development, we have focused on solutions requiring a low amount of resources, allowing us to forgo the purchasing of a graphical processing unit (GPU). This reduces energy consumption leading both to lower costs and a cleaner environment. When deciding on components to purchase, such as a Mini-PC and cameras, important factors we took into consideration were the lifetime of the product and energy efficiency. There is a balancing act between high quality, energy efficiency, and cost. We believe both the ROV components and the topside components for the final setup, were solid products that will last a long time, reducing waste, while not being overly extravagant.

8 Discussion

This section will detail how we arrived at our different solutions, what we learned in the process of developing the solutions, and what we would have done differently if we were to start over.

8.1 Abandoned features

This project deviates from our proposed bachelor's thesis on several different points, however, the nature of the project is still centered around image processing, computer vision, and autonomy. This section discusses what features were discarded and why they were discarded.

8.1.1 Sonar

In the early parts of the Subsea project, we had to come to agreements with the other groups on which features should be installed on the ROV, such as sensors, cameras, sonar, etc. This way we could order the parts early, and the groups responsible for constructing and powering the ROV could plan how to implement and join the components. After some discussions with the other groups, the installment of the sonar was de-prioritized and ultimately abandoned.

8.1.2 3D Modeling

Of the tasks we were given in the MATE competition, we considered the 3d modeling task to be the most challenging. We had originally planned to use the sonar as the driving force behind the modeling while using image processing more as an additional utility. Without the sonar, the task, unfortunately, proved too time-consuming for us, and we were unable to properly solve this in the given time. We made some progress as reflected in section 8.6, but it ultimately remains unfinished.

8.1.3 Autonomous Handling

We had originally planned to create a general autonomous driving mode. Which would make the ROV drive around on its own without crashing into anything and capturing data of the surrounding area. As this feature had no relevance to the MATE competition, and the captured data was less relevant without a sonar, this feature got placed very low on our priority list. The plan was to complete this after the actual MATE tasks had been completed. But due to other tasks, especially 3d modeling, taking up more time than originally planned this feature was eventually completely dropped.

8.1.4 Machine Learning

We also wanted to use machine learning to do some object recognition in our project. The idea was, if we had any spare time at the end of the project, we could try to recognize different objects using object recognition with machine learning. There are two reasons why we didn't get into machine learning. The first reason is that we ran out of time. The second is that in order to train an AI, a large dataset is needed. For a consistent and solid result upwards of 10-20thousand images would be needed [37]. Since we did not have a completed ROV until the last week, we were unable to gather a large dataset.

8.2 Development methodology

Overall, using concepts and principles from TDD and DDD had a positive impact on the project. Resulting in a clear workspace and little to no bugs. Here is a simple example (one of many) where both TDD and DDD had a significant impact:

During the development of autonomous docking, we created the following function: Listing 33. In this listing, there is a small bug, that is not so easy to detect just by looking at it or reading through. But thanks to TDD and DDD principles we managed to catch this bug early in the development.

```

def differance_between_centers(frame_center, red_center):
    return ((frame_center[0] - red_center[0]), frame_center[1] - red_center[1])
```

Listing 33: Function before testing. Not working properly.

One of the principles of DDD is to divide large parts of a program into smaller ones. So we divided the calculation of displacement of two centers into its own function. This also made it easier to write tests for this simple, yet important task. After writing the tests and running them, it failed 2 out of 4 tests. Luckily it is a small function and modularly separated from the rest of the program so we could easily fix it. Turned out that all it was missing was to be multiplied by (-1) on the x-axis, see Listing 34.

```

def differance_between_centers(frame_center, red_center):
    return ((frame_center[0] - red_center[0]) * (-1), frame_center[1] - red_center[1])
```

Listing 34: Function after testing. Working properly.

In this example, the DDD concept of dividing large functions into smaller more focused ones had a positive impact. Together with the unit tests we had written, fixing this bug went quickly and pain-free. Both of these development methods fulfilled their purpose in this example resulting in an efficient work flow.

While writing tests for all functions creates a large initial time-consuming task, it makes maintaining and building upon the code significantly easier. While we may not have saved a ton of time considering the amount of time it takes to write tests, considering how much easier the code is to maintain and develop for future groups makes it a worthwhile investment in our opinion.

8.3 Challenges with Autonomous Docking

In order to solve the autonomous docking task, we need to navigate the ROV autonomously using a red center point located inside the docking station as a reference point. The process of solving this task began with a lot of trial and error. As we got further and further into the assignment, we learned new skills, we kept finding new solutions and improving upon the previous code iterations.

This section will discuss the different processes and solutions we went through during our development and explain what decisions were made. Additionally reflect upon what concepts we discarded, what concepts we kept, and overall how the development of this program went.

This section focuses solely on the process of autonomous docking. The integration of autonomous docking into the rest of the system will be discussed in section 8.8.

8.3.1 Initial idea, Pillow

As we had never done any image processing before, and had little knowledge of how to even import and open an image in a python-script we started by researching how to open an image in Python. We immediately found the Pillow Library. It is a third-party library, see figure 35 for the command to install it.

```
pip install Pillow
```

Listing 35: Command to install the Pillow-library to python.

The idea using Pillow was to loop through the frame pixel by pixel, checking if the RGB value of the pixel was red, and saving the pixel coordinates to a list. See figure 36 for the first version of the find_red function.

```python
from PIL import Image

def find_red(image_file):
    red_px_list = []

    # Open the image
    img = Image.open(image_file)

    # Get the size of the image
    width, height = img.size

    # Loop through each pixel of the image
    for x in range(width):
        for y in range(height):

            # Get the RGB values of the pixel
            r, g, b = img.getpixel((x, y))

            # Check if the pixel is within the red range
            if r >= 180 and g <= 80 and b <= 80:

                #append red pixel coordinate to list
                red_px_list.append((x, y))

    return red_px_list
```

Listing 36: First code written for autonomous docking.

The code works as intended. It takes in a frame, loops through the pixels, and saves the pixels with RGB values within a preset red range. For the input parameter, we gave it a 1080p image, since the cameras on the ROV are in 1080p. After writing this function we noticed it took quite some time to run the code. Because of this, we tested how long it took to execute the code. See table 4 for the results.

Time to run: find_red() with Pillow
2.0252
2.0713
2.0229
2.0399
2.0296
2.0437
2.0305
2.0460
2.0817
2.0431
Avg time: 2.0434

Table 4: Time it took to execute the code, in seconds. Using a 1080p image. Average time of 2.0434s.

With such a long time to run the code, we already encountered our first problem. It runs too slowly. The image we use is in 1080p format, which means 1920*1080 pixels, which is equal to 1920*1080 = 2073600. This is where Python, as a programming language, shows one of its weaknesses. The compiling speed is too slow for our needs. As the ROV is supposed to run in real-time and make decisions based on what it sees, it is important to get as much information as possible in order to be as accurate as possible. In other words, to get and process as many frames as possible within a short period of time. With the current iteration, it runs at roughly 0.5fps. Based on previous experiences and discussions with last year's computer science group, as well as other groups working on the Subsea project, we wanted the program to at least be able to perform at a rate of 10+ fps. Because of this, we had to come up with a different solution for processing a frame.

8.3.2 Using OpenCV to process a frame

Since looping through an image did not work we had to come up with another idea. Looking through posts online we discovered OpenCV (Open Source Computer Vision Library). It is a third-party library compatible with Python and made for image processing and computer vision. The library is huge, with inbuilt functions capable of solving many image-processing problems. Reading through the OpenCV documentation and posts online we found examples of how to isolate an object using color, in an image. And how to get the coordinates of that object. With a lot of trial and error, testing out several different masks, we found a method that works to get the center pixel of the red reference point in the frame. See listing 17 in chapter 6.3 for the final version of the **find_red()** function.

Since the time it takes to run the code is very important for solving this task we benchmarked this code as well. See table 5 for the results:

Time to run: find_red() with OpenCV
0.0356
0.0336
0.0335
0.0347
0.0342
0.0387
0.0337
0.0332
0.0345
0.0335
Avg time: 0.0345

Table 5: Time it took to execute the code, measured in seconds and using a 1080p image. Average time of 35ms.

With the results listed in the table above we are able to process roughly 28fps. After a lot of tweaking and testing, we managed to improve from the very first code using Pillow to the current code using OpenCV, resulting in a 56x increase in processed fps.

8.3.3 Finding red underwater

When trying to locate the red button in the docking station, we decided to use Pythons inRange function and simply look for red color. One thing to keep in mind is that light with long wavelengths gets absorbed by the water quicker compared to light with shorter wavelengths. This means that the color red decays quickly underwater with the water acting as a natural filtration of red [36]. See figure 44

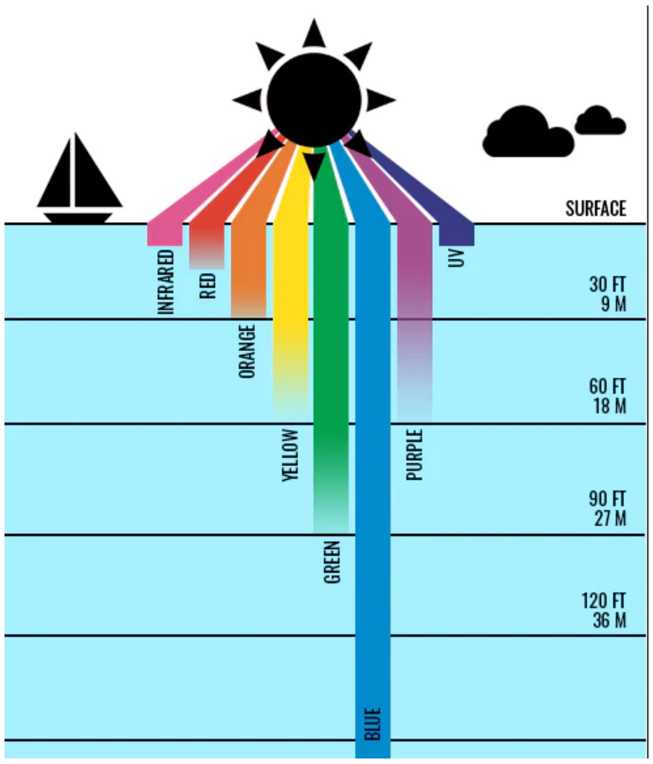


Figure 44: Illustration of light absorption at different wavelengths/ colors.

Knowing this, the program assumes that since the ROV is underwater, much of the natural red colors which could occur in the image will be naturally filtered out, and the red button in the docking station, although having also lost some of its red color, will be the only large mass of red color in the image.

8.3.4 Check rotation relative to docking station

Another problem we encountered during the development of Autonomous docking was how to know that the ROV was aligned, not rotated, relative to the docking station. See figure 45 and figure 46 for a visual representation of the problem.

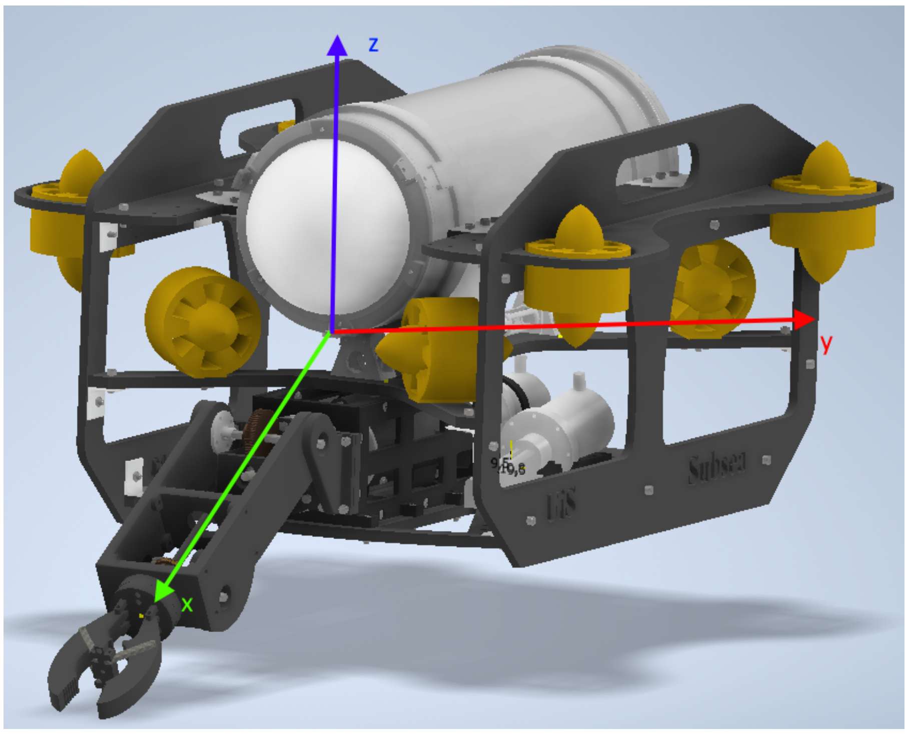


Figure 45: Visualization of the ROV's axes. x-axis is forwards, y-axis is left to right and z-axis is up or down.

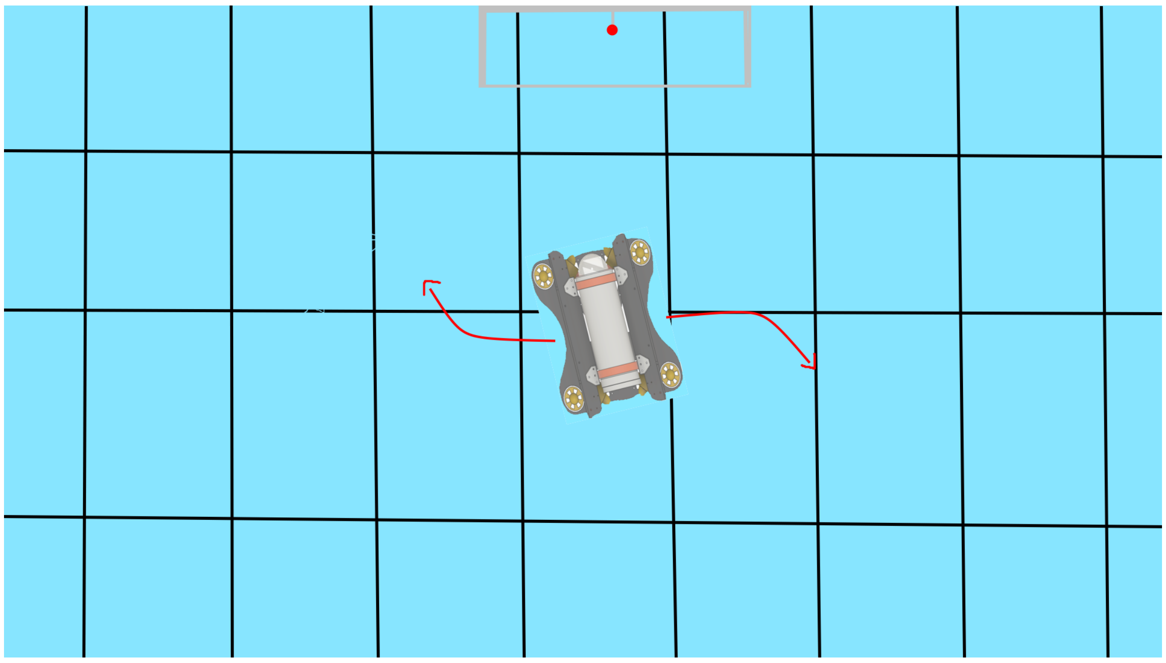


Figure 46: The problem is that the ROV is rotated on the x, y plane relative to the docking station. In this position, it can not go forwards. The ROV needs to rotate to the right in order to be aligned with the docking station. When it is aligned, it can go forwards and enter the docking station.

Talking to one of our mentors, Thomas R. Choat, regarding the issue, he gave us some advice on how to solve the problem. He told us that in previous years similar problems had been solved by looking at the pool grouts. In order for this solution to work, we have to assume that the docking station is aligned with the grouts on the pool floor. With this assumption, we can use the downwards-facing camera to calculate the angle between the grouts and the ROV. Knowing the angle between the ROV and the grouts, the program assumes it knows the angle between the ROV and the docking station. knowing the angle, the ROV can send driving commands to rotate itself into alignment and continue the docking process.

The code for autonomous docking is made into one single class. The reason for this is that it is easier to call from the Execution class that calls the different autonomous programs. Additionally having a class makes it easier to store objects and values within a program such as frames and driving data. Similarly to other solutions, we decided to split the functionality between several different functions. This makes them easier to test, maintain and write. This is part of our overarching principle of segmenting our code into small parts, a principle we have taken from DDD.

8.3.5 Testing the functionality

Unfortunately over the course of this bachelor's project, we did not get the chance to test autonomous docking on the ROV in the water. This was due to the ROV being close to being finished four days before the submission date for the bachelor's thesis. This put a limitation on our chance to try our features in the water. The water test was set to be launched on 11. April, but was ultimately postponed by one month. This was because of the ROV not yet being finished and ready for the water. The production of the ROV chassis and some electrical components were the reason for the wait. Though our programs were not yet completed either, we had some functionalities we could try out. At that point, autonomous docking was done, but still, we could not test it. Running this program in the water today would probably yield a poor result because there are some small things within the program that would need to be tweaked for optimal function. This may include:

- Adjusting color range for red centerpoint and for grouts, to find the most optimal contours.
- Adjusting how much thrust to give to the motors.
- Adjusting or changing filtration algorithms for finding correct contours.

Though we believe all of these points are covered and we have great logic for them, improvements could still be made. And given we never got the chance to test it in the water we cannot know for sure how well it works. For this thesis, we assume the autonomous docking works fine, based on the testing we did on land.

8.4 Discussion of Solution to Autonomous Transect Maneuver and Frog Count

8.4.1 Frog Count

We encountered a number of problems in the process of developing a solution for counting frogs, with the main issue being the grout joints between the floor tiles. To count the frogs we required a method with which to isolate the frogs in the image so that they could be correctly counted. We also had to keep track of which frogs were counted from the previous image, such that the same frogs were not counted several times. We had already realized from previous tasks that to achieve usable compiling speeds, using OpenCV was our best option. Contour detecting with OpenCV is crucial in this task when it comes to detecting frogs. Since contour detection is fully automated and does not take in any relevant parameters, our only options for isolating frogs are either using image manipulation prior to finding image contours or using some form of contour filtration after the contours are already found. We ended up using a mixture of these two methods in the end. Image manipulation prior to finding contours is responsible for the majority of filtering and mainly consists of trying to delete the grout joints without also removing the frogs. The first method we attempted to count frogs with was OpenCV's background subtraction method. This method takes in two pictures, an original picture and a picture of the background in the original picture. The function is then able to isolate the objects which do not appear in the background image. In theory, this should work well on a pool floor, where a background image of floor tiles can be easily provided. During testing, we found that when the background is moving, this function no longer works, and we had to look for other solutions. In the transect and dockings tasks we used the inRange function to solve similar tasks, however, for the frogs we had no good way of knowing exactly what color the frogs would be, and creating inRange functions for all possible colors would therefore be unfeasible. We could try to isolate the grouts using an in range function, but after some testing, we concluded with this solution not being consistent enough. Frogs of similar color to the grouts would be counted as well. Finally, we started looking into thresholding. The idea we had here was that we had to remove all the white tiles and dark gray grouts from the image in order to properly count the frogs. Using thresholding, we can use a very high value, removing all the white, and a low value, removing all the dark gray. This solution was also deemed too inaccurate, as white and black frogs would not be counted with this method. However, using trial and error, we found that using adaptive thresholding methods on the red value of the RGB image, the ROV managed to correctly isolate only the frogs (and not the grouts) in all color scenarios except for red frogs. To compensate for this we created a different function as well, which uses a grayscale image together with OpenCV's otsu and triangle thresholding methods to isolate the frogs. This method is unable to detect dark-colored frogs inside of grouts, as the grayscale image makes it too difficult to separate the frogs from the dark grouts, but using both of these functions in unison allowed us to find frogs of all colors, inside or outside of grouts. After manipulating the image and finding the contours, we also used the method in listing 22 to filter out several contours. The method uses OpenCV's approxPolyDP function to create a polygonal approximation of the contours. The polygons are then filtered out if they have less than 4 sides or more than 10 sides. These are somewhat arbitrary values that are based solely on testing. The idea is that the frogs on the seafloor should produce a polygon with approximately this many sides, and our epsilon value is tuned to reflect this. Continually the code fits a rectangle around the polygon and compares the width to the height. The frogs should produce fairly square shapes, as opposed to rectangular ones, so if the relation between the height and width is too low or too high the contour is faulty.

To prevent frogs from being counted twice we made a function that filters out overlapping frogs, see listing 23. We use this function after using our two methods for detecting frogs, as well as for making sure we don't count the same frog twice from one frame to the next in the video feed. A problem we ran into here, is that our code for detecting frogs is not 100% accurate. Since the code is analyzing a lot of frames, the code will inevitably pick up random noise as a frog, and will sometimes fail to see an actual frog for a frame or two. To combat this we made a function that keeps a counter going for each detected frog in a frame, and our frog counter is not incremented unless our code detects the same frog 10 times in a row, see listing 25. This gives us some protection from noise being counted. This value can also be easily increased or decreased depending on how strict we want the code to be. Finding the right values for this method is a fine balancing act, as having too high or low of a threshold for the function will mean either actual frogs not being counted, or random noise being counted as a frog. In the end, based on testing, we settled on 10 as the threshold for how many frames in a row a frog must be seen.

Another solution we considered using to count frogs was to use machine learning to train an AI model. However, we soon realized that in order to train an AI model, with solid and consistent accuracy, we would need a lot of data. [8] The amount of data varies from the complexity of the model, and how difficult it is for the AI to learn. It is important to notice that the quality of the dataset is almost as important as the quantity. It is better to have a smaller dataset with high-quality and accurate images, rather than a large amount of inaccurate, low-quality images. Since the dataset was the bottleneck we looked into other AI options. Another alternative that bypasses the problem of training AI models using large datasets, was to use a pre-trained model. We found YOLOv8 by Ultralytics [17]. YLOv8 is the latest family of YOLO based on Object Detection models from Ultralytics providing state-of-the-art performance [34]. It is built as a unified framework for training **Object Detection**, **Instance Segmentation**, and **Image Classification**. But the YOLOv8 repository also comes with five pre-trained models. Each of the models is made to solve different things. YOLOv8 Nano is the smallest and fastest, and YOLOv8 Extra Large is the most accurate but at the same time the slowest/ most taxing to run. These are the five models:

- **YOLOv8n**
- **YOLOv8s**
- **YOLOv8m**
- **YOLOv8l**
- **YOLOv8x**

The idea was that these models are trained to recognize common daily objects and perhaps they are trained to recognize frogs as well. If that was the case then we could use these pre-trained models to count the frogs. This is the code snipped we used to try testing the pre-trained models, see Listing 37:

```python
from ultralytics import YOLO
import cv2

model = YOLO("yolov8s.pt")


frame = cv2.imread("frogs.jpg")
results = model(frame, show=True)
cv2.waitKey(0)
```

Listing 37: Code used to test the pre-trained YOLO AI models

The code above is very simple. We import the YOLO models from the Ultralytics library. Then we choose which model to use. Finally, give the model a frame to analyze and show the output. We tried using all five models and found YOLOv8s to give the best results with our testing image. The image we used to test these models was one of the images provided by MATE, related to frog count. See figure 47 for the output of YOLOv8s:

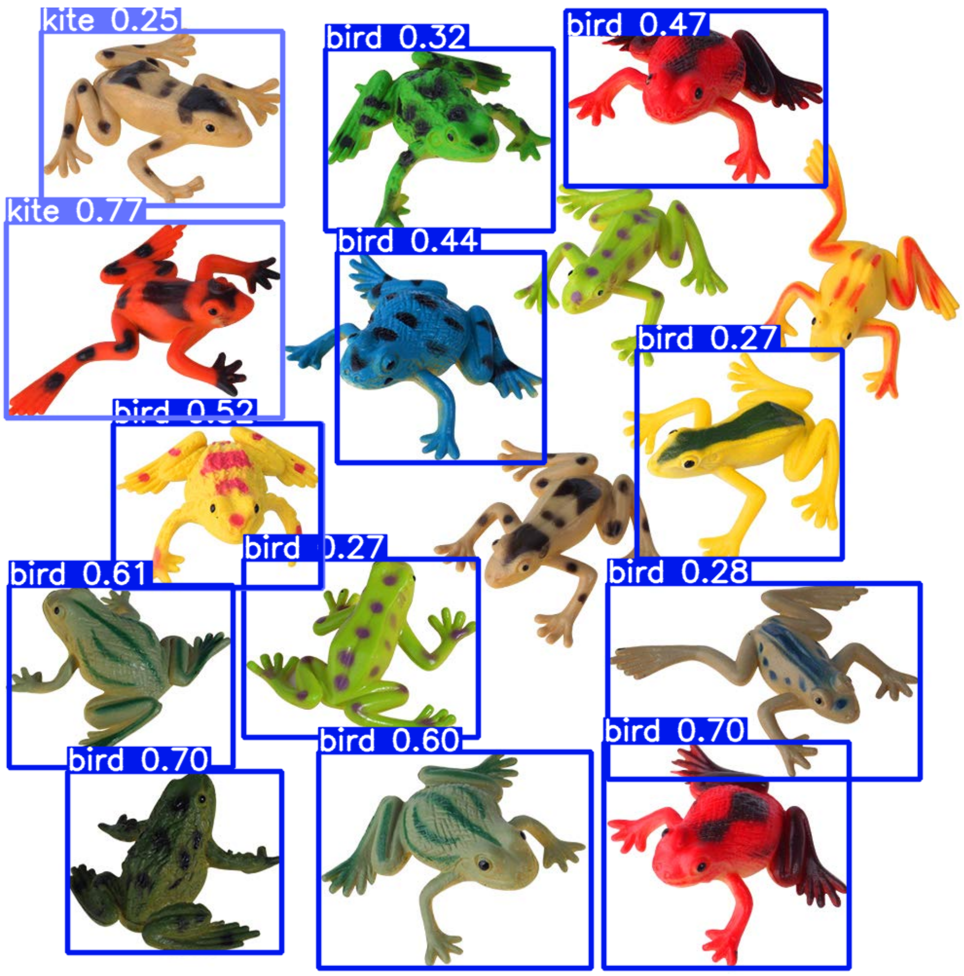


Figure 47: Output from YOLOv8s. The boxes highlight objects that are detected. The label describes what object it is and the number describes the certainty/ accuracy.

As shown in figure 47, we realized that the models are not trained for frogs. None of the labels were correct, mostly birds. And the accuracy/certainty score was low overall, best case being 0.7 and avg case being 0.47. Not a terrible accuracy score, but not great either. Given the incorrectness and inconsistency in the labeling, we decided that using pre-trained AI models was not a good fit for this task.

8.4.2 Transect Maneuver

We encountered far fewer problems in developing code for the transect maneuver compared to counting frogs. Thanks to other groups working on the ROV, we only had to take rotation in the horizontal plane into account when maneuvering the ROV. Our solution revolves around detecting the two large blue tubes on each side of the pool floor using a color range. The program then calculates the angle of the pipes so it can correctly rotate the ROV to face forward, after that it calculates how close the ROV is to each pipe to make sure the ROV stays in the middle of the pipes during the maneuver. It's important to maintain good stability on the ROV at all times, as drifting too far to one side would cause us to see a red pipe beneath the ROV, and our task would fail. Because of this, the ROV will stop driving if it detects the angle of the pipes to be over a threshold, or the displacement between the pipes to be too large. Then, after the angle and displacement have been corrected to acceptable levels will the ROV start driving forward again. We also made sure the code for fixing the angle of the ROV takes precedence over fixing the displacement, as fixing the displacement with a faulty driving angle would lead to faulty corrections. We also made sure the code for fixing displacement looks at the middle of the pipes for corrections, since this area is the least susceptible to faulty angles. As there is no time limit for this task we opted for slow and steady movements, this is reflected in the driving data packets we send to the ROV. Moving slowly is also beneficial to our frog count method as it looks at the overlap of frogs between the current frame and the last, and a slow-moving ROV will give a more stable frog count result.

8.4.3 Overall Thoughts

A large hindrance throughout the entire frog count developing process has been our lack of good testing data. Due to reasons beyond our control, we were unable to receive data from the ROV itself, and have had to resort to sub-optimal solutions like manually swimming across a pool with a camera while filming downwards. While this gives us some estimate as to how well the code is working, it is far from accurate to the final test using the ROV's camera. Looking back we might have been able to use the ROV from the previous year to get better test data or acquire a handheld camera more similar to the camera of the ROV. While on one hand, our subpar testing data makes it harder to solve the specific problem we were given, it also forced us to produce a more general solution to the problem as opposed to specifically solving it for this task only. The development of the transect maneuver on the other hand went very smoothly, although also here were we severely hindered by the inability to properly test our solution on the ROV, the logic works but the values of the driving commands we send need to be tuned depending on how much the ROV moves. In the end, we opted for low values in general, which gives stable results but is less efficient than needed.

8.5 Seagrass Monitor Discussion

The seagrass growth analysis task was fairly simple. Due to the high contrast difference between the white and green squares, isolating the green squares was a fairly simple task. Especially as the lines separating the green squares and the white ones are themselves light gray, making it easy to filter out along with the white squares. The only real difficulty in the development process concerns the area surrounding the seagrass module. After testing we decided that using simple binary thresholding, as opposed to searching for an RGB range, gave the greatest consistency. The only limitation of this method is that some small spots of the blue background get picked up by the thresholding and counted as squares. To counteract this we used some filtration methods on the contours after they were detected. Contours that are too small, not square-like enough, or too big are filtered out.

A component of this task we also had to think about was how we would take the pictures. The MATE challenge says nothing about how this should be done, so we were free to solve this how we wanted to. We decided that the most consistent solution would be taking two pictures manually, one before and one after, then running the pictures through the program to see the amount of growth. The alternative would be to use a solution similar to autonomous docking where the ROV autonomously moves into position, but this seemed unnecessarily complicated for this challenge. There are some limitations to the way we have decided to solve this. Firstly some key variables are hard-coded into the program. Things like thresholds for how large or small a square can be before being filtered out would ideally be dynamically calculated based on the distance between the ROV and the seagrass. Using a binary threshold will also not work if there are large contrast differences like sunlight on the seafloor. Powerful sunlight will cause a large shadow beneath the ROV and the binary values could cause errors. This is shouldn't an issue for us as this year's MATE challenge will be located indoors, and our solution is tested to work with indoor lamps. If shadows become an issue the program can be rewritten to use adaptive thresholding methods, which take shadows into account [30]. See figure 48 for an example of this.

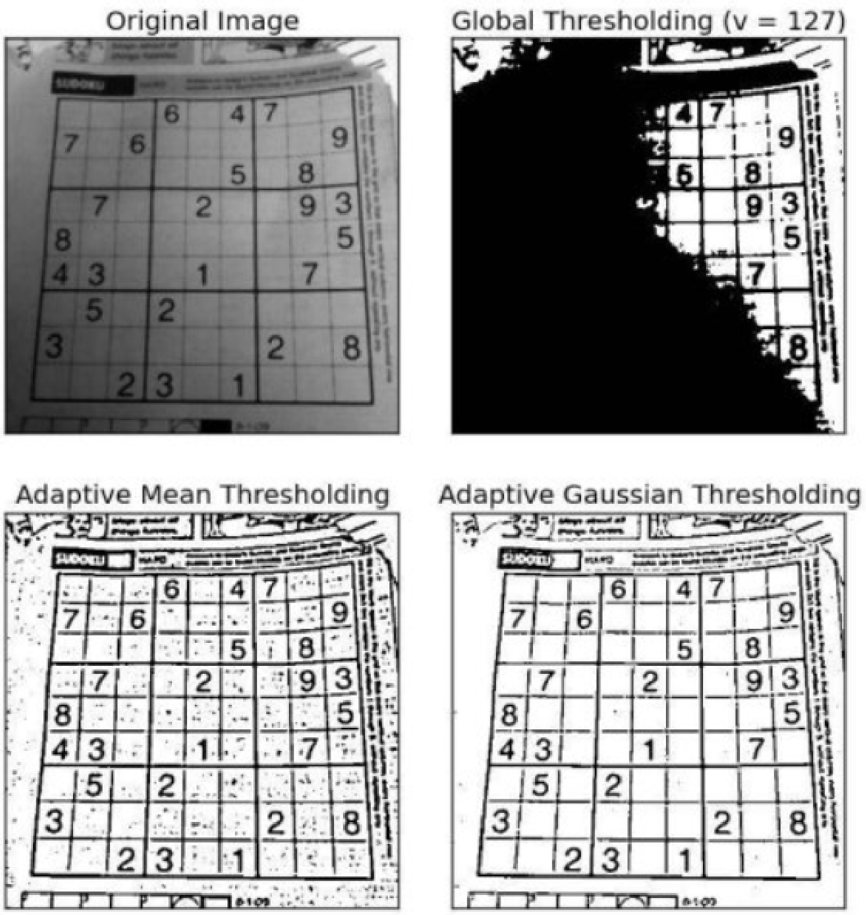

Original Image
Global Thresholding (v = 127)
Adaptive Mean Thresholding
Adaptive Gaussian Thresholding

Figure 48: Shows different thresholding methods used on an image with varying light conditions. Adaptive methods manage to overcome the varying light. Picture taken from [30]

8.6 3D Modelling

One of the problems we tried to solve during this bachelor's was 3d-modeling using computer vision. As previously mentioned we did not manage to complete this task, but we made some progress and got some incomplete results. This section will demonstrate and discuss the process and progress we made.

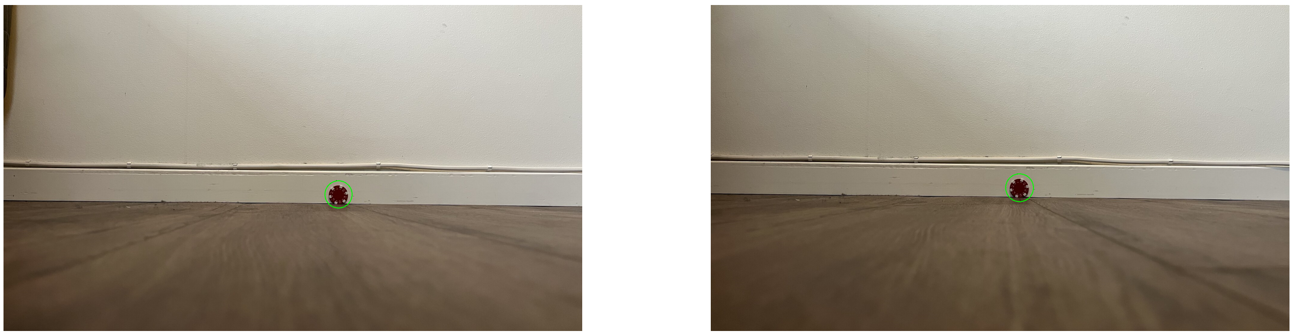
8.6.1 Measure depth

The first thing we did was to measure depth using the stereo cameras. If the program could measure depth, it could also measure size, and this way it would perhaps be able to use the sizes to create a matrix that could be plotted in 3D. See listing 38 for the code used to measure depth:

```
1  import numpy as np
2  
3  def find_depth(center_left, center_right, frame_left, frame_right, baseline, focal, fov):
4      _, width_right = frame_right.shape
5      _, width_left = frame_left.shape
6  
7      if width_left == width_right:
8          focal_pixel = (width_right * 0.5) / np.tan(fov * 0.5 * np.pi/180)
9  
10     else:
11         print("Left and right camera frames do not have same pixel width!")
12 
13     x_right = center_right[0] #x_koordinate value
14     x_left = center_left[0] #x_koordinate value
15     y_right = center_left[1]
16     y_left = center_left[1]
17 
18     disparity_x = x_left - x_right
19     disparity_y = y_left - y_right
20     total_disparity = np.sqrt((disparity_x ** 2) + (disparity_y ** 2)) #if the two cameras are
        not properly alligned, this handles that problem
21     depth = (baseline * focal_pixel) / total_disparity
22 
23     return abs(depth)
```

Listing 38: Function used to measure depth.

In listing 38 the function takes in seven parameters to calculate depth. Frame_left and frame_right are used for their shape. Center_left and center_right are the center pixel coordinates of the common object both cameras detect. Baseline is the distance between the two cameras in the stereocamera setup. Focal describes how much the camera lens zoom is. FOV is the angle the camera "sees", how wide the camera is able to capture. All these parameters are used when calculating the distance between an object and a stereocamera. In line 8, focal lenght is converted to focal pixel. This is so that we can use the focal pixels combined with the pixel disparity of the object between the to images to calculate the distance. In line 20 the program uses the pythagorean theorem to calculate the total disparity bweteen the two centers. Based on the total disparity and the focal pixel, the program can calculate the depth of the object. This function returns some decent results, see figure 49 for a test and for the output:


(a) Left stereocamera test frame, with object detection highlited in green.

(b) Right stereocamera test frame, with object detection highlited in green.

Figure 49: Notice that the test frames are not identical but that one is to the side of the other. The difference between the two frames is called the disparity and this is what the program uses to calculate the depth.

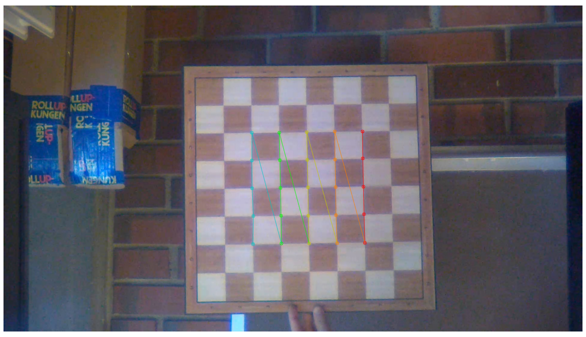
The test in figure 49 was done with two images taken by an iphone 13 on land. The baseline from the cameraposition for the two images was 6cm. The focal lenght for the camera was 24mm and the fov was 65degrees. We ran two tests with this setup. We used a tape measure in both cases to be sure of the exact distance. In the first test the tape measure measured 125cm. The program measured 119cm. In the second test the tape measure measured 93cm, the program measured 101cm. Both tests fell within 10% accuracy. The distance measured may be less accurate at very close and very far distances, this could have affected the outcome of the measurement. Some other variables that could be tweaked to better improve the outcome would be: To have a higher resolution camera and a more accurate algorithm for detecting the common object. This would improve the accuracy of the calculation of disparity, which in turn would have lead to a more accurate and consistent outcome. We also tried this code with the stereocamera setup on the ROV, but the quality of the cameras was a lot weaker leading to inconsistent and inaccurate results.

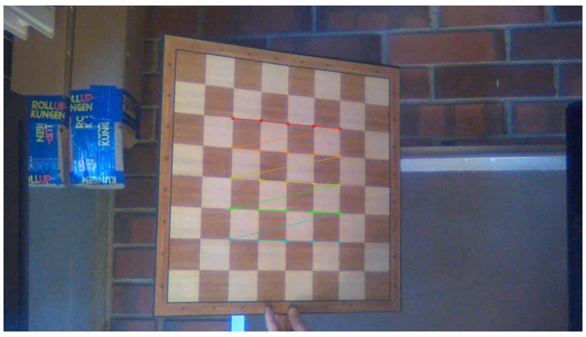
Knowing the distance to an object the program could calculate the size of the object using trigonometry. After the program calculated the size of the object was when the difficult part started. How to make size measurements to create a 3d-plot. Unfortunately we did not pursue this idea. Instead we tried out another more complex and advanced technique.

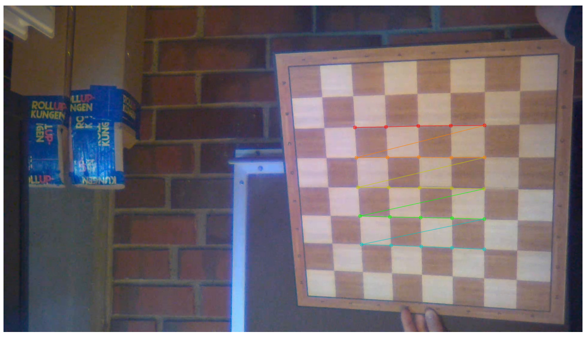
8.6.2 Cameracalibration and triangulation

In this method the idea is to take multiple photos, find common points in the images, join them and plot them using stereovision and triangulation. This method will be further discussed later in this section, but there are some preparation that needs to be done beforehand:

Firstly, the cameras need to be calibrated. We calibrated each camera using the **cv2.calibrateCamera()** function. This function requires imagepoints/ cornerpoints from multiple images as parameters. Finding these points is commonly done using multiple images of a chessboard, taken by the camera. And using the **cv2.findChessContours()** method to find the cornerpoints. With these points the program can create a camera-matrix for each of the cameras. These matrices fix distortions and warping of images taken by the camera. Since a cameralense is round and convex an image taken by the camera, if not calibrated properly can become distorted and or warped. These faults would negatively impact the process of finding common points and triangulate a 3D-model. See figure 50 for the images and output from the camera calibration process:


(a) Calibration 1


(b) Calibration 2


(c) Calibration 3

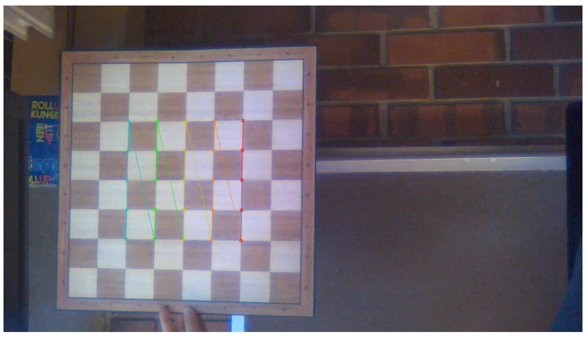

(d) Calibration 4

Figure 50: Four images taken by the left stereocamera to calibrate the left stereocamera. 4x4 cornerpoint detection used. Cornerpoints highlited with multiple colors in the center of the chessboards.

After the camera calibration has returned camera-matrices, we can use these to calibrate the stereocamera setup. This is done the same way as calibrating a camera, except we use the **cv2.stereoCalibrate()** function instead. This function takes in cornerpoints from both cameras and both camera-matrices as parameters. It returns the rotational matrix and the translation matrix as calibration matrices for the stereocamera setup. These matrices are used to compute the rotation and displacement of the image. In other words they are used by the program to "understand" where the chessboard, or other detected object, is relative to the stereocamera.

After getting the calibration matrices from calibrating the stereocamera. The program can now use these matrices to join images of an object to triangulate and create a three dimentional plot of the object. For the triangulation process, the program looks for keypoints in each of the images. This is done by using what is called a SIFT. This is an algorithm that essentially "sifts" an image finding characteristical/ identifying points in an image. See listing 39 for codesnippet of using sift to find keypoints and match them with another image:

```
sift = cv2.SIFT_create()

kp1, des1 = sift.detectAndCompute(gray1, None)
kp2, des2 = sift.detectAndCompute(gray2, None)

bf = cv2.BFMatcher()

matches = bf.knnMatch(des1, des2, k=2)
```

Listing 39: Codesnippet for finding keypoints in an image and match them.

As shown in listing 39 the program creates a sift object. By using this object's detectAndCompute() method the program can find keypoints in an image. After finding keypoints in two images, these points need to be matched against eachother to find common points between the two objects. As shown in the listing, it is done by first creating a BFMatcher() object. From this object the program uses the knnMatch() method to match together the keypoints. See figure 51 for visual representation of keypoints matched together between two images:

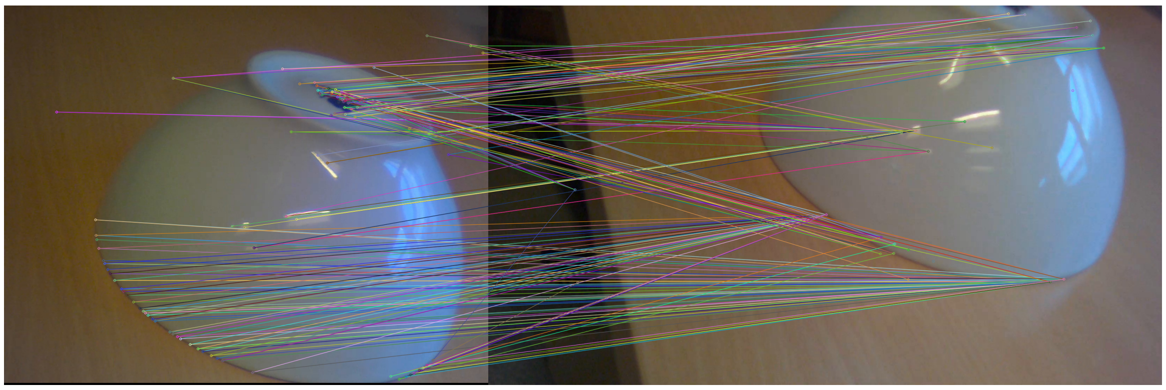


Figure 51: Visual representation of the different keypoints and how the program matches them. Keypoint being a dot and line being a match.

After the program finds keypoints and matches between the keypoints, it can use triangulation to create points in three dimensions. See listing 40 for codesnippet of the triangulation:

```
mtx1_extrinsic = np.hstack((mtx1, T))
mtx2_extrinsic = np.hstack((mtx2, T))
points_3d = cv2.triangulatePoints(mtx1_extrinsic, mtx2_extrinsic, pts1_2, pts2_2)
points_3d_homogeneous = np.hstack((points_3d, np.ones((points_3d.shape[0], 1))))
points_3d_normalized = cv2.convertPointsFromHomogeneous(points_3d_homogeneous.T).reshape(-1,
    3)
```

Listing 40: Codesnippet for using triangulation to create 3D points.

In listing 40, in line 1 and 2, the code adds the translation vector to the camera matrix of both cameras using np.hstack(). This inbuilt numpy function adds two arrays together horizontally. This is used to transform 3D world coordinates to 2D coordinates in an image. The **cv2.triangulatePoints()** function is used to create the 3D points. The last two lines in the listing change the format of the point into a format that can be plotted. See figure 52 for the final 3D plot output.

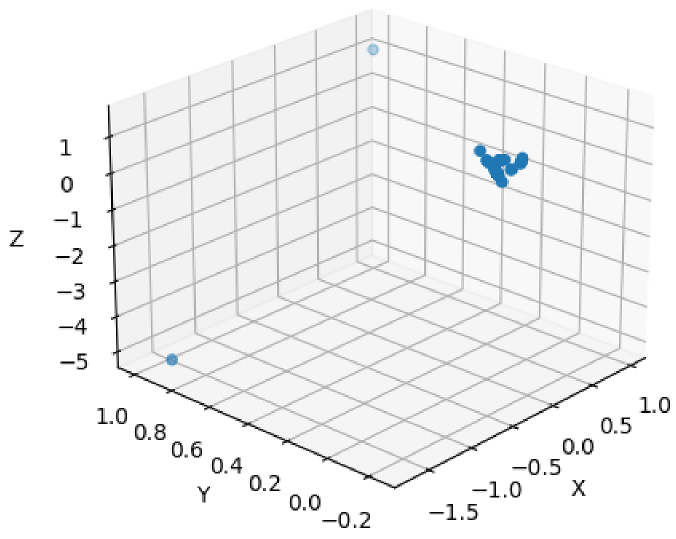


Figure 52: The 3D plot generated from the 3D modelling code.

The code in figure 52 is working in the sense that it runs without errors, but it clearly does not produce the wanted output. One of the reasons we think is affecting the output, is that we only used two images for the testing of the code. More images is required to create a more accurate model. Another factor we think may impact the result is the stereo camera and the calibration. The quality of the stereo camera is low with regards to color contrast and sharpness of the images it captures. This in turn affects the camera and stereo camera calibration. Additionally we used a 4x4 chess grid to calibrate the cameras across four images. An 8x8 chess board grid across more images would lead to better camera matrices and better stereo camera calibration.

There are still a lot of work needed to complete the 3D modelling task and we are not quite sure how everything works just yet. Regardless of solving the task or not, we learned a lot from trying to solve this problem including calibration of camera and stereo camera, camera matrices, keypoint matching and triangulation. There is still a lot more to learn about these concepts, but we

8.7 Camera feed

While developing the camera feed solution, we encountered several challenges. The first challenge was opening all four cameras that the ROV has, simultaneously. In the beginning, the code opened the cameras sequentially, resulting in slow updates from the cameras. It was immediately clear that a change was needed. The tasks required cannot have slow-running camera streams, as the image processing needs to happen in real time.

An attempt was made using Python's multiprocessing module for extracting frames directly from the GStreamer feed. This gave desired results, but it was taxing on the computer to run all the cameras all the time. For this, a solution had to be created, where the user can decide which camera is opened and when it is opened. To solve this problem, we created the camera manager class.

At first, the **camera manager class** handled everything for one camera at a time. This meant opening the GStreamer pipeline through a feed, retrieving the frames from the class, and showing the frames for each camera. With this implementation, each camera would need an initialization of this camera manager class, which would lead to a lot of redundancy. Because all the methods for opening the camera, retrieving frames, and showing frames can be shared, it was easier to handle each camera separately and feed them into a more general manager class. For this purpose, a **camera class** was created, which contained information about each individual camera. Each camera on the ROV would need to initialize a camera class, but this was now a lightweight class with only a few methods for opening it and extracting frames. This way the camera manager could have instances of a camera object, and use or stop each camera as needed. This makes the code more modular as it is easier to change one object, or class, without affecting the others.

8.7.1 GStreamer and OpenCV

The frameworks used for handling the camera data are GStreamer and OpenCV. To get good results, communication with the communications group was important, as they were responsible for setting up a pipeline from the ROV to the code. However, setting up the OpenCV library in Python with GStreamer required some work. Both of these libraries are not native to Python, i.e. they are written in a different language, and the functionality is ported over [33] [39]. Because of this, it was required for a custom installation of openCV from source, with gstreamer compatibility on. The tutorial for this will be the third item in the appendix. Without the custom installation, Python's version of OpenCV could not open the GStreamer feeds that the communication group set up.

To prevent such issues, a potential fix could be to not use Python as the main programming language. These two libraries paired together are better optimized for the C and C++ programming languages, and it could have been more efficient to write in them. [33][39] This decision could also improve some of the issues the Python programming language has, as discussed in the background portion of the thesis, such as dynamic typing, and execution speed. [6] Reasons for this include them being a compiled language instead of an interpreted language. While interpreted languages have to be interpreted first before being compiled, these languages skip this step entirely[18]. These also have static typing, which saves time for the compiler, as it does not have to spend valuable time calculating datatypes for variables [18]. Ultimately we decided a custom installation of OpenCV from source, with gstreamer compatibility in Python was the best solution.

8.8 Integration with GUI

At the start of the project, there was no focus on the integration with GUI. The focus was mostly on making sure the individual functions worked. When we finally started work on integrating our program into the GUI, it proved to be more time-consuming than we originally anticipated.

It might have been more efficient to wait longer before attempting to integrate the program into the GUI, as much of the GUI functionality was not fully completed when we began. However, due to internal Subsea deadlines, the ROV had to be ready for water testing 11. April, forcing us to integrate our program at an unideal time. We managed to integrate the program in time, but due to other bottlenecks in the ROV production, the water test was heavily postponed.

An improvement that also could have been made was spending more time collaborating with the GUI group to make the transition as easy as possible. Some challenges that occurred during the integration were:

- Driving Data
- Multiprocessing Queue
- Parallelism

8.8.1 Multiprocessing Pipe vs. Queue

As the GUI was not fully done when we integrated our program, information about how the GUI connects to the ROV was limited. The GUI group was responsible for sending commands down to the ROV from controllers, and for receiving the network data from the ROV. The structure of the driving data was non-negotiable as was decided beforehand by the electronic groups. This is important to be familiar with for the image processing tasks, as they are also required to send driving data to the ROV. Figuring out how to efficiently send driving data packets took some time. This is because we could not utilize a normal list or array, as several processes in the program would need to use it. This would need to be initialized in the GUI main file, as the controller would also need access to this.

Starting off a multiprocessing Pipe was utilized, which can have shared data within different processes[22]. This worked fine when the controller was tested to send data down to the ROV, however, when more than one source of input data was tested, it started to fail. This is because a pipe can only have one sender and one receiver at a time, this limits the functionality that is required for the driving data[22].

Reading through the official multiprocessing documentation it was straightforward to find a different data type that fits our purpose, the multiprocessing queue. A multiprocessing queue is a built-in data type in Python's multiprocessing module, that is compatible with multiprocessing and multithreading. This meant that the queue could be accessed by several links in the program chain, while still maintaining the same values. The queue will update everywhere, whenever you add or remove items[22]. This property allows both the controller and autonomous program to send data to the same receiver, and be sent down to the ROV. Although they would not need to happen simultaneously, as that would make driving the ROV difficult, it would still be important when the driving modes are changed.

8.8.2 Driving mode

To prevent any sort of crashing or emergency during the autonomous driving modes, it was decided that if the controller was to be used during an autonomous task, the autonomous task would immediately stop. To achieve this, a multiprocessing value was used, for the same reasons as for the multiprocessing queue. This data type is part of Python's multiprocessing module and works the same way. It can be accessed by several different processes while retaining the correct information[22]. Using this as a flag for running the autonomous tasks means that whenever the controller moves, all the autonomous functions stop immediately. This does lead to problems occurring if the controller is accidentally touched while performing an autonomous task, but that would be a human error. We could have instead implemented the failsafe on a single designated button on the controller, which would reduce the chance of human error.

8.8.3 Parallellism

Problems also occurred when multiprocessing was first attempted. The GUI group was using a mix of multiprocessing and multithreading in their program, as well as turning the daemon property of the process to true. When attempting to create new processes for the cameras like in section 6.1, the code would yield an error. The reason for this was that daemon processes cannot have children. It turned out that the main process for starting up the GUI process, with daemon active, makes processes inside of that code unable to run in parallel. To fix the issue, the GUI main process had to run without the daemon on, as that is meant for processes that are independently run in the background[32][22]. After this was solved, processes could now run inside of another process, but the limits have not been extensively tested as the testing on the physical ROV started late, due to bottlenecks in production. However, the project will be continually tested until the MATE ROV competition in June, and potential problems will be addressed before then.

8.8.4 The Execution class

The execution class was created for the purpose of keeping the GUI and image processing sides separate and modular. We believe that the class succeeded in fulfilling its purpose. Any changes made to the image processing code will in no way shape or form affect the GUI, as it has no dependencies on the GUI groups code. The only input for it is a multiprocessing queue and a multiprocessing value for the flag, which are static and will not be affected by other changes on their part. Writing code like this with modular design in mind makes it easy to replace or add parts without destroying existing functionalities and is a general good practice.[19] This is how the GUI code uses the execution class:

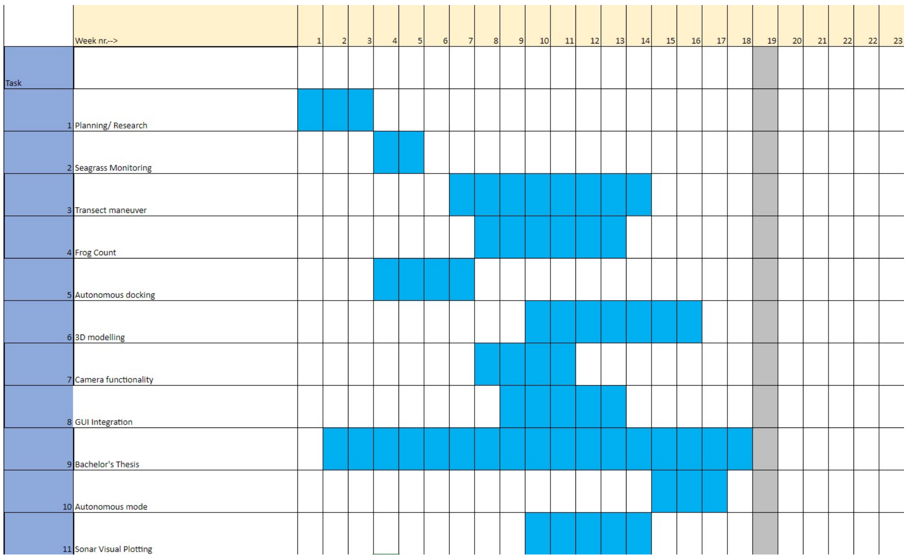
```python
class GUIWindow:
    def __init__(self, ..., queue, manual_flag)
        self.Exec = ExecutionClass(queue, manual_flag)
```

Listing 41: Execution class initialized in the GUI code simplified.

From here the GUI buttons can call the methods from the execution class as discussed in section 6.2. We believe that this was the best way to go about this, and would recommend this type of coding practice.

8.9 Work Flow

Coming into the subsea project at the beginning of the year, we decided to follow the advice of previous Subsea members and made a GANT schema, showcasing which tasks we planned on doing, and the time we estimated the tasks to take. Although we knew this timetable would inevitably be broken, it still provided valuable internal feedback as to how we were doing timewise on different tasks. See figure 53 for the GANT schema.



	Week nr.-->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	22	23
Task																				▨					
1	Planning/ Research	■	■	■																▨					
2	Seagrass Monitoring				■	■														▨					
3	Transect maneuver							■	■	■	■	■	■	■	■					▨					
4	Frog Count								■	■	■	■	■	■						▨					
5	Autonomous docking				■	■	■	■												▨					
6	3D modelling										■	■	■	■	■	■	■			▨					
7	Camera functionality								■	■	■									▨					
8	GUI Integration									■	■	■	■	■						▨					
9	Bachelor's Thesis		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	▨					
10	Autonomous mode															■	■	■		▨					
11	Sonar Visual Plotting										■	■	■	■	■					▨					

Figure 53: Image showing GANT schema made the first week of the project, showing our plans for development and expected time consumption of each task

While predicted timetables should not be taken too seriously as they are expected to be broken, we still feel we assigned too little time to the planning/research task. Looking back, we were overly eager to start programming and should have spent more time acquiring good test footage, test functions, and research before starting to problem-solve. This ended up costing us valuable time in the early parts of the project, as we got stuck on early solutions which were hard to test. We quickly learned from this, and in the later parts of the project, we focused on having good tests and test data before trying to come up with solutions.

9 Conclusion

In conclusion, we consider our bachelor thesis project to be a success overall. We managed to solve most of the image processing tasks given by MATE, except for 3D modeling, which turned out to be tougher than we anticipated. We successfully implemented the program into the ROV's backend system and GUI. We thoroughly tested the topside system on land, and it performed as intended.

While we haven't yet participated in the MATE challenge to get a definitive confirmation of our solutions, our internal testing showed positive results for frog count, transect maneuver, autonomous docking, and seagrass monitoring. Working on these tasks as part of the Subsea project gave us valuable experience in computer vision and collaborating with other engineering groups. We also learned a lot about time management while working on such a large project.

We are excited to continue developing our skills as software developers and embrace new opportunities for growth. This project has taught us important lessons, and we approach the future with humility and a commitment to improving our abilities in the field.

10 References

References

[1] *2023 ROV Explorer Manual*. Jan. 17, 2023. URL: https://files.materovcompetition.org/2023/2023_EXPLORER_Manual_FINAL_1_17_2023_withcover.pdf.

[2] *Accelerated GStreamer — Jetson Linux Developer Guide documentation*. URL: https://docs.nvidia.com/jetson/archives/r35.2.1/DeveloperGuide/text/SD/Multimedia/AcceleratedGstreamer.html (visited on 05/14/2023).

[3] *Application Development Manual*. URL: https://gstreamer.freedesktop.org/documentation/application-development/index.html?gi-language=c (visited on 05/14/2023).

[4] Asad Ali Asad. *Domain-Driven Design (DDD)*. GeeksforGeeks. May 21, 2020. URL: https://www.geeksforgeeks.org/domain-driven-design-ddd/ (visited on 05/14/2023).

[5] *Bibliography management in LaTeX*. URL: https://www.overleaf.com/learn/latex/Bibliography_management_in_LaTeX (visited on 05/14/2023).

[6] *C++ VS Python benchmarks, Which programming language or compiler is faster*. URL: https://programming-language-benchmarks.vercel.app/cpp-vs-python (visited on 05/14/2023).

[7] British Oceanographic Data Centre. *First Floats*. URL: https://www.ukargo.net/about/float_history/first_floats/ (visited on 05/14/2023).

[8] Eugene Dorfman. *How Much Data Is Required for Machine Learning? – PostIndustria*. URL: https://postindustria.com/how-much-data-is-required-for-machine-learning/ (visited on 05/14/2023).

[9] *Find if two rectangles overlap*. GeeksforGeeks. Feb. 19, 2014. URL: https://www.geeksforgeeks.org/find-two-rectangles-overlap/ (visited on 05/14/2023).

[10] *Float Cycle: IMOS.org.au*. URL: https://imos.org.au/argoworks/float-cycle (visited on 05/14/2023).

[11] *Floats & Drifters - Woods Hole Oceanographic Institution*. https://www.whoi.edu/. URL: https://www.whoi.edu/what-we-do/explore/instruments/instruments-floats-drifters/ (visited on 05/15/2023).

[12] *Floats | GO-BGC*. URL: https://www.go-bgc.org/floats (visited on 05/12/2023).

[13] *Getting Started with MATE Underwater Robotics*. URL: https://materovcompetition.org/get-started (visited on 05/12/2023).

[14] *GlobalInterpreterLock - Python Wiki*. URL: https://wiki.python.org/moin/GlobalInterpreterLock (visited on 05/12/2023).

[15] *Henry Melson Stommel Medal - Woods Hole Oceanographic Institution*. https://www.whoi.edu/. URL: https://www.whoi.edu/who-we-are/about-us/people/awards-recognition/henry-melson-stommel-medal/ (visited on 05/12/2023).

[16] *History*. MTS: ROV. URL: https://rov.org/history/ (visited on 05/14/2023).

[17] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. *YOLO by Ultralytics*. Version 8.0.0. Jan. 2023. URL: https://github.com/ultralytics/ultralytics (visited on 05/14/2023).

[18] Bryce Limón. *Compiled vs interpreted language: Basics for beginning devs*. Educative: Interactive Courses for Software Developers. URL: https://www.educative.io/blog/compiled-vs-interpreted-language (visited on 05/14/2023).

[19] Millie Macdonald. *Modular programming: Definitions, benefits, and predictions*. Blueprint - Blog by Tiny. URL: https://www.tiny.cloud/blog/modular-programming-principle/ (visited on 05/14/2023).

[20] Sea Technology magazine. *A Brief History of ROVs Sea Technology magazine*. Sea Technology magazine. Oct. 18, 2019. URL: http://sea-technology.com/a-brief-history-of-rovs (visited on 05/14/2023).

[21] *MATE - Marine Advanced Technology Education :: About*. URL: https://www.marinetech.org/about/ (visited on 05/12/2023).

[22] *multiprocessing — Process-based parallelism*. Python documentation. URL: https://docs.python.org/3/library/multiprocessing.html (visited on 05/12/2023).

[23] Jeff Novotny. *A Programmers' Guide to Python: Advantages & Disadvantages*. Linode Guides & Tutorials. Mar. 9, 2023. URL: https://www.linode.com/docs/guides/pros-and-cons-of-python/ (visited on 05/14/2023).

[24] *OpenCV: Canny Edge Detection*. URL: https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html (visited on 05/14/2023).

[25] *OpenCV: Class Index*. URL: https://docs.opencv.org/4.x/classes.html (visited on 05/14/2023).

[26] *OpenCV: Contours : Getting Started*. URL: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html (visited on 05/13/2023).

[27] *OpenCV: Flags for video I/O*. URL: https://docs.opencv.org/3.4/d4/d15/group__videoio__flags__base.html#gga023786be1ee68a9105bf2e48c700294da38dcac6866f7608675dd35ba0b9c3c07 (visited on 05/14/2023).

[28] *OpenCV: Introduction*. URL: https://docs.opencv.org/4.x/d1/dfb/intro.html (visited on 05/14/2023).

[29] OpenCVCamera. *OpenCV: Camera Calibration*. URL: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html (visited on 05/14/2023).

[30] OpenCVThreshold. *OpenCV: Image Thresholding*. URL: https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html (visited on 05/14/2023).

[31] *Overview*. Pillow (PIL Fork). URL: https://pillow.readthedocs.io/en/stable/handbook/overview.html (visited on 05/14/2023).

[32] *PEP 3143 – Standard daemon process library | peps.python.org*. URL: https://peps.python.org/pep-3143/ (visited on 05/15/2023).

[33] *Platforms*. OpenCV. URL: https://opencv.org/platforms/ (visited on 05/15/2023).

[34] Sovit Rath. *YOLOv8 Ultralytics: State-of-the-Art YOLO Models*. Jan. 10, 2023. URL: https://learnopencv.com/ultralytics-yolov8/ (visited on 05/14/2023).

[35] *Remotely Operated Vehicle (ROV) Services*. Oct. 1, 2003. URL: https://www.standard.no/pagefiles/978/u-102r1.pdf (visited on 02/09/2023).

[36] Jean Rydberg. *Why You Need Strobes Underwater*. Ikelite. URL: https://www.ikelite.com/blogs/buying-guides/why-you-need-strobes-underwater (visited on 05/14/2023).

[37] Ryan Sevey. *How Much Data is Needed to Train a (Good) Model?* How Much Data is Needed to Train a (Good) Model? Aug. 4, 2017. URL: https://www.datarobot.com/blog/how-much-data-is-needed-to-train-a-good-model/ (visited on 05/14/2023).

[38] *threading — Thread-based parallelism*. Python documentation. URL: https://docs.python.org/3/library/threading.html (visited on 05/12/2023).

[39] *Tutorials*. URL: https://gstreamer.freedesktop.org/documentation/tutorials/index.html?gi-language=python (visited on 05/15/2023).

[40] Ultralytics. *Home*. citation key: URL: https://docs.ultralytics.com/ (visited on 05/14/2023).

[41] Jash Unadkat. *What is Test Driven Development (TDD) ?* BrowserStack. URL: https://browserstack.wpengine.com/guide/what-is-test-driven-development/ (visited on 05/14/2023).

[42] *unittest — Unit testing framework*. Python documentation. URL: https://docs.python.org/3/library/unittest.html (visited on 05/14/2023).

[43] *Welcome to Python.org*. Python.org. May 11, 2023. URL: https://www.python.org/about/ (visited on 05/14/2023).

[44] *What Is an Underwater Rov?* Blue Robotics. URL: https://bluerobotics.com/learn/what-is-an-rov/ (visited on 05/12/2023).

11 Appendix

- Image processing Repository: https://github.com/UiS-Subsea/Bachelor_Bildebehandling
- GUI Repository: https://github.com/UiS-Subsea/Bachelor_GUI
- OpenCV Installation Manual: https://docs.opencv.org/4.7.0/da/df6/tutorial_py_table_of_contents_setup.html