

# Visual Servoing with BlueROV – Practical Report (P1 & P2)

## Introduction

In this project, we aimed to develop an autonomous system for the BlueROV underwater robot that can visually detect a target object and move toward it without manual control. We wanted the robot to continuously track the target's position and adjust its movement in real time, based only on the live camera feed.

To accomplish this, we implemented visual servoing, a method that uses feedback from the robot's camera to control its movements. Our system was designed in two main phases: object detection and tracking (using colour for buoys, and later, Aruco markers for more advanced tasks), and moving and steering the robot automatically using the tracking data (visual servoing).

## Tools and Setup

For this project, we used the BlueROV as our main test platform. We equipped it with both USB and onboard cameras to provide visual input for the tracking system. To keep things safe, we began our initial tests in the air and, after making sure everything was working correctly, continued with more realistic experiments in a water tank.

On the software side, our system ran on ROS2, and all of our code was written in Python. We used OpenCV for the actual image processing and the `cv_bridge` package to handle conversions between ROS images and standard OpenCV formats. To monitor and analyse our results, we relied on PlotJuggler, which allowed us to plot error signals and control commands as the robot was running.

We kept our project organised by separating everything into dedicated folders. The `bags` directory stored our ROS bag files, which were used for offline testing and data analysis. The `launch` directory held all the necessary launch files, making it easy to start the different nodes and set up new experiments quickly. Finally, the `script` directory contained all our core Python scripts, including those for tracking, camera calibration, and control logic.

Within our ROS2 workspace, we developed a few key nodes. The `image_processing_mir.py` script was in charge of tracking a single object based on its

color and sharing its position and size as ROS topics. The listenerMIR.py script received this tracking data and implemented the proportional controller that sent movement commands to the BlueROV. As an extension, we also developed a listener node that supports Aruco marker tracking, making it possible to work with multiple objects and handle multi-degree of freedom (multi-DOF) tasks in the future.

## **Object Detection and Tracking (Practical 1)**

We trained the BlueROV to follow an orange buoy by feeding camera frames into a ROS image-processing node. Each frame is converted to HSV and thresholded to isolate the buoy's colour, creating a binary mask. From that mask, we use pixel-average method to find the buoy's pixel-based centre and area, then translate those into meters using our camera's calibration data. We publish these values on two topics: `tracked_point`, which streams the buoy's current `[x, y, area]` and `desired_point`, where the robot should go. We defaulted to the image's centre, but adjustable with a mouse click. This process turns raw video into precise, metric position and size data, giving our control loop exactly what it needs for smooth autonomous navigation.

### Issues:

During the integration of control and image tracking files, the listener node never received valid tracking data, so the robot could not move autonomously. We found two mistakes. First, the image-processing node was publishing on the topic `'track_point'` while the listener node was subscribing to `'track_object'`. Therefore, the callback never worked and the position variables remained at zero. Second, even if the topic names had matched, the listener's callback function unpacked four values (`x_m, y_m, area_px, area_m`) from each message, but the publisher was only sending three (`x_m, y_m, area_m`), causing a format mismatch.

### Solutions:

On the Listener side, we renamed its topic to `'track_point'` so that both ends matched exactly. Then we either reduced the published array from four elements to three elements—`[x_m, y_m, area_m]`. Once both the topic name and the data layout were aligned, the listener began receiving valid position data and the robot was able to move autonomously as intended.

We started by testing the tracking code offline using ROS bag files to make sure everything was working as expected. With time running short, we switched to live trials in the water tank

for a more realistic check. While testing, we spent time adjusting the HSV thresholds to make sure the buoy detection stayed reliable even when the lighting conditions changed.

### Results :

As you can see in the video attached below, the image-processing node successfully tracked the orange buoy in most frames. As long as colour segmentation was well-calibrated, the binary mask cleanly highlights the buoy and the center is detected with the white dot marks.

### Limitations and future work:

During our practical session, we saw that the buoy was sometimes missed or the detection was unstable. To improve accuracy and stability, our professor recommended replacing our simple pixel-average with OpenCV's 2D moments. Because moments use the mask's full shape, they keep the center and area accurate even when the buoy's edges are uneven or partly hidden. Next, we'll add `cv2.moments()` to our code and compare its performance against our current method under different lighting and water conditions.


## **Visual Servoing Control (Practical 2)**

With reliable tracking, we developed a controller that uses the detected `[x, y, area]` data to move the robot. The controller follows a simple but reliable logic: the robot turns left or right (yaw) to keep the object centered horizontally, tilts up or down (pitch) to adjust its vertical position, and moves forward or backward (surge) depending on how big or small the object looks which gives an idea of how far away it is.

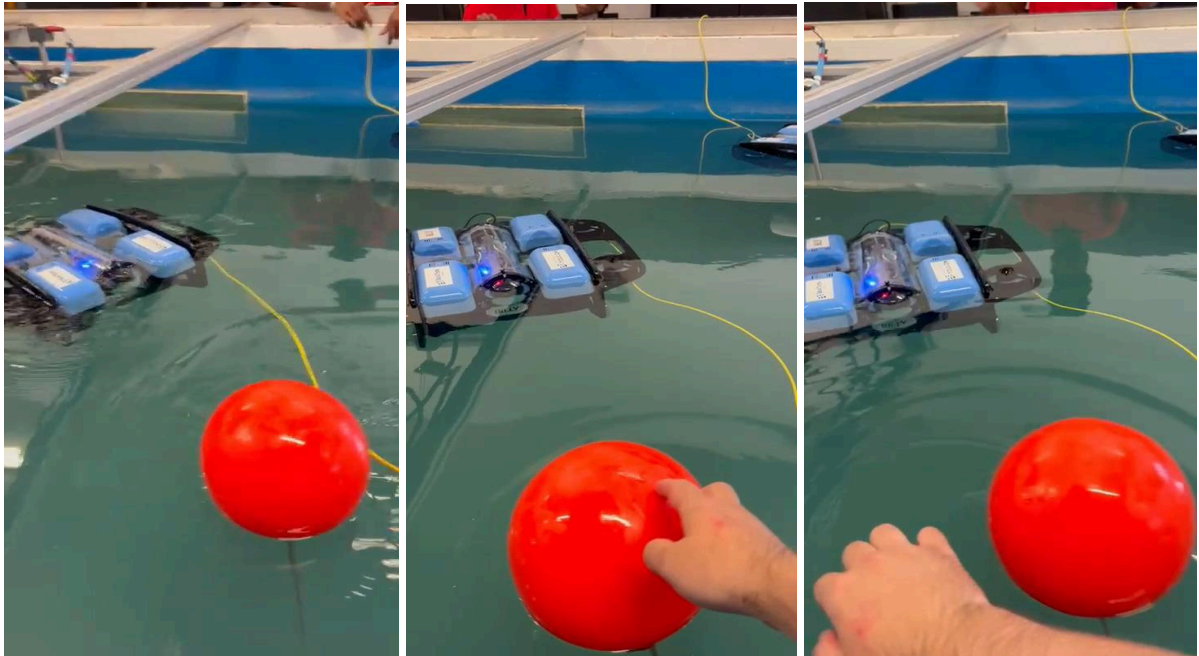
The `listenerMIR.py` node listens to the `/tracked_point` and `/desired_point` topics, calculates the difference between the current and desired positions and size, applies proportional gains, and then sends out movement commands either as PWM signals or ROS Twist messages, depending on how the test was set up.

Testing was planned to start with simulated movement using rosbag data. Once confident, we tested live in air, then in the water tank, checking how well the robot could follow and center the target. PlotJuggler was used to verify the controller's real-time response and check error signals, velocity, and feature tracking.

To support the analysis, we also captured a short video clip

 Visual Servoing with BlueROV - Practical work #2.mp4 during live testing in the water tank, showing the robot responding to the detected buoy. This video visually demonstrates

how the BlueROV adjusts its motion in real time using the controller output. Here are a couple of frames from the video to give a quick look, and you can find the full video in the supplementary materials.



#### Issues:

We faced 2 major issues regarding controlling the robot. First of all, it was too difficult to find the right settings for the controller gains ( $K_p$ \_pitch for vertical alignment,  $K_p$ \_yaw for horizontal centering, and  $K_p$ \_surge for maintaining distance). If they were too high, the robot would overreact and oscillate, but if they were too low, it became sluggish and slow to correct its position. Second, we also had to deal with occasional tracking problems from changes in lighting or reflections in the water, which sometimes caused the robot to momentarily lose the target and make unexpected movements.

#### Solutions:

To work around these problems, we experimented with a range of controller gain values for pitch, yaw, and surge. For example, we started by setting the pitch and yaw gains (how strongly the robot reacts to up/down and left/right errors) to values like 0.5, 1.0, and 1.5, and then watched how the robot behaved. If it moved too aggressively or started to oscillate, we'd dial the gain down. If it seemed too sluggish or slow to correct, we'd bump the gain up

a bit. For the surge (the forward/backward movement based on how big the object looked in the camera), we did the same kind of tweaking, usually trying gains between 0.2 and 1.0. Through this trial-and-error process, we eventually settled on gains of about 1.0 for pitch, 1.2 for yaw, and 0.5 for surge, since those gave us the best balance between smooth, stable following and quick enough reactions.

We also found that fine-tuning the color detection ranges and adding a bit of extra filtering really helped the robot ignore quick blips and stay locked onto the object, even if the lighting changed or there were some reflections in the water. In addition, we made sure to set up our experiments in spots with consistent lighting and to keep an eye out for any glare that might mess with the camera. In the end, the whole system became more reliable and easier to work with, even when things weren't perfect.

As a result, the BlueROV was able to follow the target smoothly, making quick and steady adjustments along the way. Surge control kept the distance to the object steady, and controller gains were adjusted to reduce oscillations and improve stability.

#### Limitations and future work:

We had planned to begin by replaying rosbag data for a simulated run before moving on to testing in a water tank. Unfortunately, our rosbag simulation failed due to lack of knowledge on ROS2. In addition, we also struggled to visualize topic data in PlotJuggler such as coordinate. To fix this, our next step is to code the rosbag playback and resolve our PlotJuggler setup issues. Once simulation and visualization are stable, we can validate the system under different conditions, making smooth tracking below the water.

### **Aruco Marker Tracking (Bonus/Future Work)**

Our proposed implementation of the Aruco marker-based visual servoing would start with developing a detection script using OpenCV's Aruco module. This would utilize the `cv2.aruco.detectMarkers()` to identify multiple markers simultaneously and extract their corner coordinates. We would then create a ROS2 node subscribing to the BlueROV's camera feed.

The advancement over our current implementation would be computing the transformation matrices between the camera frame and each detected marker. This would require using the camera calibration parameters and the

`cv2.aruco.estimatePoseSingleMarkers()` function to determine the rotation and

translation vectors. The transformation matrices would allow us to represent marker positions in the BlueROV's reference frame.

For visual servoing with 6DOF control, we would implement an algorithm that would:

1. Compute the error between current marker positions and desired positions
2. Apply a control law (extending our current existing proportional controller) to generate velocity commands in all DOF
3. Convert the velocity commands directly to PWM signals for the BlueROV's thrusters with appropriate scaling factors.

We believe this approach would enable better control capabilities than our current implementation, allowing for precise positioning and orientation control relative to multiple markers simultaneously. While we did not get to complete this implementation, the groundwork currently written in our system provides a good foundation for this implementation.

## **Conclusion**

Through this project, we learned how to bring together computer vision and real-time control to make an actual robot react intelligently to what it sees. We developed a full pipeline from robust object/marker detection to proportional control and real-world testing with the BlueROV. Along the way, we learned to tune parameters, debug ROS2 communication, and evaluate system performance using bag files and live data.

The next step will be to finish multi-marker, multi-DOF control using Aruco markers, enabling the robot to perform more complex underwater tasks.

## **Appendix**

The HSV values we used for detecting the orange buoy can be found in our `camera_parameters.py` script, and are also mentioned in the experimental descriptions throughout the report. All the camera calibration parameters along with the formulas we used to convert from pixel measurements to real-world distances are also stored in that file.

For anyone wanting to review or replicate our tests, we saved our main experimental data as ROS bag files. The file named 2023-12-18-17-23-28.bag contains the results of our air tests focused on object tracking, while 2023-12-18-17-24-28.bag records the experiments we performed in the water tank to test visual servoing.