ELEC330

Assignment 3 detail report

Turtle in water –Teri–Transformative, Efficient, Robust, Intelligent Underwater Bio-Robot

Date: ***May 14, 2025***

Abstract

This report presents the development of TERI (Transformative, Efficient, Robust, Intelligent), a biomimetic underwater robot inspired by sea turtles. This project aims to simulate and evaluate the autonomous underwater navigation, perception and mapping capabilities of robots using ROS 2 and Gazebo.A realistic underwater environment, *Atlantis.world*, was constructed to realize fluid dynamics including buoyancy and drag. The robot's propulsion was modeled using thrusters and joint position controllers, enabling fin-based motion control. TERI was equipped with simulated LiDAR and camera sensors to perform real-time obstacle detection and red object recognition. A state machine algorithm enabled the robot to navigate complex environments. SLAM capabilities were explored, achieving partial map generation but facing challenges such as frame drift and scan misalignment. The project demonstrates the feasibility of integrating biomimetic design and intelligent control to achieve autonomous underwater behavior. Future improvements will focus on enhancing SLAM accuracy and transitioning toward physical implementation.

1. Introduction

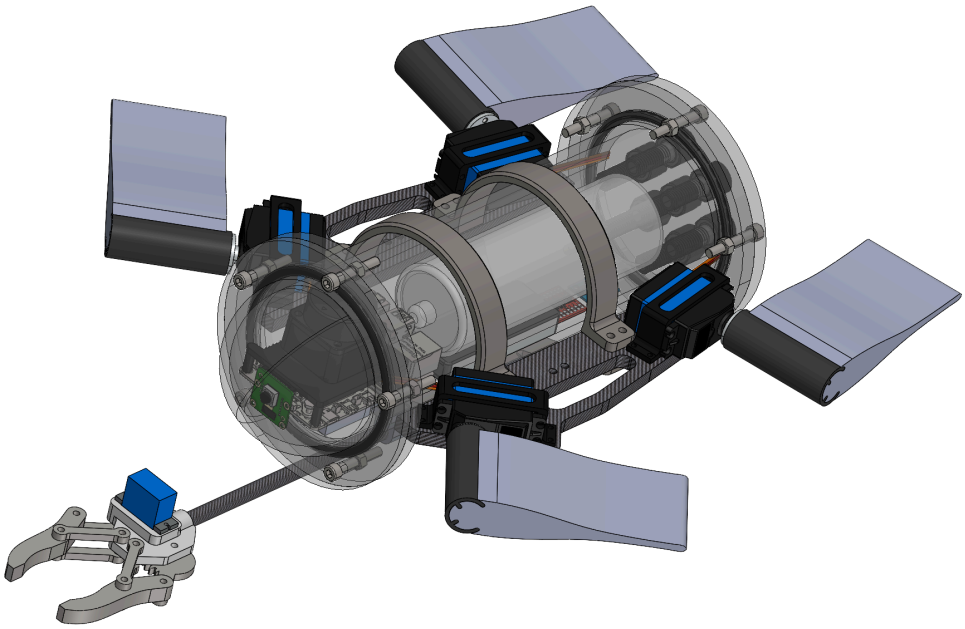

Figure 1. Bionic underwater robot - TERI

In recent years, biomimetic robotics has emerged as a promising field by drawing inspiration from nature to solve complex engineering challenges. This report presents TERI (Transformative, Efficient, Robust, Intelligent underwater bio-robot), a bionic underwater robot designed to emulate the swimming mechanics and adaptability of sea turtles.

The primary objective of TERI is to autonomously navigate and explore complex underwater environments. To achieve this, a simulated testbed named *Atlantis.world* was constructed using Gazebo, replicating realistic underwater physics including buoyancy, resistance, and diverse terrain. TERI was equipped with key functionalities such as object recognition, obstacle avoidance, and environmental mapping.

The system integrates mechanical design, motion control, sensor fusion, visual perception, and SLAM (Simultaneous Localization and Mapping) into a cohesive robotic platform. LiDAR and camera data enable environmental awareness, while state machine navigation algorithms allow for real-time decision making. The project aims to evaluate the feasibility and performance of a fully simulated biomimetic robot capable of autonomous underwater operations.

This report outlines the design methodology, implementation strategies, simulation results, and key challenges encountered during development. Through the TERI project, we explore the potential of combining biomimicry, physics simulation, and intelligent control to advance underwater robotics research.

2. Mechanical and Hardware Design

Compared to the initial conceptual design, the new design places greater emphasis on the practical feasibility of the robot. In simulation and real-world applications, new sensors had to be integrated into the robot's design. Due to the limited internal space and budget constraints, component selection required careful consideration of performance, ROS compatibility, cost, and physical dimensions.

Table 1. Bill of Materials

Components	Supplier	Catalogue Code	Unit Price (£)	Quantity	Subtotal (£)	Justification
Raspberry Pi 4B 4GB	Rapid Electronics LTD	75-1008	43.3	1	43.3	Strong computing power, good ROS compatibility
Lidar Module	Amazon	B0DCW3S4L8	24.99	1	24.99	Used for scanning the terrain.
Camera Module	RS Components	913-2664	14.27	1	14.27	High-quality imaging for object recognition tasks
MPU-6050 Module	Amazon	B07DJ4KMBF	4.99	1	4.99	Provides acceleration and gyroscope data for robot orientation estimation
2000mAh 7.4 V Li-ion Battery	Amazon	B0C6THFXXQ	12.5	1	12.5	Long battery life, stable power supply
9-36V to USB 5V Transformer	Amazon	B09B829DL9	5.99	1	5.99	Efficiently steps down battery voltage to 5V for Raspberry Pi
USB C Cable 30CM	Amazon	B0B95Z3RX8	3.99	1	3.99	Used to power the Raspberry Pi
ROV Underwater Sealed Compartment 90mm × 240mm	Ovision	N/A	38.7	1	38.7	Serves as the main structure of the robot, housing electronic components and the ballast tank.
Mini Self-Priming Gear Pump	Amazon	B0DGGB2V59	7	1	7	Used for injecting and extracting water into the syringe (ballast tank).
200ml Liquid Syringe with Tube	Amazon	B0DF7PT5PR	3.5	1	3.5	Serves as ballast tank.
TB6612FNG Dual Motor Driver Module	Amazon	B087R8ZR8F	3.5	1	3.5	Drives the water pump.
Waterproof 35kg Digital Servo	Amazon	B0CBBS6KX6	18.99	4	75.96	Powers the fins to control the robot's movement.
eSUN PETG 3D Printer Filament	Amazon	B0834JTP1K	19.99	1	19.99	Used for 3D printing clamps, grippers, and other structural parts.
TINMORRY TPU 3D Printer Filament	Amazon	B08HVQF4D8	14.99	1	14.99	Flexible material for 3D printing the fins.
PVC Board 8mm Thick	Amazon	B0DBZW62ZL	19.69	1	19.69	Used to Secure the servo motor after cutting.
0.39 Inch PVC Rigid Tube Round Pipe 8mm	Amazon	B0DP1YZH8P	4.99	1	4.99	Connect gripper and robot body.
Waterproof 1.8kg Digital Servo	Amazon	B0BKQGNWYL	6.78	1	6.78	Enables actuation of the gripper mechanism.
M2 x 16mm Hex Socket Head Cap Screws	Amazon	B0F18196VW	0.0599	2	0.1198	Essential component for robot functionality
M3 x 6mm Hex Socket Head Cap Screws	Amazon	B0CJFPTSQ4	0.04495	19	0.85405	Essential component for robot functionality
M3 x 10mm Hex Socket Head Cap Screws	Amazon	B0CJFM3F3V	0.04495	24	1.0788	Essential component for robot functionality
M3 x 16mm Hex Socket Head Cap Screws	Amazon	B0CJFMFQXX	0.0799	18	1.4382	Essential component for robot functionality
M3 x 20mm Hex Socket Head Cap Screws	Amazon	B0CJFPWB6D	0.0899	4	0.3596	Essential component for robot functionality
M4 x 25mm Hex Socket Head Cap Screws	Amazon	B0CJFLYQT1	0.1598	24	3.8352	Essential component for robot functionality
M2 Self Locking Nuts	Amazon	B0CDPXDP7Q	0.0849	2	0.1698	Essential component for robot functionality
M3 Self Locking Nuts	Amazon	B0DK1FNVXF	0.1058	49	5.1842	Essential component for robot functionality
M4 Self Locking Nuts	Amazon	B093TCF5G2	0.2245	8	1.796	Essential component for robot functionality
M3x4x5mm Threaded Insert Knurled Brass Nuts	Amazon	B0CRVLLGH4	0.4317	8	3.4536	Essential component for robot functionality
Electrical Cable	Amazon	B0C7TJ9191	8.99	1	8.99	Essential component for robot functionality
				TOTAL PRICE:	332.40925	

The bill of materials above demonstrates a well-balanced design in terms of cost, functionality, and integration. With a total cost of approximately £332, the robot achieves a comprehensive suite of robotic capabilities. Core components such as the Raspberry Pi 4B, LiDAR, camera module, and IMU provide robust computational and perception support for autonomous navigation and object recognition. The inclusion of waterproof servos, a gear pump, and gripper components further enhances the robot's mechanical interaction with its environment, making it suitable for underwater tasks. The use of PETG and TPU filament materials enables the production of custom 3D-printed structures—including fins and mounts—ensuring mechanical adaptability and facilitating rapid prototyping. Overall, the selected components are widely available, cost-effective, and compatible with ROS-based systems, making the design both technically feasible and economically efficient for experimental research.

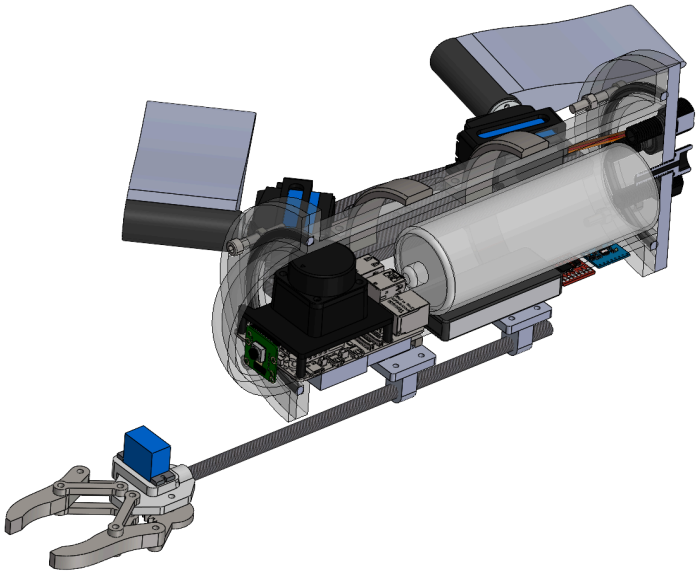

Figure 2. Internal structure

Compared to the initial design, mechanical structural optimization primarily focused on internal layout. Given the limited space within the robot, the placement of components required a comprehensive approach to ensure balanced weight distribution and prevent hardware interference (such as obstructing the LiDAR). As shown in Figure 2, the installation positions of essential components—including the camera, LiDAR, main controller, IMU, and the TB6612 motor driver—have been carefully arranged to maximize the use of internal space and maintain functional integrity.

3. World File (Atlantis.world)

3.1 Plugins utilized in the world

Atlantis is the .world file used for the robot to navigate. To simulate an underwater world, we will need the physics of the buoyancy of objects when immersed in the fluid by adding the gz-sim-physics-system and gz-sim-buoyancy-system plugins onto the world.

- **Physics**(gz-sim-physics-system)
  Responsible for loading and managing the physics engine, enabling Gazebo to simulate the dynamic behaviors of objects, such as gravity, collisions, and motion. This is a fundamental plugin for simulating physical environments

- **Buoyancy**(gz-sim-buoyancy-system)
  Used to simulate the buoyancy of objects in a fluid. The plugin calculates buoyancy based on the volume of the object submerged in the liquid, and it allows the specification of the fluid's density [1].

  To achieve neutral buoyancy for the robot (where the buoyancy force equals the weight, allowing it to hover), we can check the robot's volume in the Gazebo simulation (Figure 3) and calculate the buoyant force. Then, we adjust the mass and center of gravity of the robot's components so that the gravitational force matches the buoyant force.

  It is worth noting that during the robot's design phase, we incorporated ballast tanks inside the robot and designed attachment points for external floats and ballast weights. This allows the robot's weight and center of gravity to be easily adjusted to achieve neutral buoyancy during actual development.

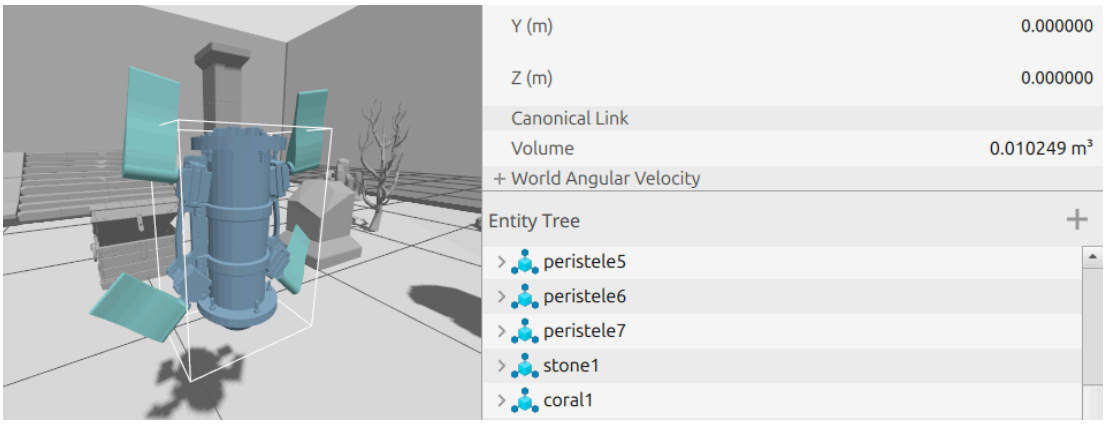

Figure 3. Checking the robot's volume

3.2 Implementation of Building the World (Atlantis.world)

Then, it is necessary to create an appropriate living environment for the robot, as shown in Figure 4.

The concept of this world is to resemble the lost city of Atlantis—an advanced civilization of its time that disappeared for centuries and was recently discovered at the bottom of the ocean—providing an environment for TERI to explore, simulating the robot's application scenarios.

We set the world's limit to be 10x10x3 units (meters), with ground_plane as the 10x10 base and wall1, wall2, wall3, and wall4 as the 10x3 side barriers. As shown in Figure 4, we can see that the walls are transparent. This is done by setting the material's opacity value to 0.1.

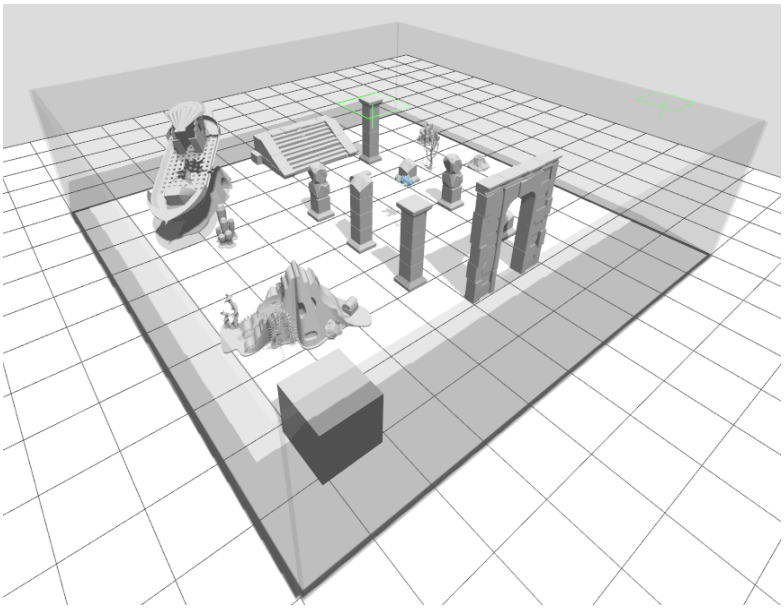

Figure 4. Atlantis.world

Atlantis has different obstacles that are exported from Gazebo's database and STL models.

- For simple objects like spheres and cubes, they can be directly defined in the world file.
- For more complex objects, such as arches and shipwrecks, SolidWorks is used to edit the model and export it as an STL file. The STL file should then be placed into the mesh directory of the work package, specifically under the collision and visual subfolders, and referenced in the world file with the correct file path [2].

  It is worth noting that when exporting models from SolidWorks as STL files, the unit of measurement must be changed to meters (the default is millimeters). Otherwise, this may result in the STL file failing to load in Gazebo or having excessively large dimensions.

  To simulate a more realistic underwater exploration setting while aligning with the objectives outlined in Assignment 1, additional elements such as a shipwreck, broken peristyles, an arch, and a treasure box are included. Furthermore, a 1×1×1 cube, a

sphere with a radius of 0.25, and a stair step with a slope are added to fulfill the required object specifications . Each object is precisely positioned by adjusting its X, Y, and Z axes, as well as its pitch, yaw, and roll rotations. The arrangement of all objects in the Atlantis environment is shown in Figure 5.

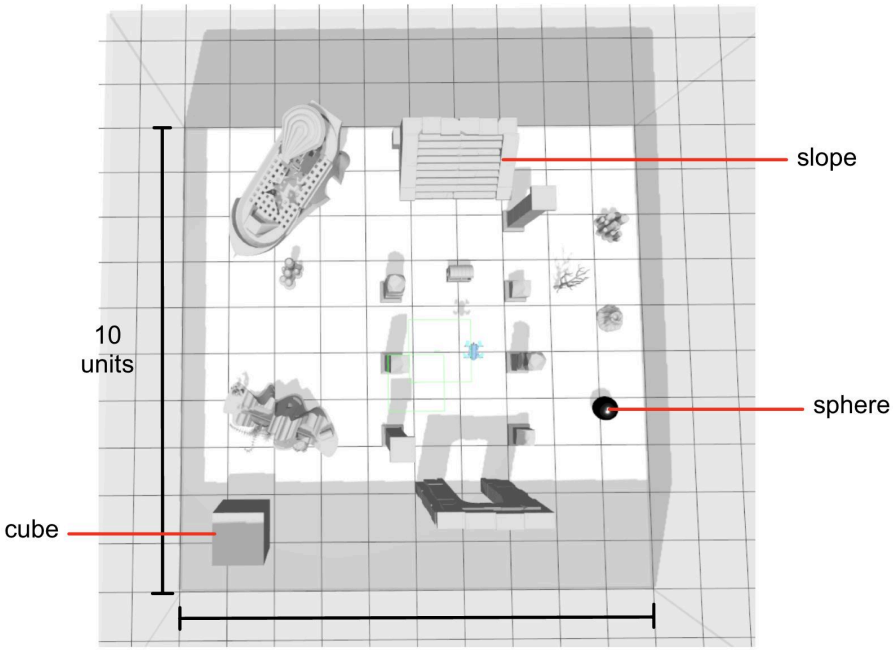

Figure 5. Bird's eye view of the world

During testing, it was observed that more than three environment-related obstacles of various sizes and shapes appeared in the robot's line of sight at certain points along its navigation path (Figure 6).

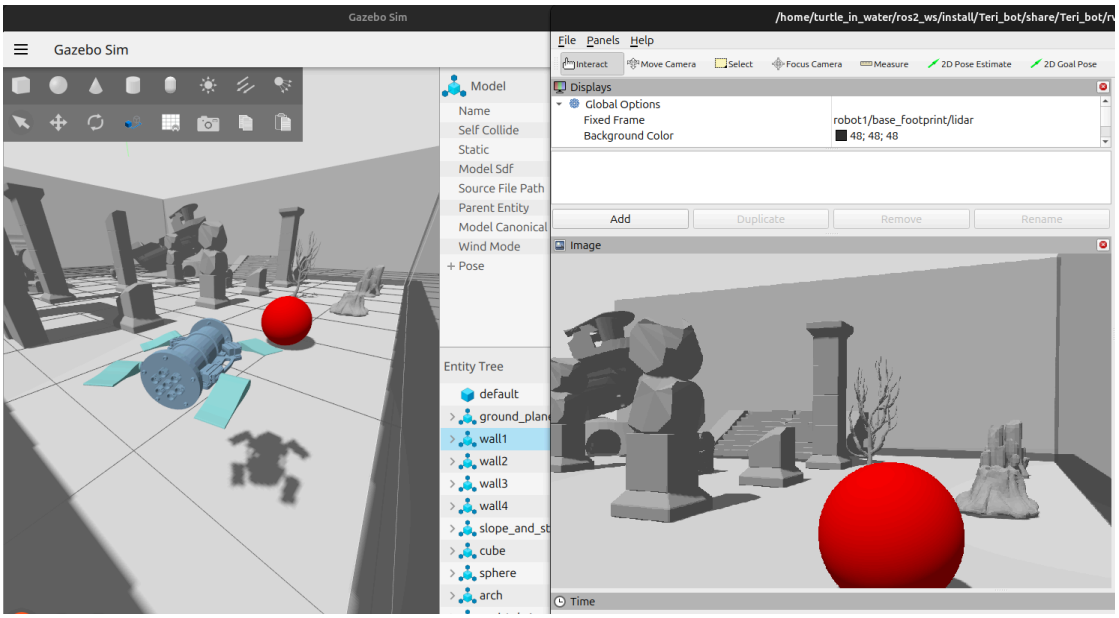

Figure 6. The robot's sight

4. Motion control (Teri_urdf.sdf)

4.1 Utilized plugins

Teri_urdf.sdf file is used to define the physical and visual properties of the TERI robot in the simulation environment. It includes necessary plugins for motion control and sensors. The file defines key parameters for the robot, such as joint configurations, environmental interactions, control types, and sensor types. This file is crucial for achieving realistic simulation of TERI's behavior, interaction with the environment, and compatibility with the simulation framework.

4.1.1 Hydrodynamics

In the simulation of underwater environments, the buoyancy plugin can simulate the buoyant force acting on objects in the fluid. However, during testing, it was observed that: :

- When the robot is loaded into the Atlantis simulation environment, if the center of mass and the center of volume are not aligned, the robot oscillates continuously back and forth due to imbalance.

This phenomenon occurs because the buoyancy plugin does not account for the forces exerted by fluid flow on the robot's motion, resulting in less realistic simulation outcomes. In real environments, fluid forces, including motion resistance and torques, significantly influence the dynamic behavior of objects. To address this issue, the Hydrodynamics plugin [3] was introduced .

The Hydrodynamics plugin simulates hydrodynamic behavior by defining the forces and torques between the robot and the fluid. This enables a more realistic representation of the resistance, acceleration response, and stability changes induced by the fluid flow acting on the object in the water.

Several parameters need to be calculated and simulated:

1. **Linear Velocity-Related Parameters**:
   xDotU, yDotV, and zDotW define the contribution of velocity changes along each axis to drag forces.
2. **Angular Velocity-Related Parameters**:
   kDotP, mDotQ, and nDotR are used to model the stabilizing torques exerted by the fluid during rotational motion.
3. **Nonlinear Components**:
   xUabsU, yVabsV, and zWabsW describe the nonlinear behavior of fluid resistance, where the drag increases proportionally to the square of velocity at high speeds.

Given the complexity of accurately calculating these parameters for a robot with a complex structure, we approximated the robot's main body structure to a cylinder after analysis. Additionally, formulas and results for calculating these parameters for cylinders were obtained through research [4].

After integrating the Hydrodynamics plugin and conducting the simulation again, the robot's motion was found to align with real-world expectations:

- By simulating the damping torques generated by fluid flow during rotation, the robot's oscillation gradually stabilized and ultimately came to rest (Figure 7).

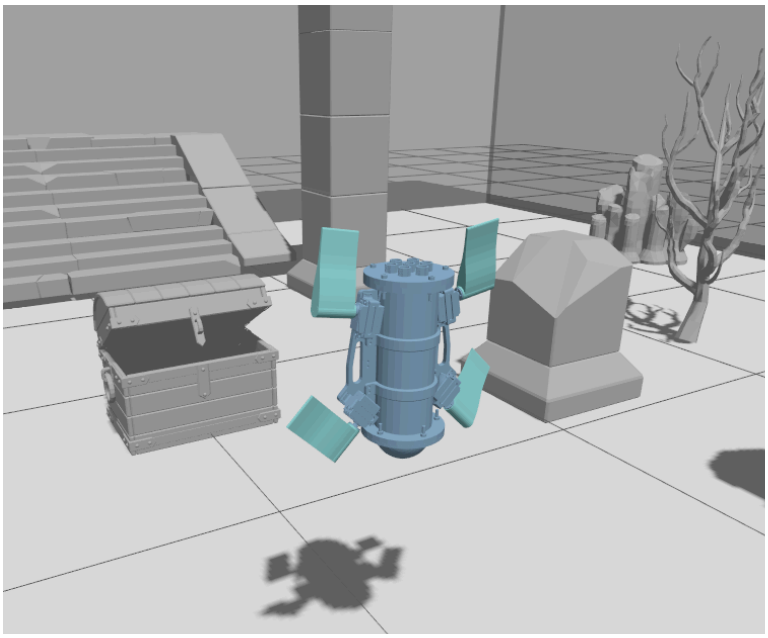

Figure 7. Rest robot

4.1.2 Joint-position-controller

The plugin [5] is used to control the angular position (Position) of a specified joint. By employing a closed-loop control algorithm (PID controller), the plugin ensures that the joint reaches and maintains the target position accurately.

Plugin configurations
To achieve control over a specific joint, the following steps and configurations are required:

- **Reference to Joint Name**

  The joint name (e.g., joint3) must be specified in the SDF file to indicate which joint the plugin controls.

- **ROS 2 Topic for Target Position**

  A ROS 2 topic (e.g., joint3_move) needs to be defined to receive the target position commands.

- **PID Parameter Configuration**

  The plugin requires tuning of the PID parameters to achieve optimal control:

  - p_gain (Proportional gain): Determines the system's responsiveness to position error.
  - i_gain (Integral gain): Addresses steady-state error over time.
  - d_gain (Derivative gain): Dampens rapid changes to prevent overshooting.

  Proper tuning ensures:

    - Faster response times.
    - High precision.
    - System stability.
- **Command Signal Limits**

  cmd_max and cmd_min are used to define the maximum and minimum control signals, preventing the application of excessive force or torque on the joint.

Plugin Workflow
After subscribing to the specified topic, the plugin computes and applies control signals to the selected joint using the PID controller formula:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

Where:

- e(t): The error between the current joint position and the target position.
- Kp,Ki,Kd: The proportional, integral, and derivative gains.

The computed control signal u(t) is then applied to the joint to adjust its position until the error approaches zero.

Joint Limit Settings

In addition to configuring the plugin, it is essential to properly set the limit parameters for the joint in the SDF file. These parameters ensure that the joint operates within realistic and safe boundaries (If these parameters are not set, <effort> and <velocity> default to 0, causing the joint to remain stationary, as no motion or force is allowed.).

- <lower> and <upper>:

  Define the joint's motion range.

  Prevent the joint from moving beyond its physical or design limits.
- <effort>:

  Specifies the maximum allowable force or torque the joint can apply.

  Protects the joint from excessive loads.
- <velocity>:

  Limits the maximum speed of the joint.

  Ensures smooth and realistic movement.

4.1.3 Thruster

After applying the Hydrodynamics and Joint-position-controller plugins, the robot's fin movements did not produce the expected forward motion, and errors were observed in the simulation.

This indicated that the Hydrodynamics plugin could not fully simulate the force generated by water on the robot, particularly the propulsion force created by the fin oscillation.

To accurately simulate the force exerted by the water on the robot (reactive thrust), the thruster plugin was introduced. This plugin provides realistic dynamic behavior for propulsion systems, enabling the robot model to move effectively in the simulation [3]. It is especially suitable for underwater robots, ships, and other applications requiring thrust.

As described in Assignment1, the propulsion force is generated by the oscillating motion of the fins, producing a reactive force from the water perpendicular to the surface of the fins (as illustrated in Figure 8). To simulate this propulsion force, the thruster plugin is utilized to generate the reactive thrust caused by fin movements.

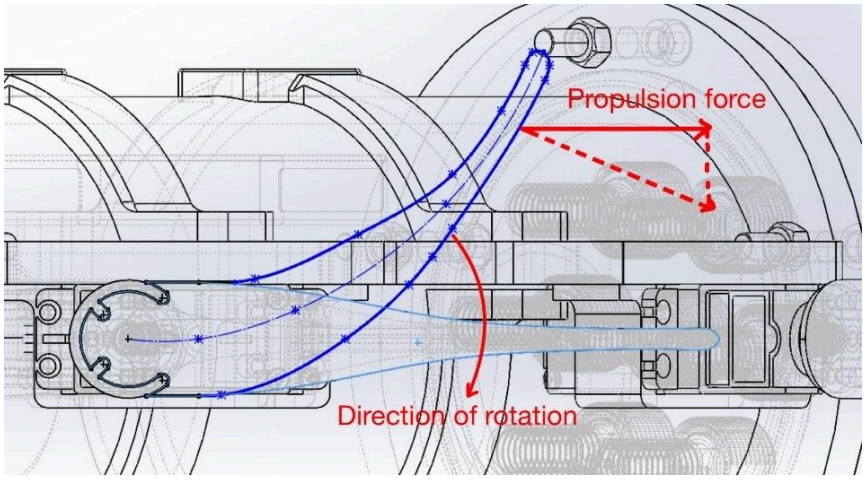

Figure 8. Propulsion force

Implementation

- **Modeling**

  In the SDF file, define the thruster model using simple cylindrical geometry.

  Assign the thruster appropriate mass, inertia, and connect it to the robot model through a joint.

- **ROS 2 Topic:**
  Bind the joint_name (e.g., pp2_joint) to the plugin and subscribe to ROS 2 thrust command topics.
  The thruster plugin automatically generates a control topic (/model/robot1/joint/pp2_joint/cmd_thrust) that does not require manual definition but must be bridged to Gazebo in the launch file.

Reaction Torque

After adding thrusters to four fins, the robot could produce forward propulsion in the simulation. However, it also exhibited rotation along the **Roll axis**. The Gazebo tutorial does not provide an explanation for this phenomenon.

Upon analysis, it was determined that this issue arose due to the **reaction torque** generated by the rotating propellers, which was not balanced between the fins.

After analysis, it was determined that this issue was caused by the reaction torque generated by the rotating propellers. Since all four propellers rotated in the same direction, the reaction torques could not cancel each other out.

To resolve this issue, we changed the thrust_coefficient of the thrusters on one side to a negative value. This allowed the propellers on that side to rotate in the opposite direction while maintaining the same thrust direction, thereby canceling out the reaction torques between the left and right sides.

After re-running the simulation, we observed that the robot achieved stable horizontal forward motion, and the roll-axis rotation issue disappeared.

Optimization

After confirming the position and direction of the thrusters, we reduced the size of the four propellers to minimize their impact on buoyancy. Additionally, we made them invisible to facilitate further development

5. Sensors (Teri_urdf.sdf)

5.1 Lidar (gpu_lidar)

The GPU Lidar sensor plugin simulates a LiDAR device, generating real-time 3D point cloud data of the environment for robotic navigation, obstacle avoidance, and environment perception. By leveraging GPU acceleration, this plugin enables faster simulation of laser scanning with higher resolution and update rates[6].

Position of the LiDAR
The position and orientation of the LiDAR relative to the robot's reference point (base_footprint) must be defined using the <pose> tag.
It is worth to note that the LiDAR should not be placed inside the robot's body, as its emitted laser beams could be blocked by the robot's shell, resulting in incorrect or incomplete data.

LiDAR Parameter Configuration

- **Update Rate (<update_rate>)**
  This parameter determines how many laser scans the LiDAR generates per second. For low-dynamic environments or limited computational resources, a lower value is recommended (e.g., 10 scans per second are used in this project due to computational constraints). In high-dynamic scenarios, this value can be increased to capture more frequent updates.
- **Scanning Parameters (<scan>)**
  The <scan> tag defines the horizontal and vertical scanning behaviors:
  - Horizontal Scan (<horizontal>): Defines the resolution (<samples>) and the angular range (<min_angle> and <max_angle>).
  - Vertical Scan (<vertical>): Similarly defines resolution and angular range in the vertical direction.

By combining horizontal and vertical scans, the LiDAR can generate comprehensive point cloud data (Figure 9).

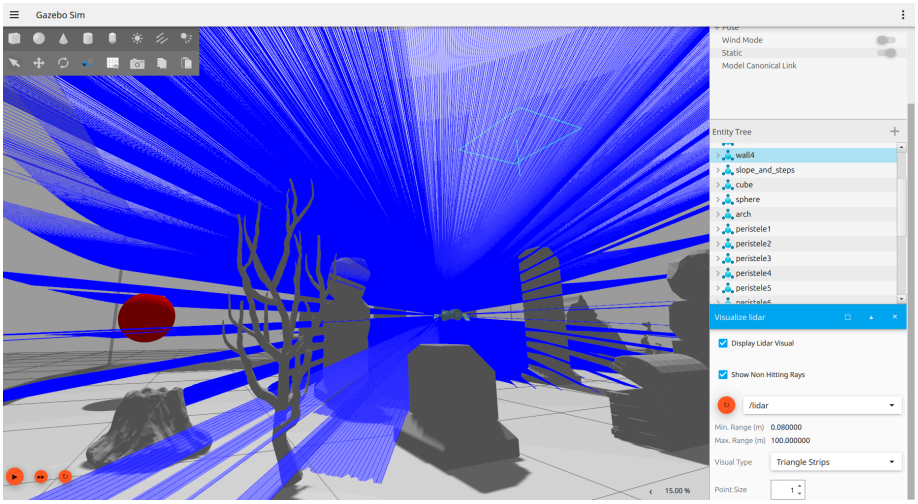

Figure 9. Lidar hitting rays

- **Range Settings (<range>)**
  These parameters set the minimum and maximum distances the LiDAR can measure, along with the resolution of the measurements. Configuring these parameters ensures the LiDAR's behavior closely matches that of real-world sensors.

Data Publishing Topic

The LiDAR publishes its scan data to a ROS 2 topic, specified as lidar(/lidar/points) in this configuration.

The robot's controllers or algorithms can subscribe to this topic to receive and process LiDAR data for tasks such as navigation or mapping.

To enable data visualization, we use the ros_gz_bridge node to convert Gazebo's internal data format (gz.msgs.PointCloudPacked) into ROS 2's standard format (sensor_msgs/PointCloud2). RViz subscribes to the LiDAR point cloud data topic (/lidar/points) and displays the 3D environment in real time. This provides an intuitive visualization of the LiDAR output, facilitating the verification and debugging of LiDAR functionality.

After completing the above steps, as shown in Figure 10, the point cloud on the left clearly displays the arch and two stone columns on the right side of the actual environment, indicating that the LiDAR successfully generates a map of the world.

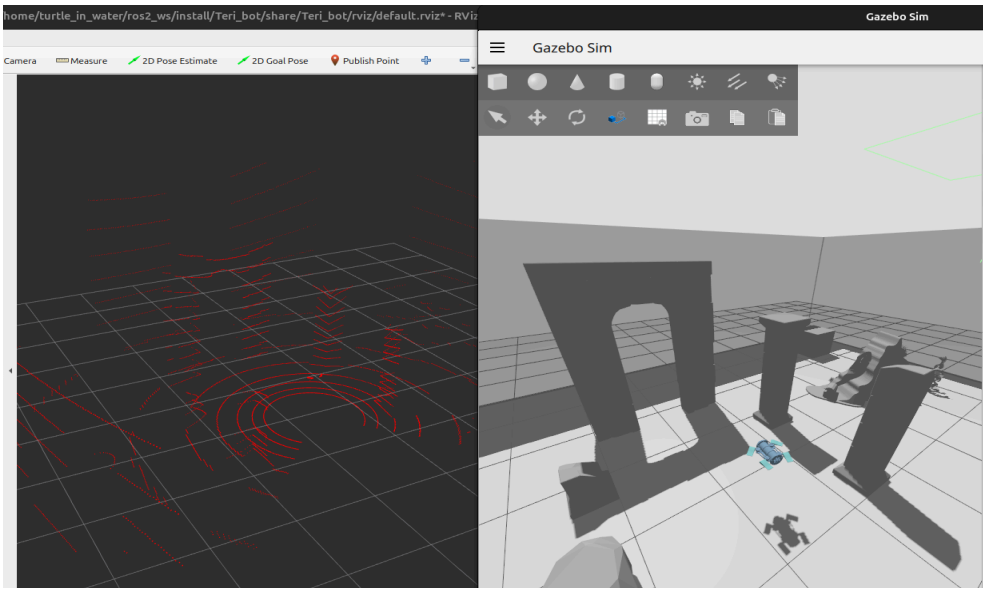

Figure 10. Comparison of point cloud and actual environment

5.2 Camera

The boundingbox_camera plugin is a camera sensor plugin primarily used in Gazebo simulations to simulate object detection functionality and capture images of the environment [7].

Position of the Camera

Similar to LiDAR, the camera's relative position must be defined. It should also not be placed inside the robot's body to prevent its field of view from being obstructed by the robot's structure.

Camera Parameter Configuration

The plugin supports configuration of parameters such as the field of view, resolution, and clipping distances, allowing it to closely match the characteristics of a real-world camera.

Data Publishing Topic

A topic needs to be specified for publishing the camera data. In this configuration, the topic is defined as camera1_image. The sensor publishes data to this topic, which can be subscribed to by RViz (Figure 11) or other nodes for visualization and processing.

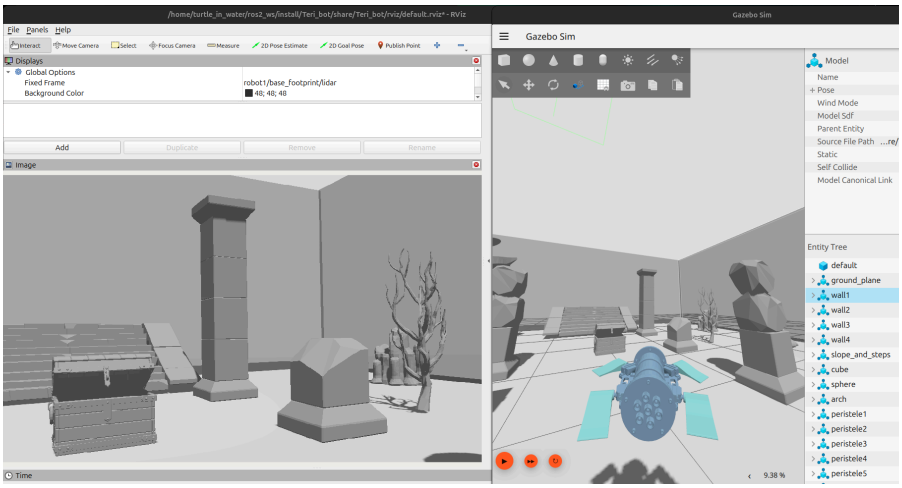

Figure 11. Camera view compared to the actual environment

6. Navigation Algorithm(joint_controller.py)

6.1 Code Structure

First, a class named JointController is defined, inheriting from ROS 2's base class Node. This class is responsible for subscribing to sensor data, making decisions, and publishing control commands. The following functions are included in the class:

1. __init__(self)

- Purpose:
  Constructor function used to initialize the node joint_controller, set up subscribers, publishers, and initial parameters.
- Key Functions:
  - Subscribe to LiDAR point cloud data:
    Subscribes to the /lidar/points topic to receive point cloud data.
  - Create joint control and thrust publishers:
    Publishes joint movement and thruster thrust values to relevant topics.
  - Initialize parameters:
    Includes thrust values, obstacle distance thresholds, etc.
  - Set up timers:
    Creates a timer for periodically updating joint movements.

2. set_thrust(self, forward=False, left=False, right=False)

- Purpose:
  Sets the thrust values for the actuators and publishes them to the corresponding ROS 2 topics.

3. timer_callback(self)

- Purpose:
  Timer callback function used to periodically update the oscillation state of the robot's joints

4. point_cloud_callback(self, msg)

- Purpose:
  Callback function to process subscribed point cloud data, implementing obstacle detection and general logic(Described in detail in the following section).
- Key Functions:

  Parse and filter point cloud data:
  Extracts coordinates from the point cloud data and filters points

  Detect obstacles:
  Identifies obstacles within the robot's surroundings.

  Update movement state:
  Based on the obstacle distance, determines whether to turn or continue moving straight.

6.2 General logic

By importing sensor_msgs_py.point_cloud2, the point_cloud_callback function is executed each time the JointController node receives a new message from the /lidar/points topic. Therefore, to promptly respond to changes in the environment, the control logic of this project is primarily implemented within the point_cloud_callback function. Figure 12 illustrates the detailed logic of this function:

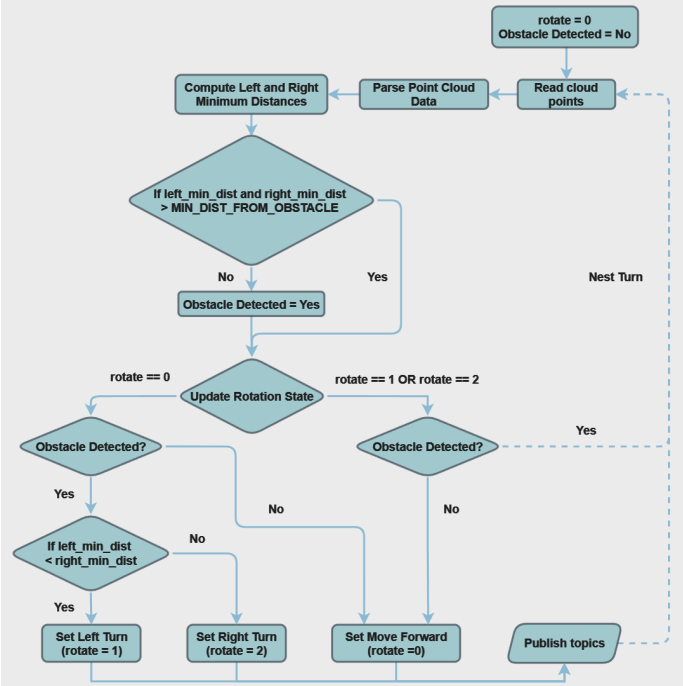

Figure 12. Logic of the **point_cloud_callback** function

6.2.1 Data Processing and Filtering

First, LiDAR data needs to be further processed to identify the relative positions of obstacles in the environment.

- **Point Cloud Data Parsing**

  The read_points function is used to parse PointCloud2 messages.

  Note: Although the definition of LiDAR involves an angular range (<min_angle> and <max_angle>), the returned values are not in polar coordinates but in (x, y, z) coordinates.

- **Data Filtering**

  Horizontal Scanning Range:

  The minimum horizontal scanning range is determined using the robot's width (robot_front_width) and the MIN_DIST_FROM_OBSTACLE threshold.

  The effective horizontal range is calculated to identify potential obstacles that fall within the robot's movement area.

  Points outside the calculated horizontal scanning width (±robot_front_width / 2) are discarded. (Figure 13).

  Vertical Scanning Range:

  Similarly, a vertical range is defined based on the robot's height (robot_front_height), ensuring detection is limited to obstacles that are within the robot's frontal vertical field of view.

  Points outside the calculated vertical height (±robot_front_height / 2) are excluded (Figure 13).

  Distance Threshold:

  Any points where the distance (x) is greater than MIN_DIST_FROM_OBSTACLE are considered safe and filtered out.

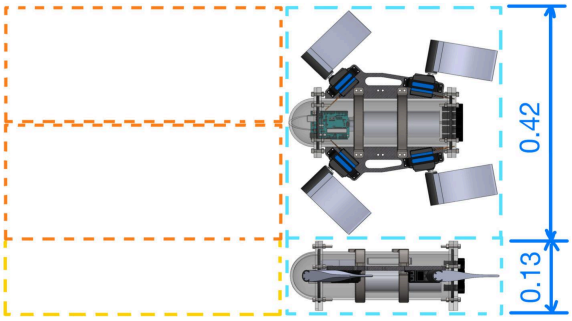

Figure 13. Minimum horizontal scanning range

6.2.2 Obstacle Avoidance

The algorithm iterates through all valid scanned distances on both the left and right sides, comparing each distance with the current minimum distance for that side to select the smaller value:

- Left Side:
  Compute the minimum distance for points where $y > 0$.
- Right Side:
  Compute the minimum distance for points where $y \leq 0$.

6.2.3 Motion State Updates

The robot's motion state is updated based on the minimum distances to obstacles on the left and right sides:

- Move Forward:
  If the minimum distances on both the left and right sides are greater than the threshold MIN_DIST_FROM_OBSTACLE, the robot continues moving forward.
- Obstacle Avoidance:
  If the minimum distance on either side is less than the threshold:
  - If the left-side distance is smaller

    The robot turns right. The right fin will stop swinging, and the left fin will increase the swing amplitude, so that the left and right forces of the robot are inconsistent, and the differential rotation is achieved.。

  - If the right-side distance is smaller,

    The robot turns left.

6.2.4 State Machine

Testing has shown that without state memory during decision-making, the robot may get trapped when obstacles are present on both sides with minimal difference in distance (as illustrated in Figure 14).

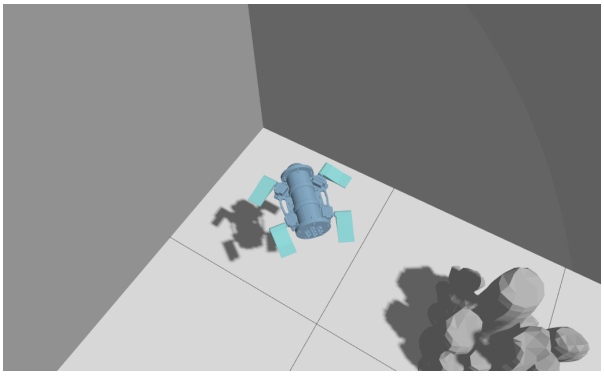

Figure 14. The trapped robot

To address this issue, a variable rotate is introduced to track the robot's rotational state. This method allows the robot to adjust its rotational behavior based on both the previous state and the current position of obstacles. The optimized state update logic is as follows:

- **No Rotation (0)**:
  - If there are no obstacles in front, the robot continues to move straight.
  - If an obstacle is detected, the robot stops forward thrust and selects a rotation direction based on the relative position of the obstacle. The rotate state is updated accordingly to reflect this change.
- **Rotate Left (1)**:
  - The robot continues turning left until there are no obstacles in front(minimum distances on both sides are greater than the threshold distance).
  - Once the path is clear, the rotate state is reset to 0, and the robot resumes straight movement.
- **Rotate Right (2)**:
  - The robot continues turning right until no obstacles are detected in front(minimum distances on both sides are greater than the threshold distance).
  - Once the path is clear, the rotate state is reset to 0, and the robot resumes straight movement.

This state memory mechanism ensures that the robot can respond continuously and consistently when encountering obstacles. After avoiding obstacles, it can effectively resume straight-line motion. By maintaining the rotate state, the robot's behavior becomes more intelligent, improving both the efficiency and safety of obstacle avoidance.

6.2.5 Autonomous Navigation Result

After implementing the above algorithm on the robot, it successfully achieved autonomous navigation in different testing scenarios and was able to drive autonomously to a previously defined object.

In Scenario 1 (Figure 15), the robot effectively explored the entire map through continuous autonomous obstacle avoidance, successfully avoiding collisions while ultimately locating the defined object.

After placing obstacles in different positions and adjusting the robot's starting point (Figure 16), the robot was still able to autonomously navigate, avoid collisions, and ultimately reach the defined object.

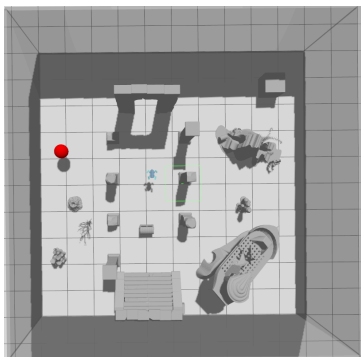

Figure15. scenarios1

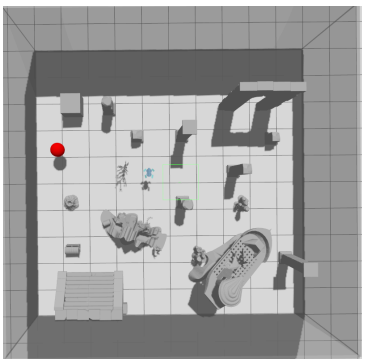

Figure16. scenarios2

7. Object identification(object_recognizer.py)

The entire system captures images using the boundingbox_camera sensor in the Gazebo simulation environment. This camera is configured with a specific field of view, resolution, and update rate. The image data is transmitted through the ROS 2 topic /camera1_image. On the receiving end, the ObjectRecognizer node uses CvBridge to convert ROS image messages into OpenCV format for processing. At the same time, the processed images are displayed in real time, with detected red circular targets annotated. The entire system achieves real-time image acquisition, processing, object recognition, and information publishing.

7.1 Node Architecture Design

7.1.1 Node Implementation

The system uses the ROS 2 framework to implement object recognition functionality by creating an independent object_recognizer node for image processing and object detection. This node inherits from ROS 2's Node base class and adopts an object-oriented design approach to realize a modular system architecture.

7.1.2 Communication Mechanism

The system adopts the Publish-Subscribe communication model of ROS 2 to achieve loosely coupled communication between nodes. The specific communication architecture is as follows:

Subscribed Topic

- /camera1_image: Used to receive real-time image data streams from the Gazebo simulation environment. The image format follows the standard ROS 2 image message type.

Published Topics

- /red_circle_detected: Publishes Boolean messages indicating whether a red circular target is detected in the current image frame.
- /red_circle_info: Publishes string messages containing the Center coordinates (x, y) and Radius of the detected circular target.

7.2 Image Processing and Recognition

A red sphere was selected as the target object due to its rotational symmetry, ensuring a consistent circular appearance across all 2D viewpoints. The sphere was rendered in red for robust color-based identification, as illustrated in Figure 17.

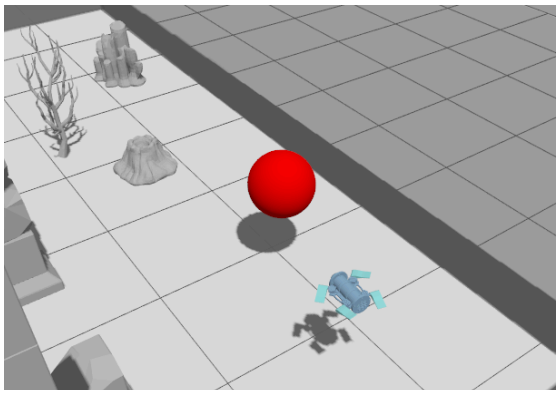

Figure 17. TERI with target object

The implemented object_recognizer node is specifically designed to detect red circular shapes. When the node is running, a new window pops up displaying the detection result in real time, as illustrated in Figure 18. The operation process is as follows:

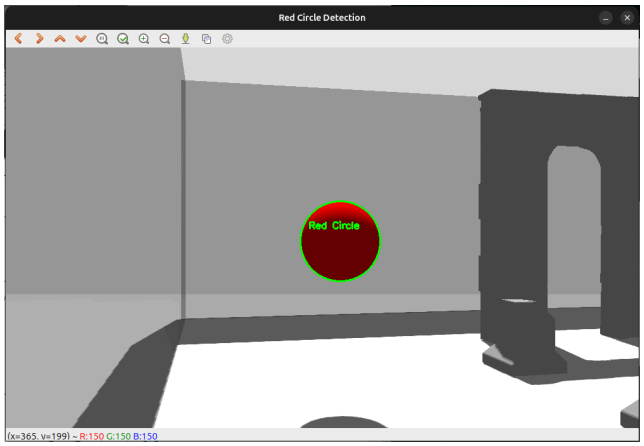

Figure 18. Image of target object

7.2.1 Image Conversion

The node subscribes to the /camera1_image topic to retrieve image frames published by the simulation environment. Incoming image messages will be converted from ROS 2 message types to OpenCV-compatible format using the cv_bridge utility at first.

7.2.2 Color Space Conversion

Then the image is converted to HSV color space, which is more suitable for color detection. HSV (Hue - Saturation - Value) is a color model that represents color using three components (Figure 19):

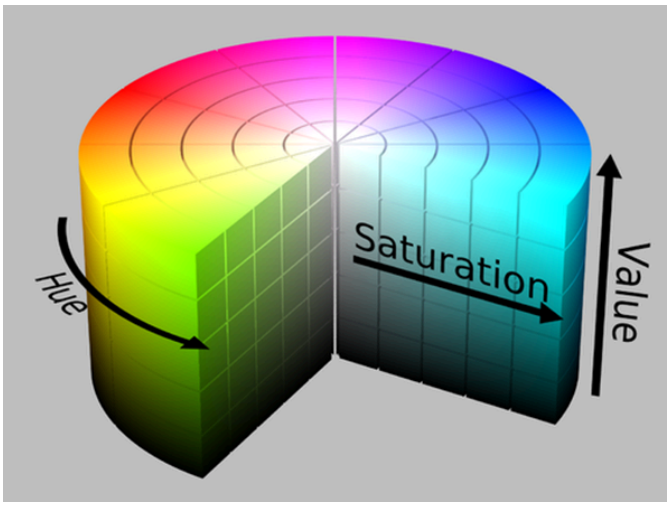

Figure 19. Three components of HSV color space

- Hue: Represents the type of color. Red is approximately at 0° and 180° in OpenCV.
- Saturation: Indicates the purity of the color, ranging from 0–255 in OpenCV.
- Value (Brightness): Describes how light or dark the color is, also ranging from 0–255.

To detect red in HSV space, two ranges are defined due to the circular nature of the hue value:

```
# First range: reddish-orange (0–10 degrees)
lower_red1 = np.array([0, 100, 100])
upper_red1 = np.array([10, 255, 255])
# Second range: reddish-purple (160–180 degrees)
lower_red2 = np.array([160, 100, 100])
upper_red2 = np.array([180, 255, 255])
```

Two ranges for red is due to the circular nature of the HSV hue scale:

The hue component in HSV is circular, ranging from 0° to 360°. Pure red is located at both ends: around 0° and 360°. This makes red span both extremes of the hue scale (illustrated in Figure 20).

Figure 20. Hue in HSV

- Around 0°: orangish-red
- Around 360°: purplish-red

It is worth noting that in OpenCV, the hue values are compressed to the range 0–180 to fit within 8-bit storage.

Why Two Ranges Are Needed

- If only one range is used (e.g., 0–10 degrees), red hues near 180 degrees will be missed.
- If a large range is used (e.g., 0–180 degrees), many non-red colors will be falsely detected.
- Using two ranges allows:
    - Accurate capture of all red hues
    - Avoidance of detecting other colors
    - Improved detection accuracy

After defining the two ranges, the two masks are combined using cv2.bitwise_or.

7.2.3 Image Preprocessing

Before object recognition, it is necessary to denoise the image. Morphological operations—opening and closing—are used to remove noise.Opening and closing are composed of two basic morphological operations: erosion and dilation.

Erosion

Principle:

A structural element (kernel) slides across the image. Only when the kernel is fully contained within the target region is the center pixel retained.This causes object boundaries to shrink inward (Figure 21).

Figure 21. Comparison before and after erosion processing

Main effects[9]:

- Eliminates edge pixels

- Shrinks target regions
- Breaks thin connections
- Removes small noise points

Dilation

Principle:

The kernel slides across the image.If the kernel overlaps with any part of the target region, the center pixel is marked.This causes object boundaries to expand outward (Figure 22).

Figure 22. Comparison before and after dilation processing

Main effects[9]:

- Fills small holes
- Enlarges target regions
- Connects nearby objects
- Smooths object boundaries

Based on these basic operations, two compound morphological operations are used for denoising:

Opening

Process: erosion followed by dilation (Figure 23)

Figure 23. Comparison before and after opening processing

Main functions[9]:

- Remove small noise points
- Break thin connections
- Smooth object outlines
- Preserve the main geometric features of the object

Closing

Process: dilation followed by erosion (Figure 24)

Figure 24. Comparison before and after closing processing

Main functions[9]:

- Fill small holes
- Connect nearby objects
- Smooth object outlines
- Preserve the main geometric features of the object

7.2.4 Contour Detection

cv2.findContours()is used to detect contours in the red mask region.For each contour, perform the following analysis:

- Calculate the contour area and filter out regions that are too small (area > 10).
- Compute the contour's **circularity** using the formula:
  circularity = 4 * np.pi * area / (perimeter * perimeter)
- Contours with circularity values close to 1 (within the range of 0.7 to 1.3) are considered circular.

8. SLAM

SLAM (Simultaneous Localisation and Mapping) is used to build a map of an unknown environment whilst simultaneously localising the robot. This is important for underwater autonomy in an unknown environment with a number of obstacles.

8.1 Packages

slam_toolbox was chosen as the core engine used for the SLAM to build and update a map creating a 2D occupancy grid.

robot_state_publisher, publishes the robots TF frames based on the URDF including base_footprint, lidar_base and joint states.

ros_gz_sim simulates the underwater environment and robot configuration from the SDF model.

ros_gz_bridge bridges the simulation topics, /scan, /odom, /map and /tf, to ROS2

Teri_bot custom nodes

slam_continous_updater, monitors the map updates and initiates loop closure and manual SLAM updates if required.

tf2_ros publishes essential transforms including base_footprint -> lidar_base via the static_lidar

rviz2 is used as a visual tool in order to validate the SLAM performance in real time.

8.2 SLAM Workflow

The SLAM work flow can be visualised in **Figure** The simulation is set up using ros_gz_sim which launches Gazebo including the Atlantis.world features and spawns the robot via the create node. Then the robot_state_publisher and static_transform_publisher broadcasts the transforms for the robot base and sensor frames. roz_gz_bridges then relays the Gazebo LiDAR, odometry and map data to ROS2 topics including /scan, /odom and /map. slam_toolbox is then loaded using online_async_launch.py entry point and relays the configured slam_params.yaml including key parameters

- map_frame : map
- odom_frame: odom
- base_frame: base_footprint
- scan_topic: /scan
- resolution : 0.05
- do_loop_closing : true

The custom slam_continuous_updater node tracks the scan and map update intervals. If no update occurs after a set number of scans a manual loop closure is triggered via service. Rviz opens after a 3 second delay using the slam.rviz config enabling real-time monitoring of the map creation, pose estimate and scan alignment.

8.3 Visualisation in RViz

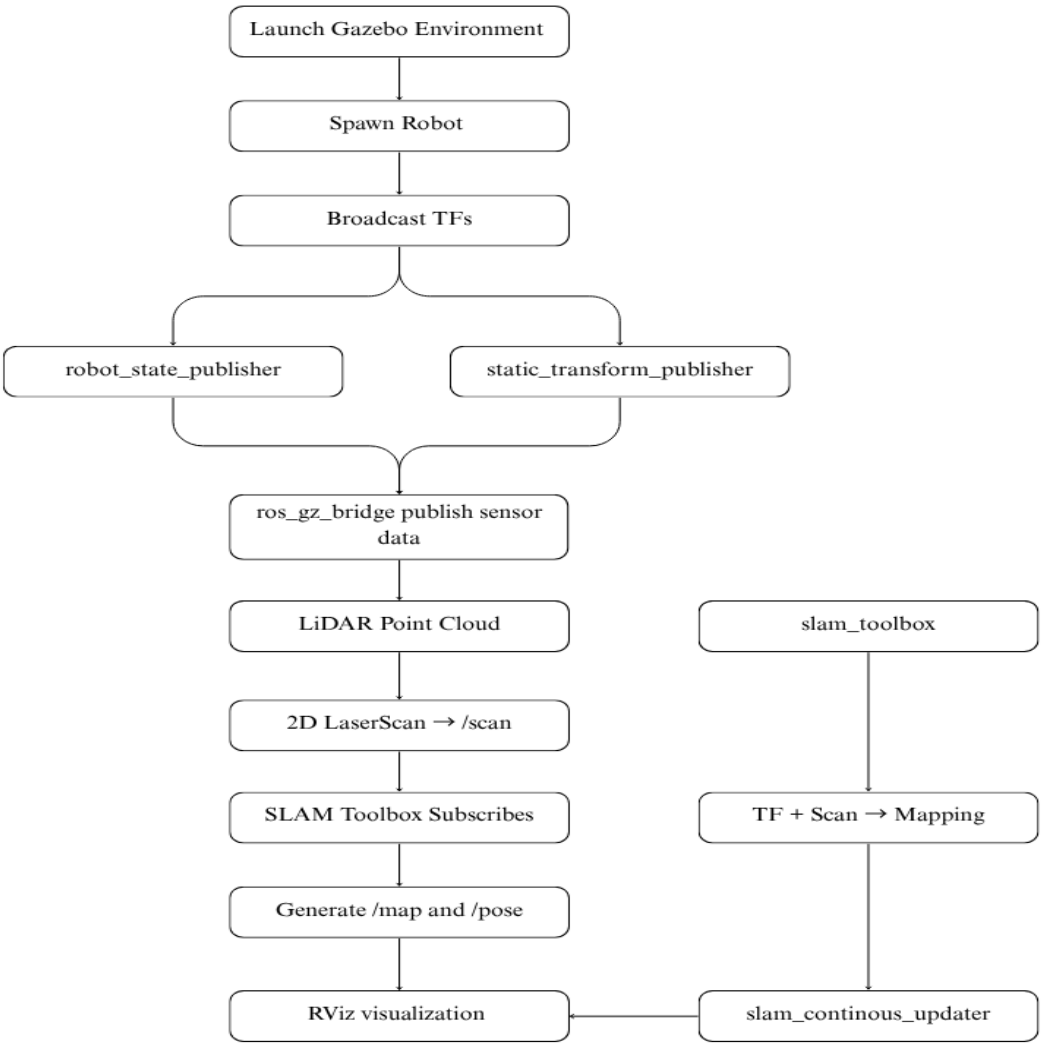

Figure 25. Visualisation pipeline in RViz

RViz is used to display /scan information for the live laser scan from the LiDAR. /map is used to create a SLAM generated map. /tf produces the robot frame tree. /pose is used to estimate the robots location within the environment. These displays ensure data alignment and ensure transformation correctness through the mapping and localisation processes.

8.4 Testing and Results

8.4.1 Test Scenario 1 - Obstructed Start Near Obstacles

In scenario 1(Figure 26), the robot was positioned in an obstructed area by a wall with a number of obstacles, in the top left quadrant of the simulation.

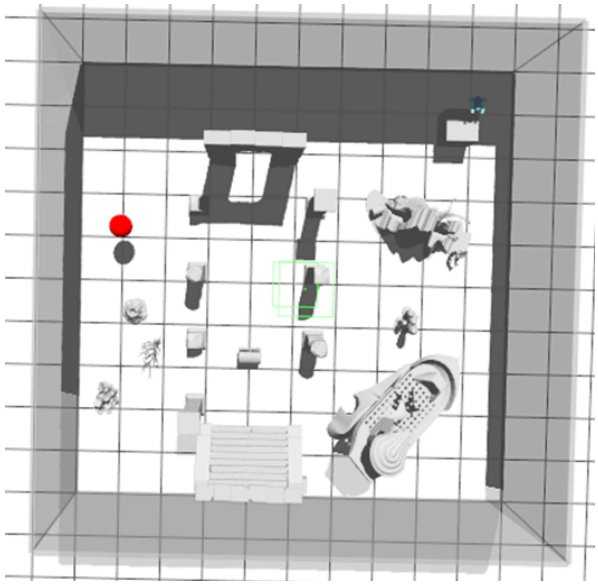

Figure 26. scenario 1

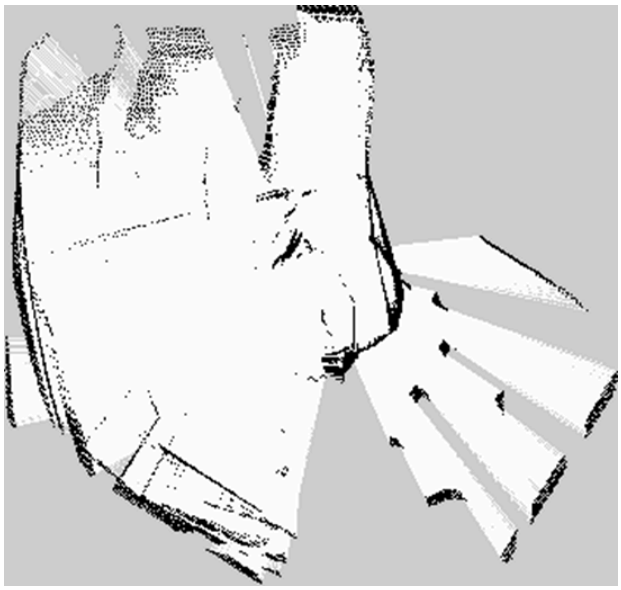

Figure 27. Generated map of scenario 1

The SLAM map shows a segmented arc and inconsistent overlays **(**Figure 27**)**. Mapping took place of the immediate surrounding but mapping is sparse and fragmented beyond the initial radius. Map quality is incomplete and noisy. Localisation stability is lower in accuracy due to a limited number of distinguishable features and poot scan match quality.

8.4.2 Test Scenario 2 - Clear Central Start

In the scenario 2 (Figure 28), the robot started in a more open centralised location, allowing for a clearer field of view

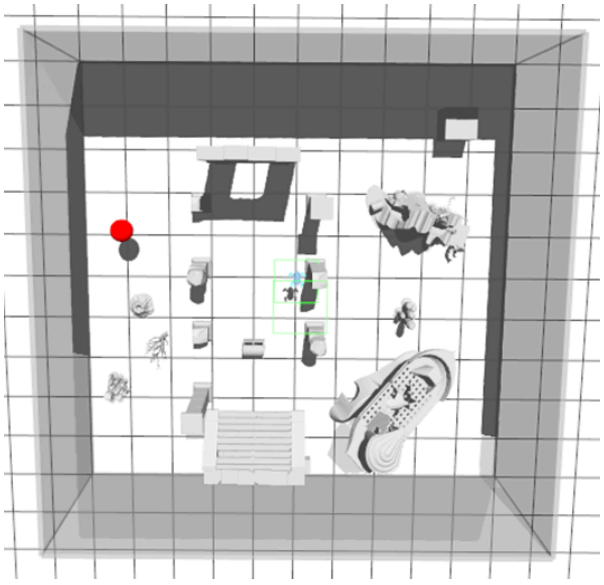

Figure 28. scenario 2

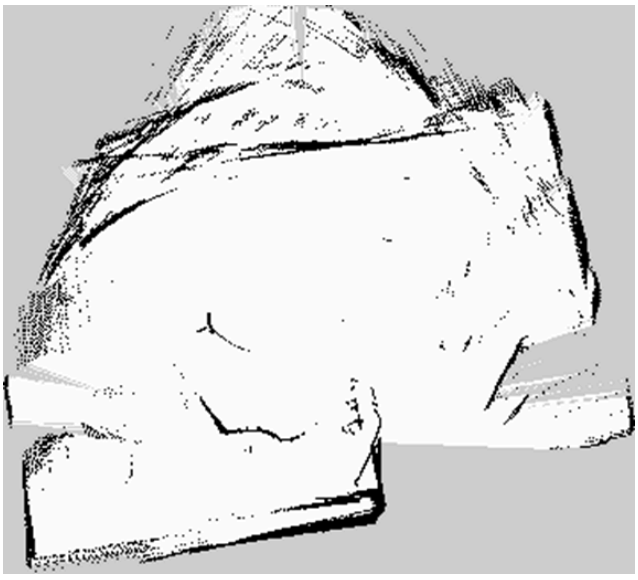

Figure 29. Generated map of scenario 2

The generated map (Figure 29) shows more structure, with overlapping scans forming a better outline of the environment, forming coherent outlines of the environment. Map quality has fewer gaps and better alignment. Localisation stability is stronger with better scan matching and pose estimates

8.5 Current Issues

Although the SLAM system is running and updating the /map topic there are some persistent issues that are affecting the accuracy and consistency of the final occupancy grid

1. **Map Overwriting and Loss of Prior Data**
   The SLAM system builds the map in real time however the generated map doesn't retain previously mapped data reliably. Currently it overwrites and degrades earlier regions, seen most commonly when the robot returns to regions that have been previously mapped. This results in
   - Blurred and fragmented maps (Figure 30)
   - Loss of structure over time
   - Incomplete maps after full scans being completed

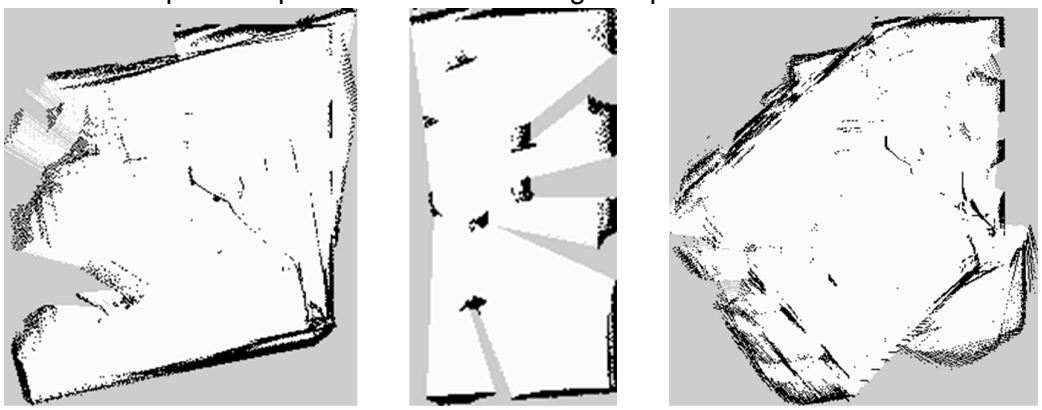

Figure 30. Blurred and fragmented maps

This could be the result of inconsistent and drifting TF frames, poor odomentry, issues with SLAM loop closure or slam_toolbox interpreting re-entered areas anew due to timing or transform problems.

2. **Red Scan Line in RViz**
   A red laser scan appears in RViz during mapping (Figure 31) which indicates a scale or incorrectly transformed scan relative to the robot base frame, duplicate scans arriving out of sign or a TF mismatch.

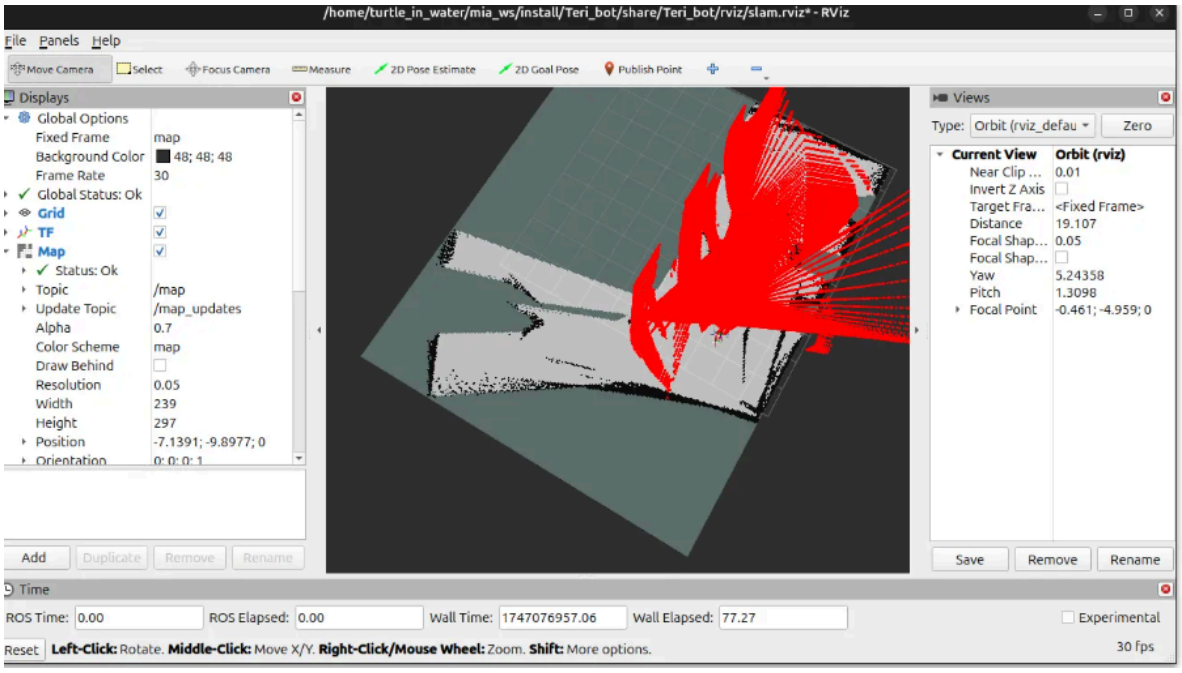

Figure 31. Laser scan

8.6 Future Improvements

In order to finalise the mapping, TF refinement is required to enable additional static transforms if any issues occur with misalignment. Implementing auto saving of maps to trigger the map ti automatically save once a certain threshold has been reached. At the current time the map builds but overwrite data from previous scans therefore this issue needs to be addressed.

9. Launch File (launch.py)

In order to initialize and startup various components of a robotic system, launch.py is designed to serve as the launch file. The script executes multiple nodes to successfully run the robotic simulation. The nodes includes:

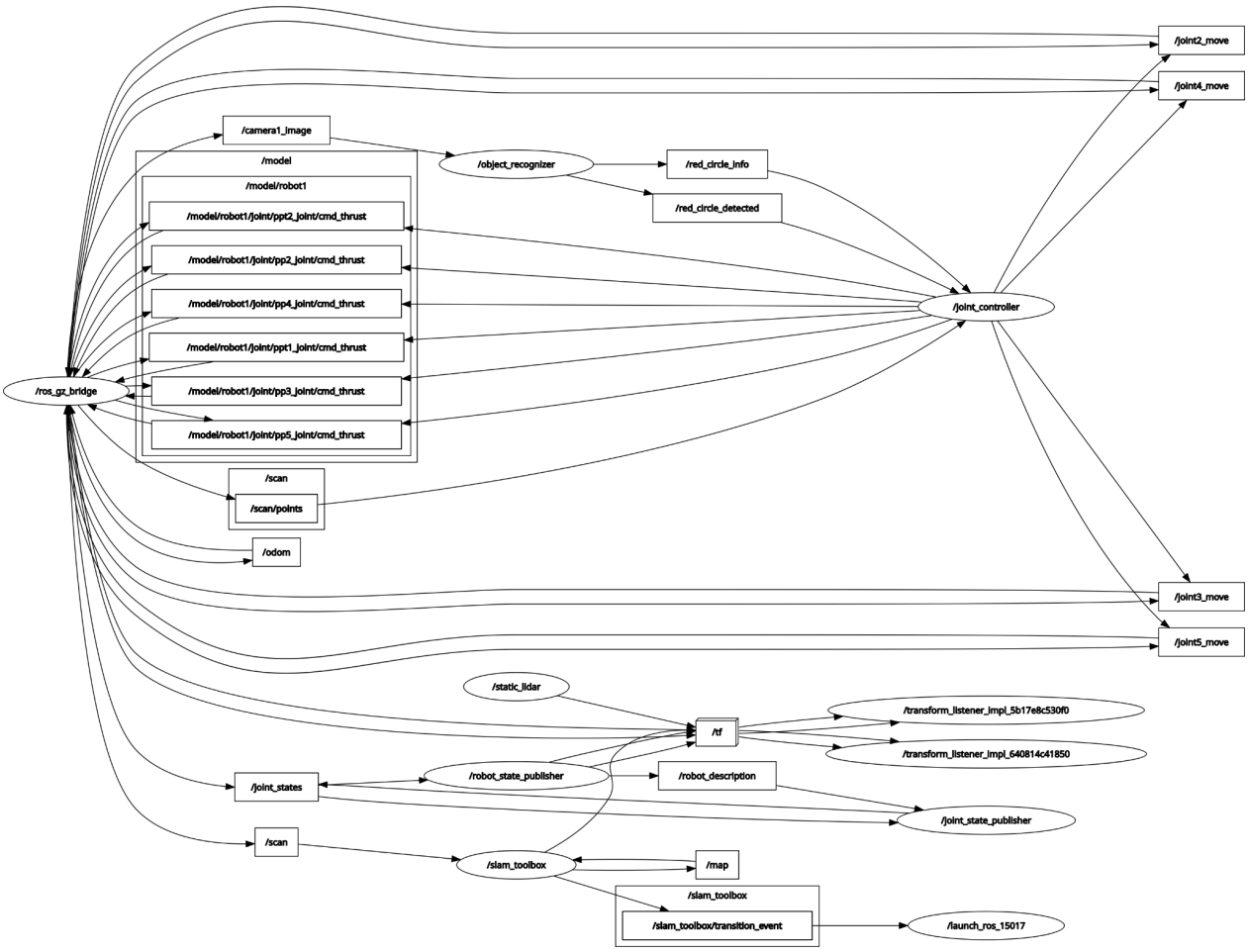

Figure 32. rqt_graph

1. ros_gz_bridge

- It serves as a bridge between ROS 2 and Gazebo, enabling the transmission of topics so that ROS 2 nodes can receive sensor data from Gazebo and joint_controller can control the actuators in Gazebo.

2. joint_controller

- Responsible for controlling the robot's joints to precisely move the robot, including actuating the fins and executing the obstacle avoidance logic in the navigation system.

3. object_recognizer

- It receives real-time images from the camera and processes each frame for recognition. Once a target is successfully identified, it publishes topics to the joint_controller to stop the robot's movement.

4. slam_toolbox

- Takes the input 2D laser scan from LiDAR from the /scan topic and TFs and converts these to output /map and /pose in order to construct an occupancy map and to estimate the pose of the robot.
- Is not currently correctly integrated into the system causing issues with mapping

5. robot_state_publisher

- Broadcasts a static frame relationship between the map and odom.

7. static_transform_publisher (base_link_to_map_publisher, map_to_odom_publisher, laser_to_map_publisher

- establishes a static frame relationship between the map, odom, base_link and laser to ensure the correct TF tree is created

The launch file serves as the foundation of TERI's testing in a simulated environment by integrating sensor data, visualization, and joint control.

10. Reflection

Environment Construction

The Atlantis.world simulation environment was successfully developed to meet the assignment's specifications. The environment included a variety of obstacles, such as slopes, cubes, and spheres, as well as additional elements like shipwrecks, treasure chests, and pillars, which provided a realistic underwater exploration setting. Custom STL models were designed and imported using SolidWorks to enhance the environmental complexity and fidelity. Furthermore, Gazebo's buoyancy and hydrodynamics plugins were integrated to achieve a highly accurate simulation of underwater physical dynamics. This project facilitated the development of practical skills in designing and simulating complex physical environments, balancing functional requirements with environmental realism. Future work will focus on incorporating dynamic obstacles and moving targets to evaluate the robot's navigation capabilities in more complex scenarios.

Robot Motion Control

The project successfully implemented essential motion control mechanisms by integrating the joint position controller and thruster plugins. The joint position controller provided precise joint manipulation via PID control, while the thruster plugin simulated propulsion forces and solved issues such as rolling caused by reaction torque, ensuring motion stability and efficiency. This aspect of the project reinforced expertise in developing and optimizing control systems for complex and dynamic environments. To further enhance the robot's motion control, future iterations will include an IMU sensor to improve attitude detection and system robustness.

Autonomous Navigation

Autonomous navigation was achieved using LiDAR for environmental mapping, enabling the robot to perform effectively in various testing scenarios. The robot successfully avoided obstacles and reached predefined targets. Additionally, a state memory mechanism was introduced to address scenarios where the robot risked becoming trapped in confined spaces, thereby improving its navigation reliability. This project provided valuable insights into the design and implementation of efficient navigation algorithms, with a focus on balancing obstacle avoidance and goal-oriented behavior. Future enhancements will include optimizing path planning algorithms for complex environments, incorporating dynamic obstacle detection and prediction, and refining map generation techniques to improve real-time accuracy and responsiveness.

Object Detection
Object detection posed a significant technical challenge during the project. While the robot successfully identified the target object using OpenCV by converting ROS image formats to OpenCV and back, it was unable to autonomously capture images of the detected object due to time and computational constraints. This experience highlighted the importance of integrating and bridging diverse software frameworks for advanced robotic functionalities. Future improvements will aim to enable automatic image capture and expand the detection capabilities to accommodate more complex object characteristics.

Environmental Mapping

Environmental mapping was a critical part of this project and was very technically demanding. While the system launched the SLAM node and established the required transformations within the simulated Gazebo environment. However during testing the generated map exhibited issues with stability and accuracy as mapped regions were being overwritten and degraded resulting in low quality output maps. These issues are linked to the lack of persistence in odometry data, potential TF misalignments and timestamp inconsistently between scan messages and TF frames. Additionally, there is a recurring laser scan artefact present in the RViz output, suggesting that some scan data is misaligned or replayed with outdated transforms.

Despite these issues, the integration of slam_toolbox with supporting slam_continous_updater demonstrates a functional SLAM pipeline. This phase of the project has highlighted the difficulty of synchronising multiple data streams, time references and transforms in the simulation environment. Therefore, future improvements will focus on introducing a reliable odometry source, re-enabling scan conversion for 2D mapping and automating map saving to preserve the map progression.

11. Contribution

Qi Cheng:
Responsible for the robot's motion control and navigation algorithm development, implementing a robust and reliable obstacle avoidance strategy using a state machine framework. Developed and deployed the object vision recognition system, including the creation of a ROS 2 node for real-time image acquisition, HSV-based color filtering, and red circular object detection using OpenCV. Contributed to system integration by facilitating coordination between the vision and motion control modules, resulting in smooth and responsive system behavior. Also led the design of the robot's mechanical structure and hardware layout, focusing on structural arrangement, motor configuration, and sensor selection. A comprehensive Bill of Materials (BOM) was prepared to evaluate and select cost-effective components, ensuring that the overall design met performance requirements while maintaining a high cost-performance ratio.

Mia Hornett:
Led the integration of the SLAM system by configuring slam_toolbox in asynchronous online mode, supported through the creation of a custom slam_continuous_updater script and node to enhance real-time map consistency and loop closure. Designed the overall SLAM pipeline including the 3D LiDAR to 2D laser scan format using lidar_converter, ready for future integration. Validated and configured the LiDAR and camera sensor plugins ensuring correct sensor placement, frame alignment and ROS-Gazebo bridging for real-time SLAM visualisation in RVIZ. This enabled the simulation of environmental mapping in complex underwater scenarios, supporting future expansion of this project.

Vincent Tirta:
Responsible for creating the simulated Atalantis world for the robot to navigate in Gazebo. Also responsible for initializing the robot to the world. Utilized gazebo plugins, such as physics and buoyancy plugins, to ensure the robot's movement is equivalent to the real underwater physics. Achieved neutral buoyancy for the robot to travel at a constant depth by setting the mass. Configured objects in its environment to display a realistic aquatic environment. Sets boundaries to keep the robot in range, and sets the lighting to show each feature clearly.

Brissenden, Aurora:
Be responsible for introducing motion control and obstacle avoidance algorithms in bench

12. Conclusion

This project successfully developed TERI, a biomimetic underwater robot capable of autonomous navigation, object recognition, and mapping in a realistic simulation environment. Inspired by the locomotion and morphology of sea turtles, TERI integrates mechanical design, motion control, and intelligent perception within a Gazebo-based underwater environment, Atlantis.world.

A core achievement of the project lies in the seamless integration of multiple subsystems. Motion stability was achieved through careful application of thruster modeling, and joint PID control. Sensory capabilities were realized using simulated LiDAR and camera, while custom ROS 2 nodes enabled real-time object detection.The robot demonstrated autonomous navigation by processing point cloud data and following a state-machine-based strategy for stable obstacle avoidance, even in densely cluttered scenarios.

The project also introduced an initial SLAM implementation using slam_toolbox, which allowed for partial mapping and localization in unknown environments. However, challenges such as map degradation, TF inconsistencies, and scan misalignment were observed, which highlighting the need for further refinement in transform stability and odometry fidelity.

Despite these limitations, the modular and extensible system architecture lays a strong foundation for future development. Potential directions include dynamic obstacle interaction, real-world hardware deployment, motion attitude control and high-fidelity environmental mapping. Ultimately, TERI serves not only as a functional prototype of an intelligent underwater vehicle but also as a research platform for exploring the intersection of biomimetics, robotics, and autonomous systems.

Reference

[1] Gazebo Tutorials, "Hydrodynamics in Gazebo," [Online]. Available: https://classic.gazebosim.org/tutorials?tut=hydrodynamics&cat=physics.

[2] Gazebo Tutorials, "Guided Tutorials - Level 2," [Online]. Available: https://classic.gazebosim.org/tutorials?tut=guided_i2.

[3] Gazebo Tutorials, "Underwater Vehicles," [Online]. Available: https://gazebosim.org/api/sim/7/underwater_vehicles.html.

[4] L. Hong et al., "Advances in Marine Engineering Research," *Journal of Marine Science and Engineering*, vol. 10, no. 8, pp. 1049, Aug. 2022. [Online]. Available: https://www.mdpi.com/2077-1312/10/8/1049?utm_source.

[5] Gazebo API Documentation, "Joint Controllers," [Online]. Available: https://gazebosim.org/api/sim/8/jointcontrollers.html.

[6] RobotecAI, "Robotec GPULidar," GitHub Repository, [Online]. Available: https://github.com/RobotecAI/RobotecGPULidar?tab=readme-ov-file.

[7] Gazebo API Documentation, "Bounding Box Camera," [Online]. Available: https://gazebosim.org/api/sensors/9/boundingbox_camera.html.

[8] ROS Tutorials, “Converting between ROS images and OpenCV images (Python)”, [Online]. Available: https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython

[9] OpenCV Documentation, "Morphological Transformations (Python)," [Online]. Available: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html.