Chapter 3: Virtual Simulation

1. Introduction

This chapter presents the virtual simulation environment developed to test and evaluate the autonomous functionalities of the cleaning boat. As part of the preliminary research phase, two well-known robotic software platforms were studied and evaluated: **Webots** and **ROS1 Noetic**.

The decision to explore both platforms was based on their unique strengths. Webots is a modern robotic simulator with an intuitive graphical interface and built-in physics, making it suitable for rapid prototyping and visualization. ROS1 Noetic, on the other hand, is a mature and modular robotic framework that offers strong community support, extensive libraries, and integration capabilities.

After a comparative analysis of both tools, considering factors such as flexibility, ease of sensor and actuator integration, documentation availability, and long-term maintainability, **ROS1 Noetic Ninjemys was selected as the primary platform** for simulation work. This choice was primarily due to its robust ecosystem, comprehensive documentation, and seamless support for developing custom nodes, which are essential for simulating real-world autonomous behavior.

The remainder of this chapter details the simulation environment setup using ROS1 Noetic, including the boat model, navigation strategy, and waste detection process. The simulation replicates the core functionalities of the autonomous cleaning boat, which include autonomous navigation, detection and localization of floating waste, and the ability to plan and follow a trajectory to collect debris in a defined area. These functionalities were implemented and tested in the virtual environment to ensure the system performs reliably before physical deployment.

2. System Architecture

The autonomous system is structured into several interconnected components, primarily leveraging the modularity inherent in the Robot Operating System (ROS) architecture. To fully understand the different components of the system, we illustrate it in this figure, which shows a clear representation of the connection and the flow of information between the sections and subsections of this system

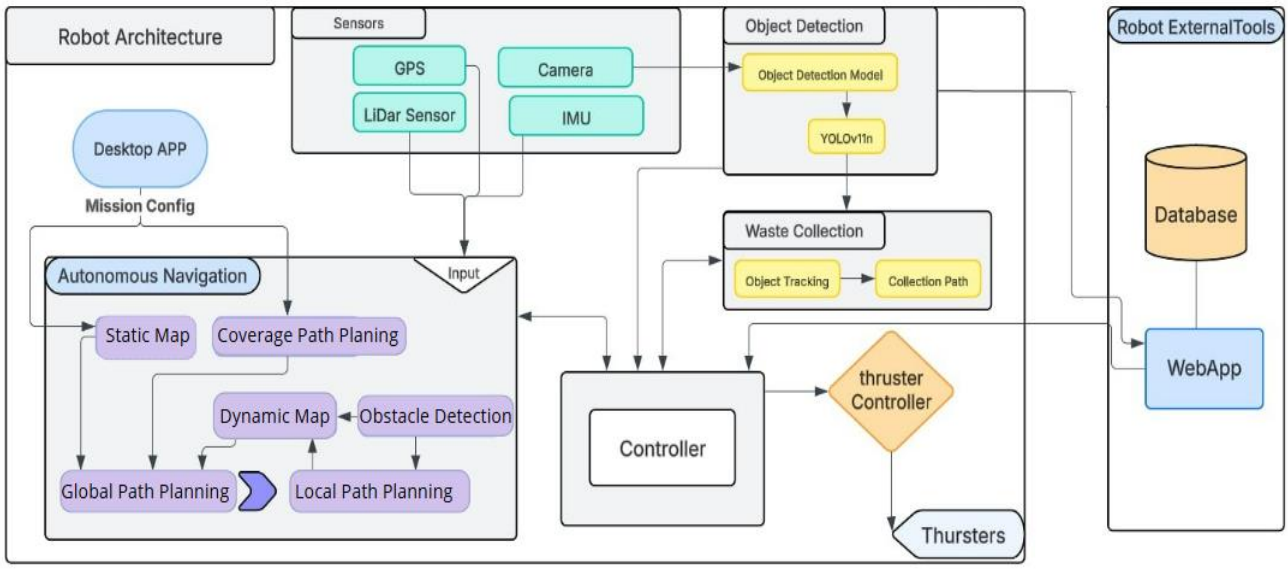


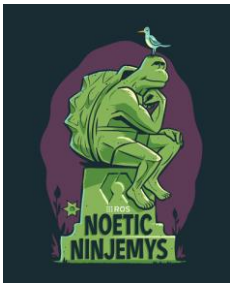
Figure 1 Robot System Architecture

Our autonomous system is divided into multiple sections or components based on ROS architecture. This will allow us to develop each section independently, allowing us to focus on the development of specific functionalities. Further in this chapter, we will talk more about the different sections and components used in this system

3. Robot Operating System (ROS) Implementation

3.1. ROS Framework and Core Packages

ROS (Robot Operating System) is an open-source framework designed to simplify the development of complex and modular robotic systems. It offers essential tools, libraries, and communication mechanisms that enable developers to build robot software in a structured and scalable manner.



This project was developed using ROS 1 Noetic Ninjemys, the final ROS 1 distribution, which is officially supported on Ubuntu 20.04 (Focal Fossa). This setup ensures compatibility with key

simulation tools such as Gazebo 11 and RViz, which are vital for modeling physical interactions, testing sensor data, and visualizing system behavior.

In this project, several ROS packages were utilized to enable autonomous navigation, localization, and environment mapping for the robotic system. The core packages included "**asv_wave_sim**", "**move_base**", "**amcl**", and "**slam_gmapping**". Together, they provided a simulation environment, path planning, localization, and SLAM (Simultaneous Localization and Mapping) capabilities.

- **asv_wave_sim:**
  A ROS package used to simulate an Autonomous Surface Vehicle (ASV) in a wave-affected water environment. It enables testing and validation of robot behaviour under realistic maritime conditions in Gazebo.

- **move_base:**
  Provides a high-level interface for autonomous navigation by combining global and local planners. It handles path planning, obstacle avoidance, and goal-reaching behaviour.

- **amcl:**
  Implements Adaptive Monte Carlo Localization for estimating the robot's pose in a known map. It uses sensor data (e.g., laser scans) and odometry to perform probabilistic localization.

- **slam_gmapping:**
  Allows the robot to build a 2D occupancy grid map of an unknown environment while tracking its own position. It leverages laser scan data and a particle filter algorithm to perform real-time SLAM (Simultaneous Localization and Mapping).

In addition to standard ROS packages, a custom package named boatcleaning was developed specifically for this project. The boat model was originally designed in SolidWorks, then exported as a URDF (Unified Robot Description Format) file and integrated into the ROS/Gazebo simulation environment. This model served as the primary representation of the autonomous cleaning boat in the project.

3.2. ROS Package Structure

3.2.1. Package Hierarchy Overview

After developing and integrating the core components, all ROS packages for this project were systematically arranged into a modular and well-organized folder structure. This approach improves the system's maintainability and scalability, while facilitating independent

development and testing of individual subsystems. The diagram below depicts the comprehensive organization of the ROS packages utilized in this project.

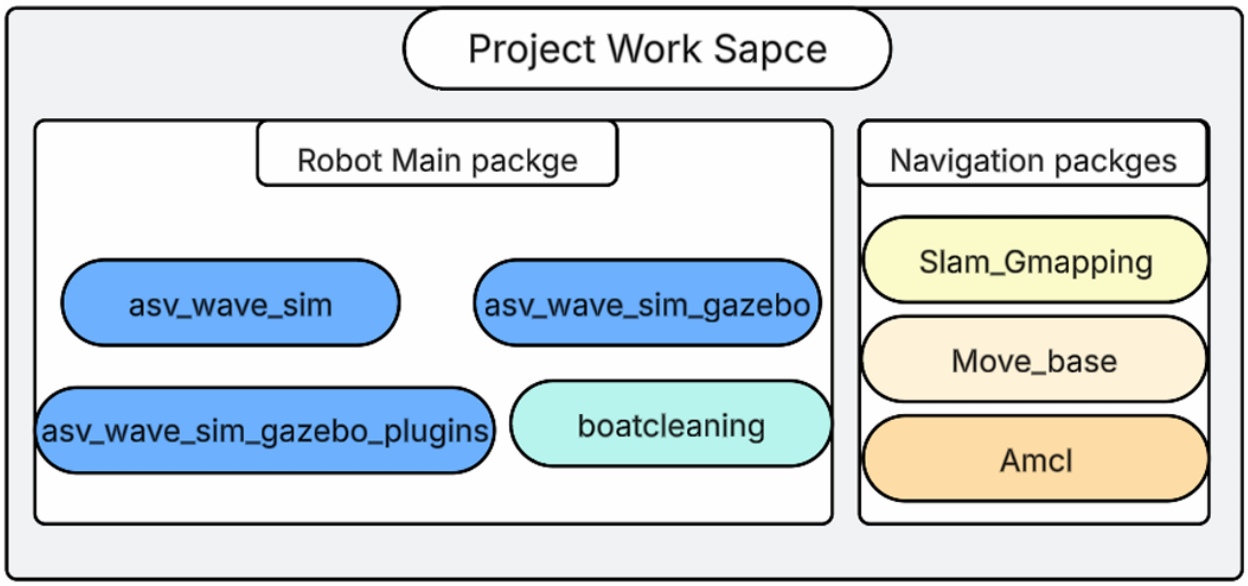


Figure 2 Package Hierarchy Overview

The "**asv_wave_sim**" package serves as the deployment environment for the final system, designed for real-world use on the autonomous boat. Within this framework, the "**asv_wave_sim_gazebo**" package is central, enabling the implementation and testing of various functionalities in a simulated environment using ROS tools such as RVIZ and Gazebo. Additionally, the "**asv_wave_sim_gazebo_plugins**" package incorporates pre-existing code to simulate water physics and ocean behavior. These packages streamline development and minimize risks of material damage or unforeseen errors during early testing phases.

The "**asv_wave_sim_gazebo**" package, which I developed, organizes the system's required functionalities in a clear and structured manner, with its internal folder structure, including directories.

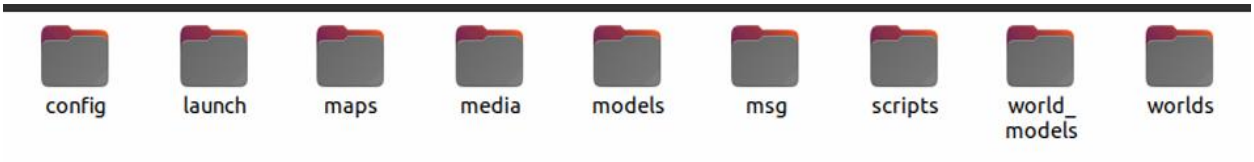


Figure 3Overview of the main directories in the ROS package structure

- **config**: This folder holds YAML files for system parameters like navigation and simulation settings. These files enable fine-tuning of the robot's behaviour for various marine environments.
- **launch**: This directory contains launch files that orchestrate the startup of multiple ROS nodes and configurations. It ensures smooth initialization of the system for simulation and deployment.
- **maps**: This directory stores PGM-format virtual maps essential for the navigation stack. These 2D occupancy grids support mapping and localization in static and dynamic marine settings.
- **media**: This folder includes visual assets like water textures to enhance Gazebo simulation realism. These assets improve the visual accuracy of the marine environment for testing.
- **models**: This folder contains 3D models and URDF files of objects like debris for Gazebo simulations. It supports testing of navigation and waste detection functionalities.
- **msg**: This directory defines custom ROS message types for efficient node communication. Messages like Detected Objects enable seamless data exchange for waste detection.
- **scripts**: This folder houses executable nodes for autonomous navigation and waste collection.
- **world_models & worlds**: These folders contain simulation maps and models defining ocean environments for Gazebo. They provide realistic scenarios for testing robot performance.

3.3 ROS1 Noetic Node Interaction and Data Flow

The fundamental building blocks of any robotic system are called **nodes**. A node is an independent executable that performs a specific task, such as reading sensor data or controlling actuators. These nodes are designed to work collaboratively by communicating with one another through topics, services, or actions.

3.3.1 Role of the ROS Master

The **ROS Master** acts as the central coordination service in any ROS1 Noetic system. Its main role is to manage the **registration of nodes**, **topics**, and **services**. When a node wants to publish or subscribe to a topic, it first communicates with the ROS Master to register its intent. The master then facilitates the **connection** between publisher and subscriber nodes by sharing their network addresses, enabling them to establish a direct peer-to-peer communication link.

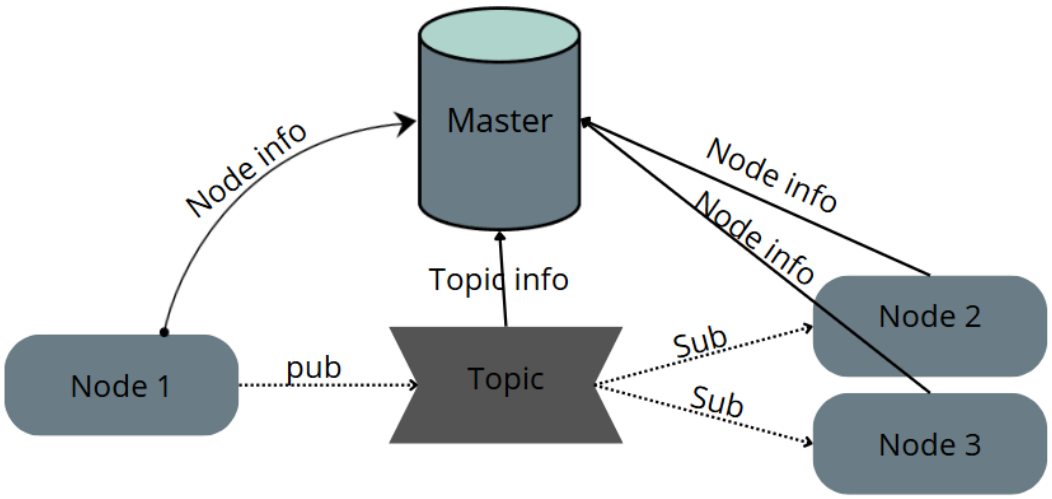


Figure 4 ros master architecture

- **Communication Between Nodes**

Communication in ROS1 Noetic is primarily handled using a **publish/subscribe model via topics**. A node that generates data (like a camera or IMU) publishes messages to a named topic, while other nodes that require this data subscribe to the same topic.

Figure 5 Communication Between Nodes

- **Importance of Handling Multiple Publishers on One Topic**

When multiple nodes publish to the same topic, such as a velocity command topic (/cmd_vel), the **subscriber will always receive only the latest message** sent. ROS does not provide built-in arbitration or prioritization between publishers.

3.4 ROS Node Architecture and Messaging

The autonomous marine robot's ROS system, which I developed, leverages a network of interconnected nodes to achieve modular functionality. This design organizes the system into

independent yet cooperative components, each handling specific tasks such as sensor processing or actuator control, seamlessly integrated within my custom boatcleaning package.

As shown in Figure {below}, the primary nodes developed for this system are described in the following sections.

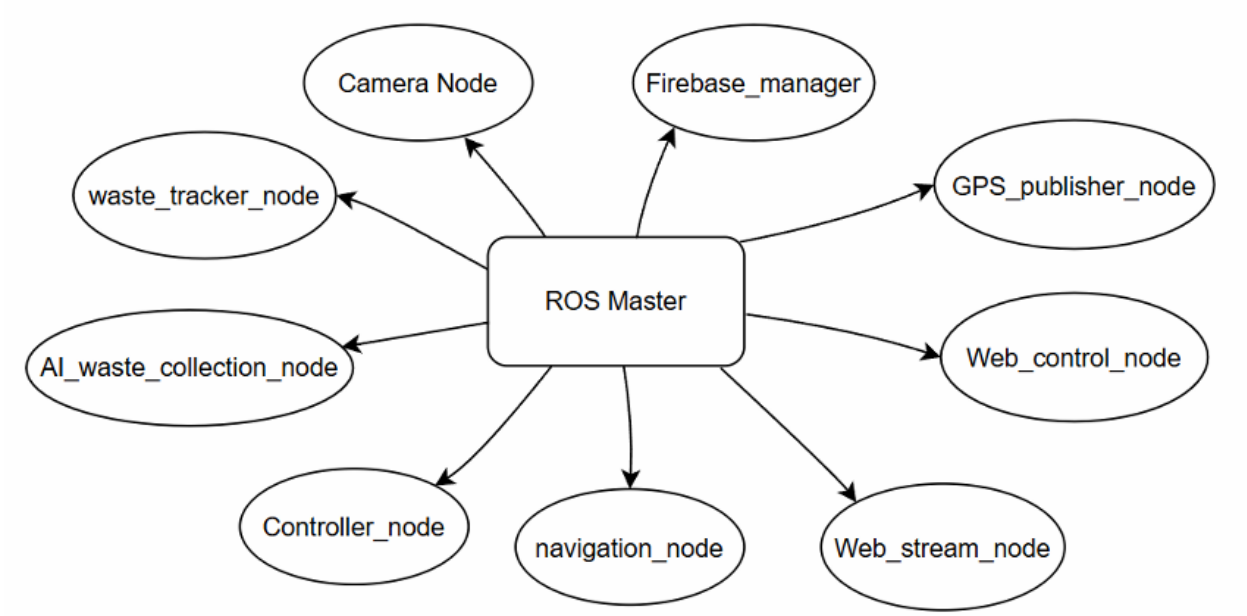


Figure 6 ROS Node Architecture

- **Camera_node**: Captures real-time video streams from the robot's onboard camera and publishes images to `/camera/image_raw`. It supplies image data to other nodes, such as the waste detection and tracking modules.
- **Firebase_manager:** This node sets up and manages the connection to the Firebase database. It handles uploading waypoints, downloading mission plans, and keeping the data synchronized between the robot and the cloud.
- **GPS_publisher_node:** This node listens to the robot's GPS data and sends those coordinates to Firebase for real-time tracking. Like the Firebase Manager, it communicates directly with the cloud, not through ROS topics.
- **Web_control_node:** Interfaces with the web-based control panel, enabling remote commands and real-time status updates between the robot and the user.
- **Web_stream_node:** Subscribes to `/camera/image_raw` and streams video and telemetry data to the web interface, allowing users to monitor the robot's progress and environment remotely.

- **navigation_node:** This node publishes updates about navigation to `/navigation_status` and listens for commands on `/navigation_control`. It also uses data from Firebase and the GPS publisher to follow the planned path.
- **waste_tracker_node:** This node tracks waste objects over time to help ensure a successful pickup. It listens to the camera stream via `/camera/image_raw` and publishes detection results and events on `/waste_info`, `/waste_detected`, and `/waste_detection`.
- **AI_waste_collection_node:** This node handles the actual collection of trash. When waste is detected, it processes the object's position and controls the robot to pick it up.
- **Controller_node:** Acts as the decision-making unit, managing state transitions (e.g., switching between navigation and waste collection modes). It subscribes to `/waste_detected`, `/collection_status`, and `/navigation_status`. It publishes to `/navigation_control`, `/collection_control`, and `/webcontroler`.

4. Boat Model

This section details the step-by-step process of creating and integrating the boat model into the ROS 1 Noetic simulation environment, including exporting the model from SolidWorks, importing it into the ROS workspace, configuring sensors (camera, LiDAR, and GPS) using URDF plugins and testing the navigation simulation.

4.1 Exporting from SolidWorks

To ensure compatibility with ROS and Gazebo, the SolidWorks model was exported using the "**SolidWorks to URDF Exporter"**, a plugin developed by the ROS community. This plugin generates a URDF (Unified Robot Description Format) file along with the associated mesh files (in .STL or .DAE format). Before exporting, all parts were properly named, assembled, and assigned coordinate frames in SolidWorks to ensure correct transformations and joint placement in ROS.

In our case, the exportation method provided mesh files specifically in the .STL format.

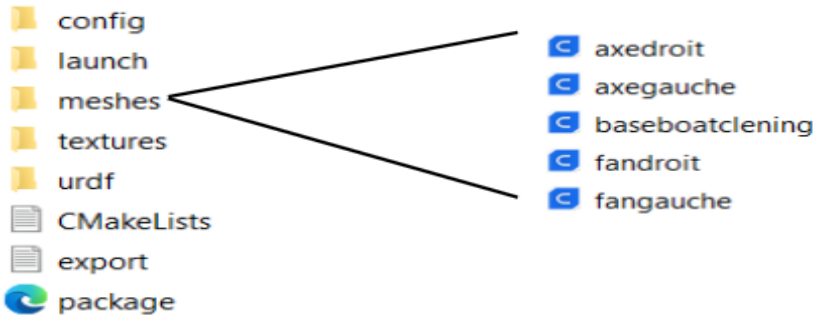


Figure 7 Meshes Folder Structure

4.2 Integration into ROS Workspace

A dedicated ROS workspace was created to organize the exported URDF files and streamline the development process. This workspace setup made it easier to modify the robot description and add custom plugin code as needed. The structure was designed to support further steps such as navigation and simulation, ensuring a smooth transition into the next phases of the project.

4.3 Sensor Integration and Plugins

To simulate autonomous capabilities, key sensors were integrated into the boat model using Gazebo plugins within the URDF description. These sensors provide essential data for perception, localization, and mapping in simulation. Each sensor was configured using <gazebo> tags inside the URDF file, specifying the plugin type, sensor parameters (such as update rate, resolution, and noise), and the sensor's pose relative to the boat's base link.

- **Camera Plugin**

The camera was integrated using the libgazebo_ros_camera.so plugin, which allows Gazebo to simulate an RGB camera and publish its data to ROS topics for use in perception tasks.

```
<plugin name="camera_controller" filename="libgazebo_ros_camera.so"> //Gazebo ROS camera plugin
  <alwaysOn>true</alwaysOn> //Keeps camera active continuously
  <updateRate>30</updateRate> //Update rate (30 Hz)
  <cameraName>camera</cameraName> //Name of the camera
  <frameName>camera</frameName> //Reference frame for the camera
  <imageTopicName>image_raw</imageTopicName> //ROS topic for raw image data
  <cameraInfoTopicName>camera_info</cameraInfoTopicName> //ROS topic for camera metadata
</plugin>
```

Figure 8 code Camera Plugin

- **Lidar Plugin**

The LIDAR sensor was integrated using the libgazebo_ros_laser.so plugin, which enables the simulation of a 2D laser scanner and publishes scan data to a ROS topic for tasks such as obstacle detection and mapping.

```
<plugin name="gazebo_ros_laser" filename="libgazebo_ros_laser.so"> //Gazebo ROS laser plugin
  <topicName>/scan</topicName> //ROS topic for laser scan data
  <frameName>lidar_center</frameName> //Reference frame for the laser sensor
</plugin>
```

Figure 9 Code Lidar Plugin

4.4 Navigation Simulation

Realistic boat navigation in simulation requires accurate modelling of hydrostatic forces and floatability. The “asv_wave_sim” package, designed for simulating ASVs in Gazebo, includes a hydrostatics plugin for buoyancy and floatation, wave generation, and water surface dynamics, enhancing navigation realism. It also offers prebuilt models like floating buoys and waste objects for scenarios such as obstacle avoidance and waste collection. Fully compatible with ROS Noetic and Ubuntu 20.04, this package ensures realistic hydrodynamic responses, supporting advanced navigation testing.

- **hydrodynamics plugin**

To achieve realistic buoyancy and water interaction, the `libHydrodynamicsPlugin.so` provided by `asv_wave_sim` was utilized. This plugin models the hydrodynamic forces acting on the boat, enabling accurate floating behavior and water resistance simulation. The configuration included damping, viscous drag, and pressure drag, three key components that contribute to the realistic motion of a vessel in water.

```
<plugin name="hydrodynamics" filename="libHydrodynamicsPlugin.so"> //Gazebo ROS hydrodynamics plugin
  <wave_model>ocean_waves</wave_model> //Wave model for water surface dynamics
  <damping_on>true</damping_on> //Enables damping forces
  <viscous_drag_on>true</viscous_drag_on> //Enables viscous drag forces
  <pressure_drag_on>true</pressure_drag_on> //Enables pressure drag forces
  <markers> //Visualization markers
    <update_rate>30</update_rate> //Marker update rate (30 Hz)
    <water_patch>false</water_patch> //Disables water surface patch visualization
    <waterline>false</waterline> //Disables waterline visualization
    <underwater_surface>false</underwater_surface> //Disables underwater surface visualization
  </markers>
</plugin>
```

Figure 10 Code Hydrodynamics plugin

With the integration of the libHydrodynamicsPlugin.so, the boat model successfully navigated in the Gazebo 11 simulation environment. The hydrodynamics plugin provided realistic water interaction, ensuring stable floating behavior and accurate responses to hydrodynamic forces such as damping. The image below illustrates the boat's successful navigation

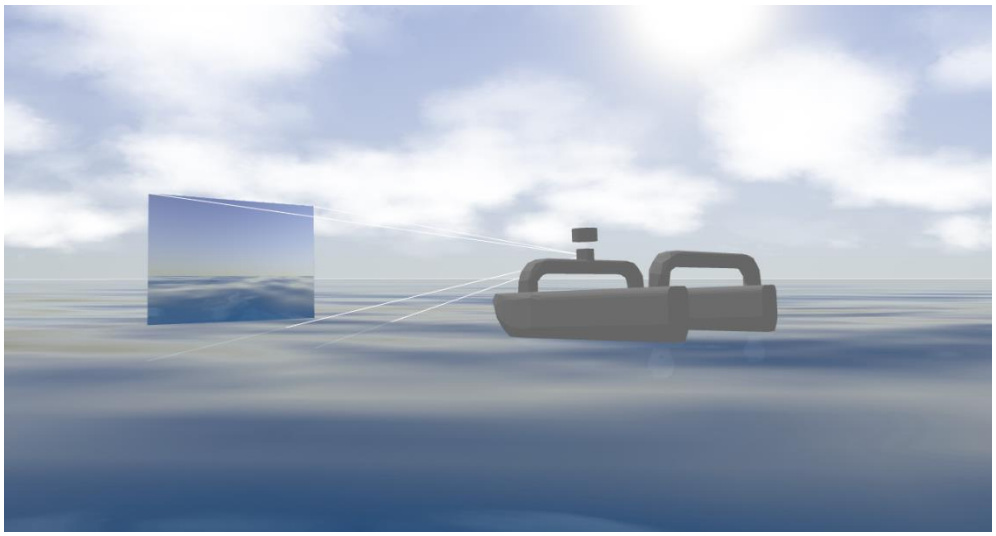


Figure 11Boat Navigation in Gazebo

5. Autonomous Navigation Simulation

The autonomous navigation system was developed within the `asv_wave_sim_gazebo` simulation package. This environment was the core platform for implementing and testing the boat's navigation capabilities. To achieve full autonomy, two additional ROS packages were integrated during this phase:

- Slam_Gmapping: Enabled simultaneous localization and mapping (SLAM), allowing the boat to build a map of its environment and localize itself in real-time.
- Move_Base: Provided the path planning and navigation stack, enabling the ASV to autonomously plan routes and avoid obstacles.
- AMCL: Implemented Adaptive Monte Carlo Localization, allowing the boat to accurately determine its position within a previously generated map using sensor data and probabilistic methods.

5.1 Mapping Package Configuration

The configuration of slam_gmapping is an essential step in enabling robust simultaneous localization and mapping (SLAM) for ROS-based robotics projects. This package utilizes laser scan data and odometry information to construct incremental 2D occupancy grid maps of the environment while estimating the robot's pose in real time. Much of the configuration for slam_gmapping is managed through YAML files, typically placed in the project's config directory. These YAML files allow users to set and adjust important parameters such as map resolution, update rates, particle filter settings, and sensor topic names in a clear and organized manner.

```yaml
global_costmap:
  global_frame: map           # Matches your gmapping map frame
  robot_base_frame: baseboatclening  # Matches your URDF base link
  update_frequency: 1.0       # Update at 1 Hz
  static_map: true            # Use the pre-built map from gmapping

local_costmap:
  global_frame: odom          # Matches your odom plugin's frameName
  robot_base_frame: baseboatclening  # Matches your URDF base link

  update_frequency: 5.0       # Update at 5 Hz
  publish_frequency: 4.0      # Publish at 4 Hz

  static_map: false
  rolling_window: true        # Keep the costmap centered on the boat
  width: 6.0                 # 6 m x 6 m local window
  height: 6.0
  resolution: 0.05           # 5 cm grid resolution
```

Figure 12 Costmap YAML Configuration

5.2 Map Generation and Visualization

The process of map generation and visualization is fundamental in autonomous robotics, as it enables the robot to understand and navigate its environment effectively. In this project, the configured `slam_gmapping` package was used to collect laser scan and odometry data, allowing the robot to construct an occupancy grid map of its surroundings in real time. Map generation not only provides essential spatial information for navigation but also plays a crucial role in planning and obstacle avoidance.

To ensure accurate localization within the map generated, the AMCL package was integrated. AMCL uses particle filtering to continuously estimate the robot's position on the map using sensor data, providing robust and reliable localization for autonomous navigation.

The static map represents a snapshot of the environment created after the mapping process is complete. This type of map is especially useful in environments where the arrangement of obstacles and features remains unchanged over time, such as offices, warehouses, or laboratories. By relying on a static map, the robot can efficiently plan routes and avoid obstacles without the need for ongoing map updates, which reduces computational demands during navigation. Furthermore, static maps are valuable for defining the boundaries of the robot's operational area. This allows users to specify precise limits for navigation, ensuring the robot focuses only on designated zones, an important consideration for tasks like surface cleaning, where it is essential to restrict the robot's activity to targeted areas.

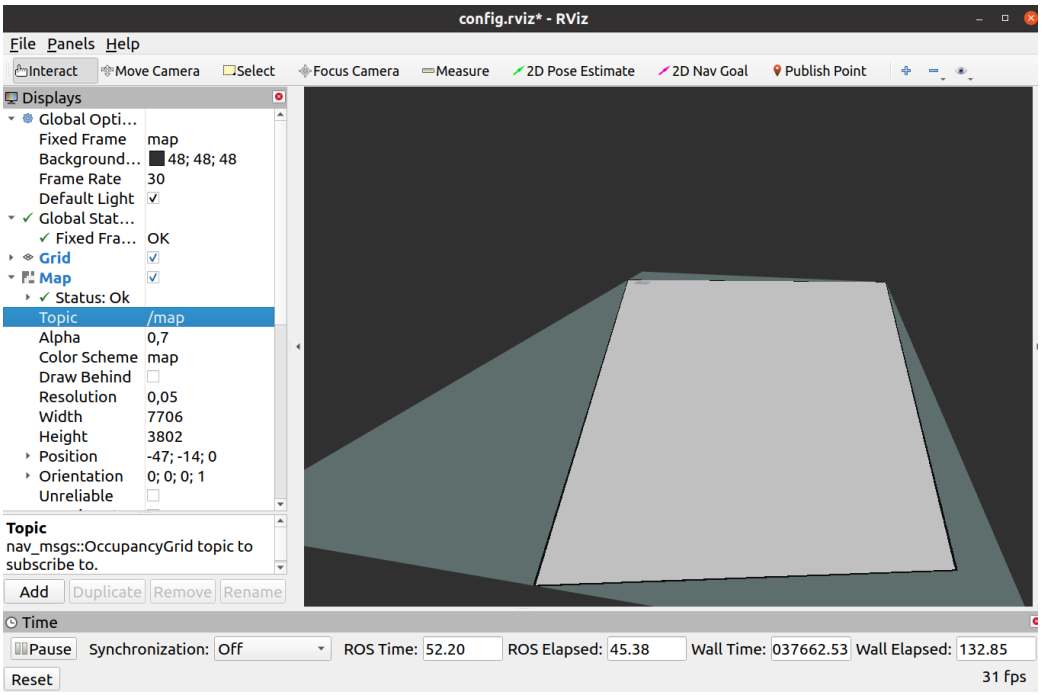


Figure 13 RViz Static Map Visualization

In contrast, the dynamic map is continuously updated as the robot moves and as the environment evolves. This approach is crucial in scenarios where the environment is subject to change, such as public spaces, warehouses with moving objects, or outdoor settings where obstacles may appear or disappear. Dynamic mapping allows the robot to adapt in real time, ensuring that its understanding of the environment remains accurate and up to date. This capability is especially important for applications requiring high levels of autonomy and safety, as it enables the robot to respond to unexpected changes and maintain reliable navigation even in unpredictable conditions.

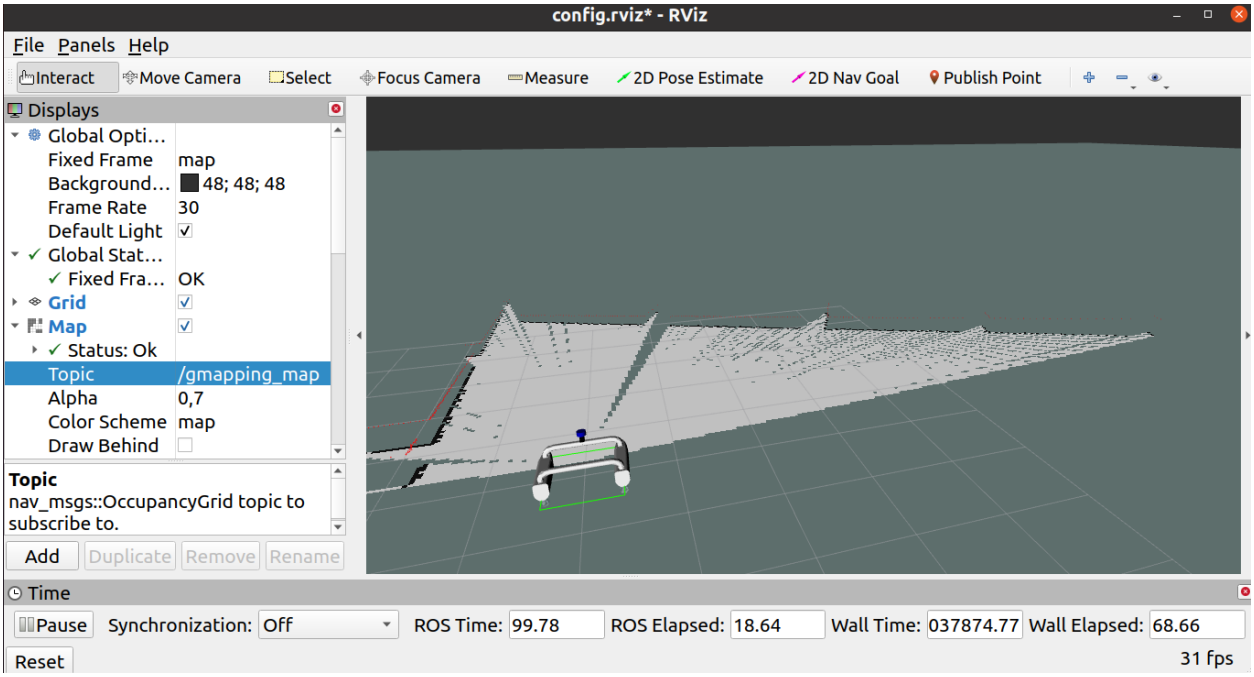


Figure 14 RViz Dynamic Map Visualization

5.3 Obstacle Avoidance

Obstacle avoidance is a crucial capability for any autonomous vehicle, ensuring not only the successful completion of navigation tasks but also the safety of the platform and its environment. In this simulation, obstacle avoidance was achieved through the integration of the move_base package, real-time sensor feedback, costmaps, and interactive goal setting using RViz.

The move_base node is the core of the ROS navigation stack for autonomous path planning and obstacle avoidance. It operates by integrating two planners and two types of costmaps:

- Global Planner & Global Costmap: The global planner uses the static (global) costmap, built from the known map, to compute an initial path from the robot's current position to the goal while avoiding fixed obstacles.
- Local Planner & Local Costmap: The local planner relies on the local costmap, which is continuously updated with real-time sensor data. This allows the robot to detect and react to dynamic obstacles, such as moving objects or people, by adjusting its trajectory in real time.

In situations where no valid path is available, such as when the robot is trapped or surrounded by obstacles, the move_base node can trigger recovery behaviors. These include actions like rotating in place to search for an escape route or clearing the local costmap to remove outdated obstacle data. Such strategies help the robot resolve navigation deadlocks and continue toward its goal, increasing the overall robustness and reliability of autonomous navigation.

6. AI-Based Object Detection and Waste Collection

A key part of this project is the implementation of an AI model for real-time object detection and waste collection in marine environments. This section presents the entire workflow, from dataset preparation to model development and integration with the autonomous cleaning system.

6.1 Dataset Collection and Preparation

The dataset for this project was entirely sourced and processed using Kaggle. Images of floating marine waste, including plastics, metal debris, and glass, were obtained directly from Kaggle datasets. The raw data underwent cleaning, standardization, and re-annotation to ensure compatibility with YOLOv11. This process included removing unusable images, correcting annotation formats, and maintaining consistent class labels. The resulting datasets were prepared for training, testing, and validation to develop robust object detection models.

6.2 Model Development

I built the core of our object detection system around the YOLOv11 model because it's excellent for spotting things in real-time. I worked with two versions to suit different needs: YOLOv11m and YOLOv11n. For the YOLOv11m, I trained it on a dataset with 15,018 images split into plastic, metal, and glass categories. I ran it for 100 epochs with 32 batches using two NVIDIA T4 GPUs on

Kaggle. The results turned out great—around 90% precision, 80% recall, and a mean Average Precision (mAP50) of 0.881. I developed this version for use with Gazebo and RViz tools, where it can help simulate and visualize the marine environment for testing and analysis.

For the YOLOv11n, I designed it specifically for the Raspberry Pi. I trained it on a larger single-class dataset of 21,907 images, all labeled as "waste," again with 100 epochs and 32 batches. It reached about 85% precision, and I focused on making it fast and responsive—perfect for real-time use on the boat. I fine-tuned both models with the datasets I prepped to tackle the challenging marine environment, making them essential for our waste collection system when integrated with Gazebo and RViz.

6.3 Autonomous Waste Collection

The waste collection process in this project relies on a smooth combination of detection, tracking, and autonomous navigation. The key element is the `Ai_collection_node`, which takes the information from the object detection system and translates it into movement commands for the robot.

When the camera detects a piece of waste and determines its position, the `Ai_collection_node` decides how the robot should move to collect it. The main goal is to manoeuvre the boat so that the detected waste is right in front of the collection mechanism.

To do this, the robot's camera view is divided into five sections. If the waste appears in one of these sections, the `Ai_collection_node` rotates the boat to centre the waste in the middle of the camera's view. Once the waste is lined up correctly, the robot moves forward to guide the waste into its collection system.

This approach helps the robot collect waste efficiently and accurately, reducing the chances of missing objects and making the overall cleaning operation much more effective.

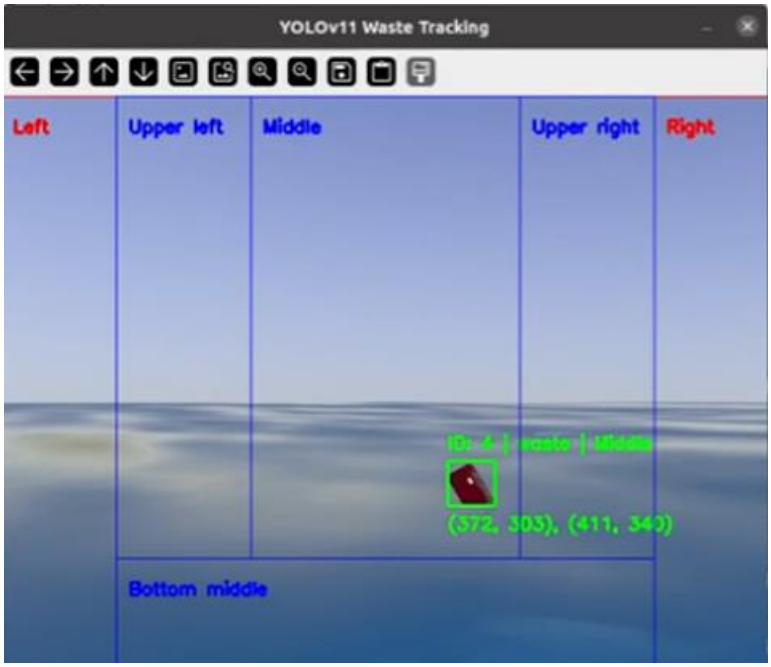


Figure 15 AI Waste Tracking Interface

7. Autonomous Cleaning Program

The autonomous cleaning program is the core of the robot's operation, seamlessly integrating mapping, navigation, detection, and waste collection into a single workflow. Each stage of the process is detailed below, with visual illustrations to support each step.

7.1 Defining the Cleaning Zone

To begin, I developed an HTML page displaying an interactive world map (by default, centered on Tunisia). On this interface, I selected the specific area that the robot should clean. The selected region then serves as the static boundary for the robot's autonomous navigation.

Figure 16 Selection of Cleaning Area on Interactive Map

The application generates the latitude and longitude coordinates of the polygon marking this area.

7.2 Planning the Navigation Path

Next, I imported these coordinates into a desktop application that converts the (lat, long) points to local (x, y) coordinates. In this app, I chose the robot's starting point for the cleaning mission. The application supports both manual and automatic waypoint generation, ensuring the robot systematically covers the entire cleaning zone.

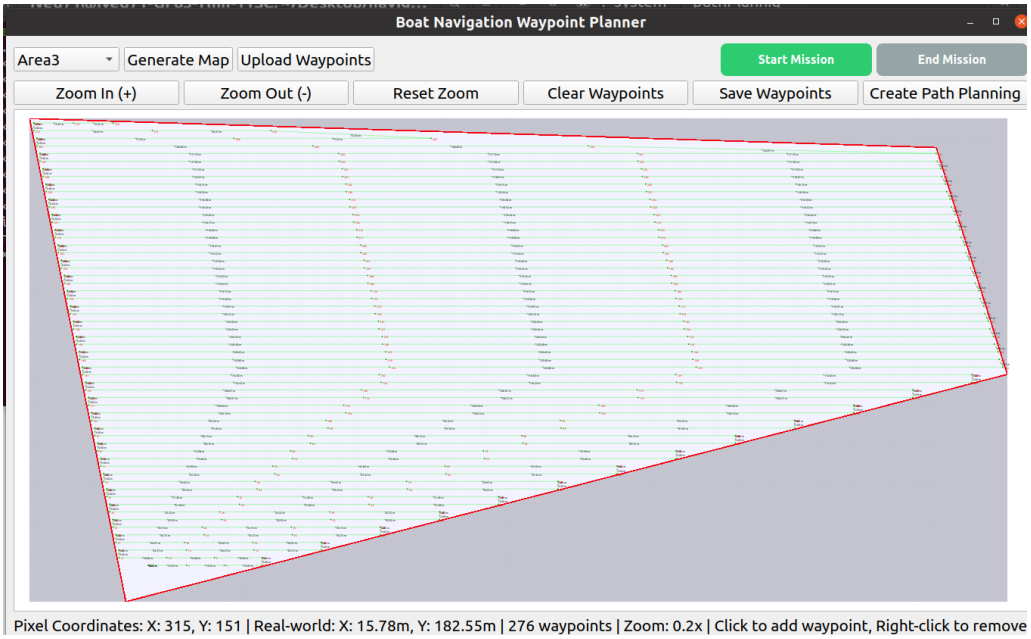


Figure 17 Waypoint Generation and Path Planning

7.3 Uploading and Synchronizing the Map

Once the navigation map and waypoints are prepared, I uploaded them to Firebase. This ensures synchronization with the main project workspace. The autonomous navigation node then retrieves these waypoints from Firebase and guides the boat through each one in sequence.

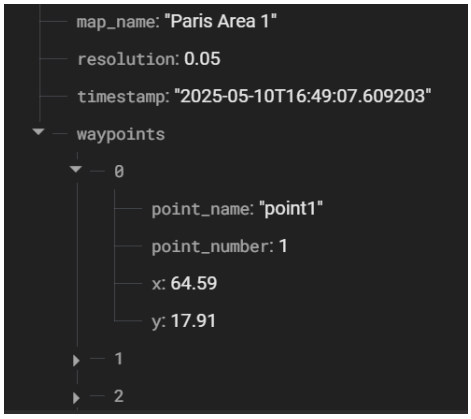


Figure 18 Firebase Waypoint Data Structure

The waypoint and map data are structured and stored in Firebase for synchronization with the robot's navigation system.

7.4 Autonomous Navigation and Waste Detection

As the boat follows its planned route, the onboard navigation node continuously manages movement commands. If the waste detection camera node spots a floating object, the controller node pauses navigation to prioritize collection.

7.5 Waste Tracking and Collection

When waste is detected, control shifts to the waste collection node. This node tracks the detected object, plotting a path to align the waste with the centre of the boat's collection area for reliable retrieval. The robot actively adjusts its heading during this phase to ensure successful collection.

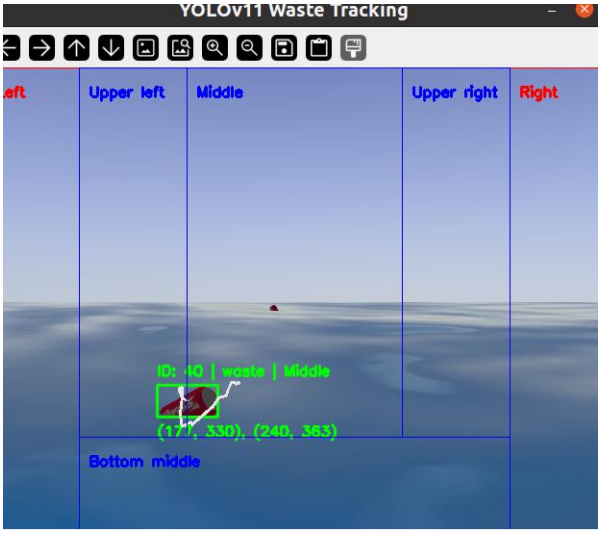


Figure 19 Waste Detection and Tracking Interface

7.6 Resuming Navigation

After waste is collected, the controller node re-enables the navigation node. The boat proceeds to the next unvisited waypoint, not returning to the starting point, and continues this process until it reaches the final waypoint and completes the cleaning mission.