HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT 1 FINAL REPORT

Students:
- Kieu Son Tung (20235571)
- Pham Chi Bang (20235477)
- Trinh Hoang Anh (20235476)

Class: DSAI 02 - K68

Semester: 2025.1

February 4, 2026

Contents

1 Introduction

In the initial phase of Project 1, I focused on establishing the necessary software infrastructure for Robotics research. This report details the successful deployment of the Robot Operating System 2 (Version: Jazzy Jalisco) and the Gazebo Harmonic physics simulator on the Windows Subsystem for Linux (WSL) platform. Additionally, this report records experimental results regarding the control of the TurtleBot3 robot and the visualization of LaserScan sensor data within the simulated environment.

2 System Environment

The system was constructed with the following specific configuration:

- **Host Platform:** Windows 11 with WSL 2.
- **Guest Operating System:** Ubuntu 24.04 LTS (Noble Numbat).
- **Middleware:** ROS 2 Jazzy Jalisco.
- **Simulator:** Gazebo Harmonic.
- **Robot Platform:** TurtleBot3 (Model: Burger).

3 Detailed Installation and Configuration Process

3.1 Windows Subsystem for Linux (WSL) Installation

To leverage hardware performance on Windows 11, I installed WSL 2 and set up an Ubuntu 24.04 virtual machine via PowerShell (Administrator privileges).

PowerShell: Initial Setup

```
# Install WSL and Ubuntu 24.04 default
wsl --install -d Ubuntu-24.04

# Update WSL kernel (optional but recommended)
wsl --update
```

After rebooting and accessing Ubuntu, I created a new user, set a password, and granted administrative privileges (sudo) to prepare for package installation.

Ubuntu: Granting Sudo Access

```
# Grant sudo privileges to current user
sudo usermod -aG sudo tung
```

3.2 ROS 2 Jazzy Jalisco Installation

The installation process followed the official documentation, including adding the GPG Key and repository.

Ubuntu: ROS 2 Setup

```
# 1. Setup Locale
sudo apt update && sudo apt install locales
sudo locale-gen en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

# 2. Add GPG key
sudo apt install software-properties-common curl
sudo add-apt-repository universe
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
     -o /usr/share/keyrings/ros-archive-keyring.gpg

# 3. Add Repository
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/
    ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-
    release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d
    /ros2.list > /dev/null

# 4. Install ROS 2 Desktop
sudo apt update
sudo apt install ros-jazzy-desktop ros-dev-tools
```

3.3 Environment Configuration (Sourcing)

To execute ROS 2 commands from any Terminal, I added the source command to the shell startup configuration file (`.bashrc`).

Ubuntu: Configure .bashrc

```
# Add sourcing command to .bashrc
echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc

# Apply changes immediately
source ~/.bashrc
```

3.4 Gazebo Harmonic and Simulation Packages Installation

Installing Gazebo Harmonic version and TurtleBot3 support packages.

Ubuntu: Install Gazebo & Packages

```
# Install Gazebo Harmonic integration
sudo apt install ros-jazzy-ros-gz

# Install TurtleBot3 packages
sudo apt install ros-jazzy-turtlebot3
sudo apt install ros-jazzy-turtlebot3-simulations
sudo apt install ros-jazzy-turtlebot3-gazebo
sudo apt install ros-jazzy-teleop-twist-keyboard

# Set robot model in .bashrc
echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
source ~/.bashrc
```

4 Experiments and Results

The testing process was conducted using a Multi-process model across three separate Terminal windows.

4.1 Terminal 1: Launching the Simulation Environment

This window is responsible for running the Gazebo backend and loading the simulation world.

Terminal 1: Launch Gazebo

```
# Launch Gazebo with TurtleBot3 world
source /opt/ros/jazzy/setup.bash
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Result: The Gazebo environment displayed successfully. The TurtleBot3 Burger robot appeared at the origin coordinates along with obstacle objects.

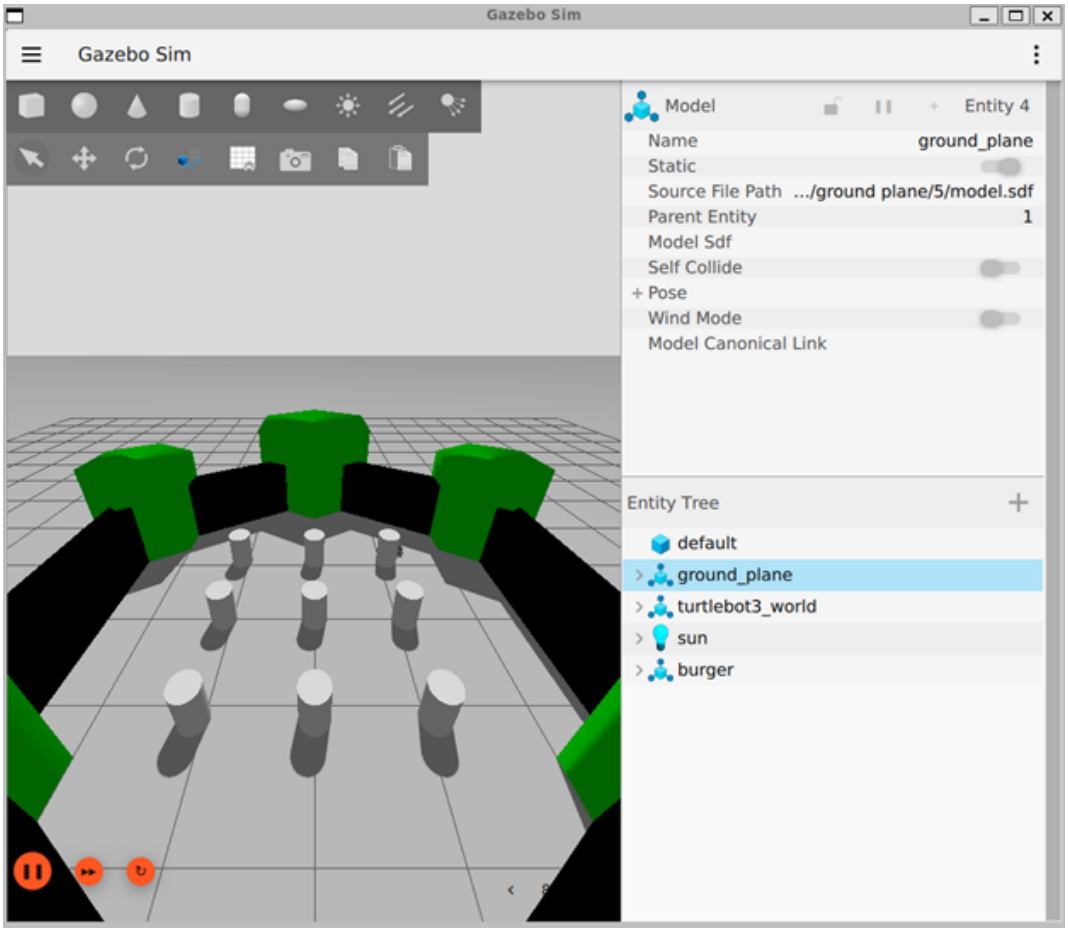

Figure 1: Gazebo Harmonic simulation interface with TurtleBot3

4.2 Terminal 2: Robot Teleoperation

This window runs the control node, sending velocity signals to the `/cmd_vel` topic.

Terminal 2: Teleop Keyboard

```
# Run teleop node
source /opt/ros/jazzy/setup.bash
export TURTLEBOT3_MODEL=burger
ros2 run turtlebot3_teleop teleop_keyboard
```

Control Operation: Using keys **w, a, s, d, x** to move the robot. The robot responded well to control commands, moving smoothly within the simulated environment.

4.3 Terminal 3: Data Visualization

This window runs RViz2 to display LiDAR sensor data.

Terminal 3: RViz2

```
# Run Visualization tool
ros2 run rviz2 rviz2
```

RViz2 Configuration Performed:

- **Fixed Frame:** Switched to `odom`.
- **Add Topic:** Added `/scan` to display LaserScan (environmental scan data).
- **Add RobotModel:** Added the robot model from the `/robot_description` topic.

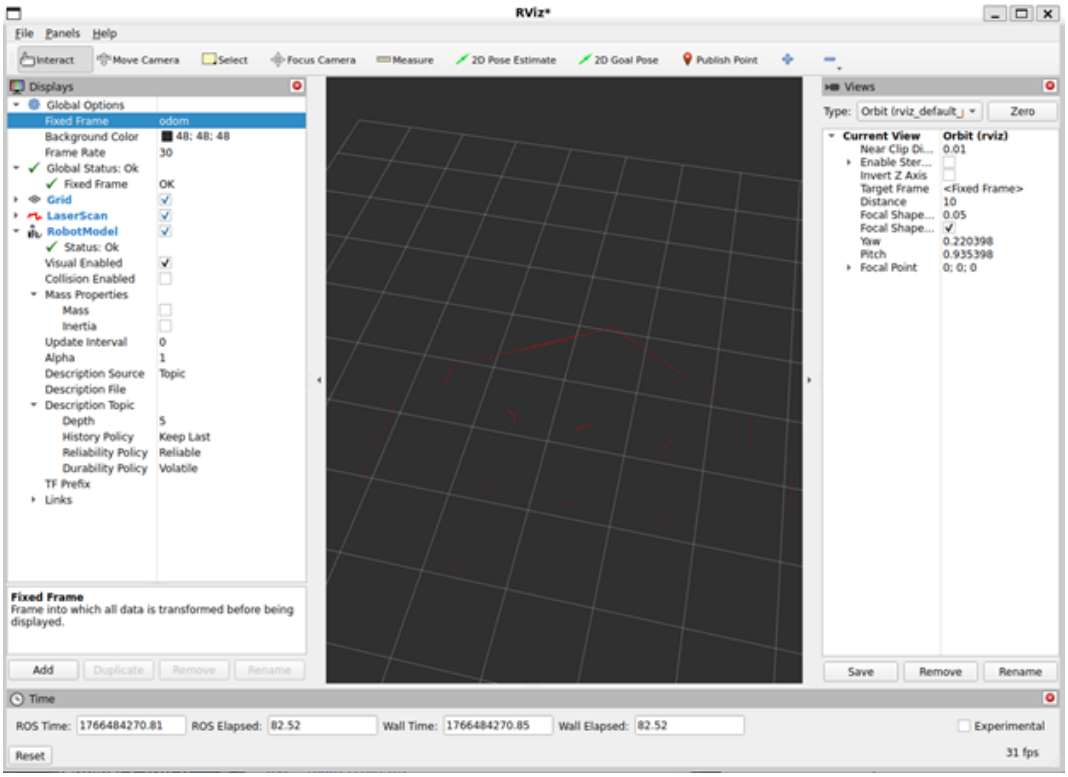

Figure 2: RViz2 interface displaying Laser scan data (red points)

5 Research on Navigation and Pathfinding Algorithms

5.1 Problem Statement

Following the successful deployment of the simulation environment and manual control capabilities, the next phase of the project focuses on **Autonomous Navigation**. The primary objective is to enable the TurtleBot3 to autonomously plan a path and navigate from a starting point to a target destination within a known map, avoiding static and dynamic obstacles without human intervention.

5.2 Navigation 2 (Nav2) Stack Architecture

In ROS 2 Jazzy, the standard framework for autonomous mobile robots is the **Navigation 2 (Nav2)** stack. Nav2 utilizes Behavior Trees to manage complex navigation tasks and divides the pathfinding process into two distinct layers:

- **Global Planner:** This component calculates the optimal path from the robot's current pose to the goal pose based on a static map. It considers the entire environment connectivity.

- **Local Planner (Controller):** This component computes the immediate velocity commands (v, ω) to follow the global path while avoiding unforeseen obstacles. It operates at a high frequency to ensure smooth motion.

5.3 Pathfinding Algorithms Analysis

Based on the Nav2 technical documentation, I have identified the core algorithms that will be applied to the TurtleBot3 platform:

5.3.1 Global Planners (Path Finding)

These algorithms are managed by the `nav2_planner` server:

- **NavFn Planner:** This is the default planner for differential drive robots. It implements classical graph-search algorithms such as **Dijkstra** or **A* (A-Star)** on a costmap grid to find the shortest path. This is the most suitable choice for our current indoor simulation setup.
- **Smac Planner:** An advanced planner that supports Hybrid-A*, designed for non-holonomic robots but also applicable to circular robots requiring high-precision planning.

5.3.2 Local Planners (Path Tracking)

These algorithms are managed by the `nav2_controller` server:

- **DWB (Dynamic Window Approach):** An evolution of the classic DWA algorithm. It samples a set of achievable velocities, simulates the robot's trajectory for a short duration, and scores them based on criteria (distance to goal, distance to obstacles, alignment with global path). The trajectory with the highest score is executed.

5.4 Implementation Plan and Usage

To implement autonomous navigation, the following steps are required within the ROS 2 environment:

Step 1: Package Installation The Navigation 2 stack and SLAM (Simultaneous Localization and Mapping) tools must be installed.

Install Nav2 and SLAM Toolbox

```
# Install Navigation 2 Stack
sudo apt install ros-jazzy-navigation2
sudo apt install ros-jazzy-nav2-bringup

# Install SLAM Toolbox for mapping
sudo apt install ros-jazzy-slam-toolbox
```

Step 2: Map Generation (SLAM) Before navigation can occur, the robot must explore the environment to create a static map using SLAM.

Launch SLAM (Async Mode)

```
# Launch SLAM Toolbox
ros2 launch slam_toolbox online_async_launch.py
```

Procedure: While SLAM is running, use the teleop keyboard node to drive the robot around the environment until the map in RViz is complete.

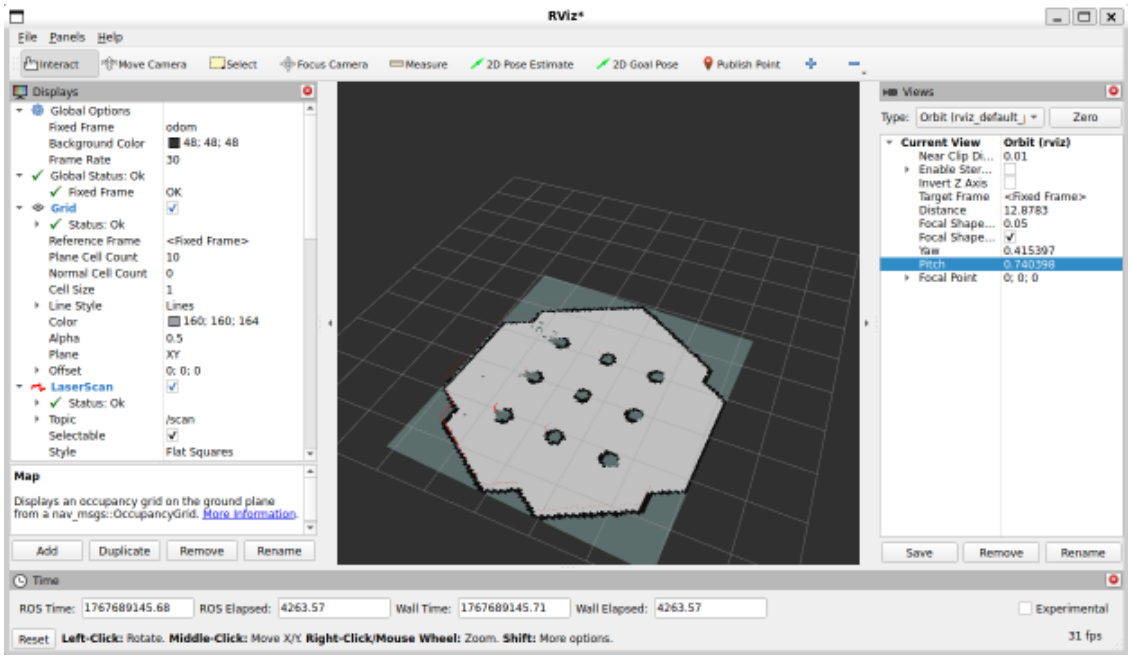

Figure 3: Completed SLAM Map of the TurtleBot3 World environment

Step 3: Saving the Map Once the map is fully constructed (as shown in Figure 3), it must be saved to the disk to serve as input for the Navigation stack.

Save Map to Disk

```
# Create a directory for maps (optional)
mkdir -p ~/robot_maps

# Save the map (this creates .pgm and .yaml files)
ros2 run nav2_map_server map_saver_cli -f ~/robot_maps/my_map
```

Step 4: Launching Navigation With the map saved, the navigation system is initialized.

Execute Navigation

```
# Launch Navigation with the saved map
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True
    map:=$HOME/robot_maps/my_map.yaml
```

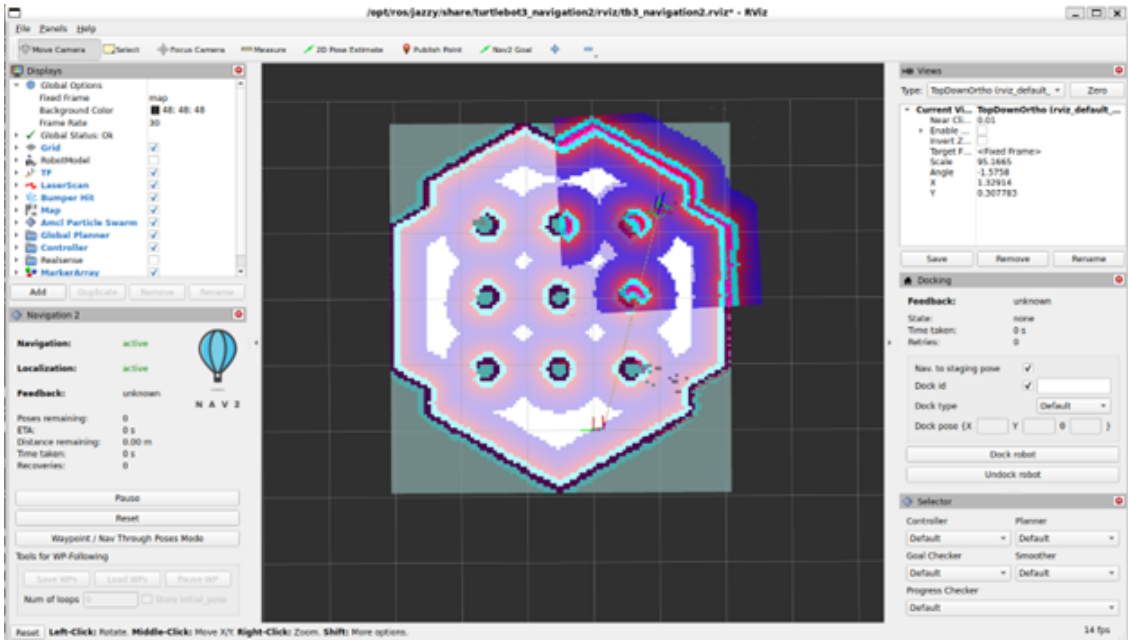

Figure 4: Navigation stack initialized successfully with the static map loaded in RViz2

Step 5: Operation in RViz The navigation process is fully controlled via the RViz interface. To initiate autonomous movement, the following procedures must be performed:

1. **Localization (2D Pose Estimate):** Upon initialization, the robot's estimated position might not match the simulation. Select the **2D Pose Estimate** tool from the top toolbar, then click and drag on the map to manually align the robot's pose with its actual position in Gazebo.

2. **Commanding the Robot (Nav2 Goal):** Select the **Nav2 Goal** tool. Click and drag at the desired destination on the map to set the target position and final orientation.

Result: As shown in Figure 5, the Global Planner calculates the optimal path (visualized as a thin line), and the Local Planner drives the robot along this trajectory while dynamically avoiding obstacles.

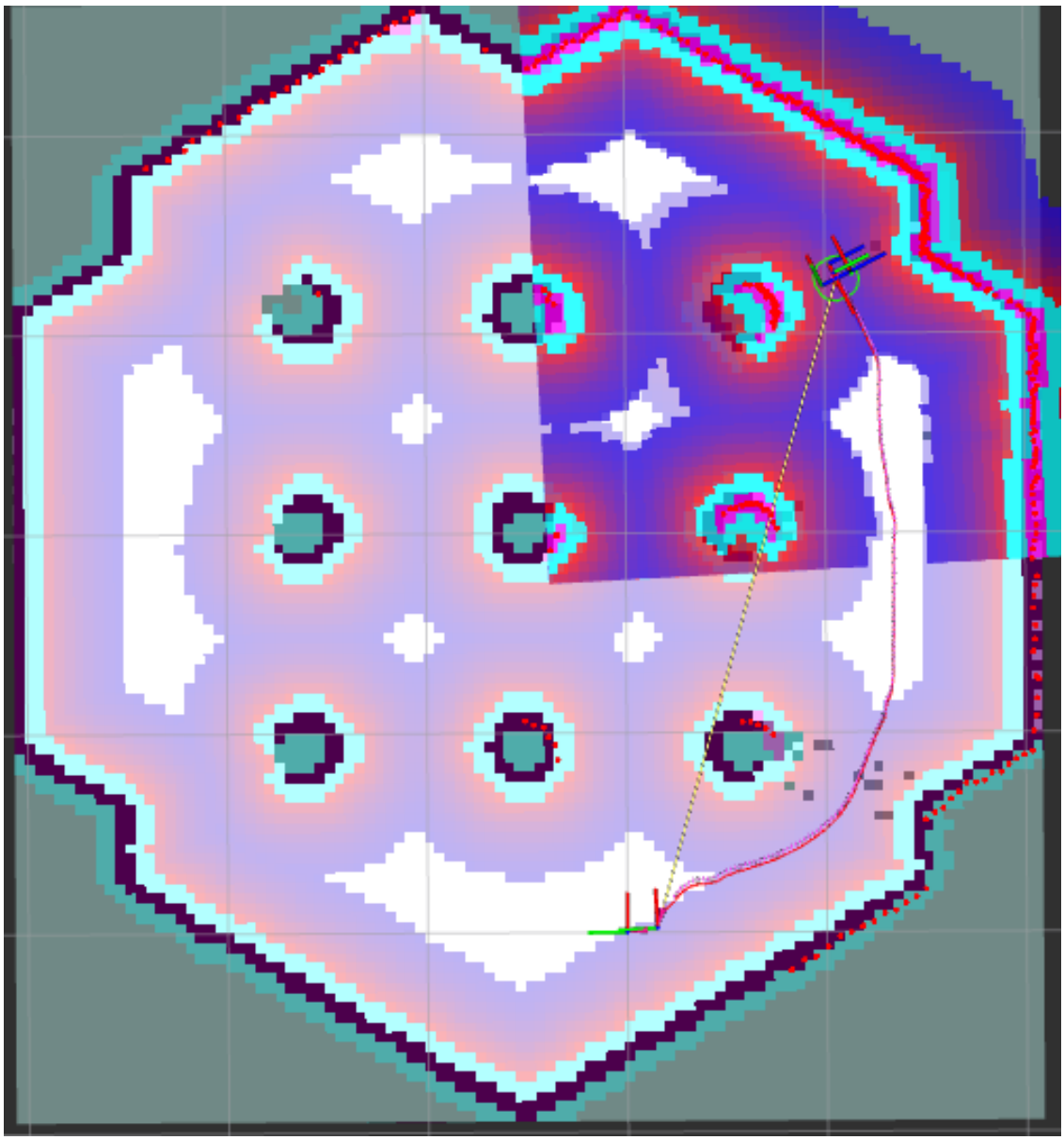

Figure 5: TurtleBot3 autonomously following the global path (green line) to the goal

6 Programmatic Navigation Control

6.1 Objective

While RViz is excellent for testing, real-world robotics applications require automated control via software. In this section, I utilize the `nav2_simple_commander` API to write a Python script that enables the robot to autonomously patrol through a sequence of pre-defined waypoints.

6.2 The Nav2 Simple Commander API

The `Nav2 Simple Commander` is a Python library provided by the Navigation 2 stack. It offers a high-level interface to interact with the Nav2 action servers, allowing developers to:

- Set initial poses programmatically.
- Send single goals (GoToPose) or multiple goals (FollowWaypoints).

- Monitor navigation feedback and cancel tasks if necessary.

6.3 Implementation: Waypoint Following Script

I developed a script named `patrol_robot.py` to simulate a security patrol task. The robot is commanded to visit a sequence of coordinate points on the map created in the previous section.

patrol_robot.py (Simplified)

```
from nav2_simple_commander.robot_navigator import BasicNavigator
import rclpy
from geometry_msgs.msg import PoseStamped

def main():
    rclpy.init()
    nav = BasicNavigator()
    nav.waitUntilNav2Active()

    # Define waypoints list
    waypoints = []

    # Waypoint 1
    wp1 = PoseStamped()
    wp1.header.frame_id = 'map'
    wp1.pose.position.x = 1.5
    wp1.pose.position.y = 0.5
    wp1.pose.orientation.w = 1.0
    waypoints.append(wp1)

    # Send the path to the robot
    nav.followWaypoints(waypoints)

    while not nav.isTaskComplete():
        pass # Wait for task completion

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

6.4 Execution and Results

To execute the autonomous patrol, the following command was used while the Nav2 stack was active:

Terminal: Run Python Script

```
python3 patrol_robot.py
```

Result: The robot successfully received the waypoint list. The Global Planner generated a continuous path connecting all points, and the robot navigated sequentially to each location without

stopping.

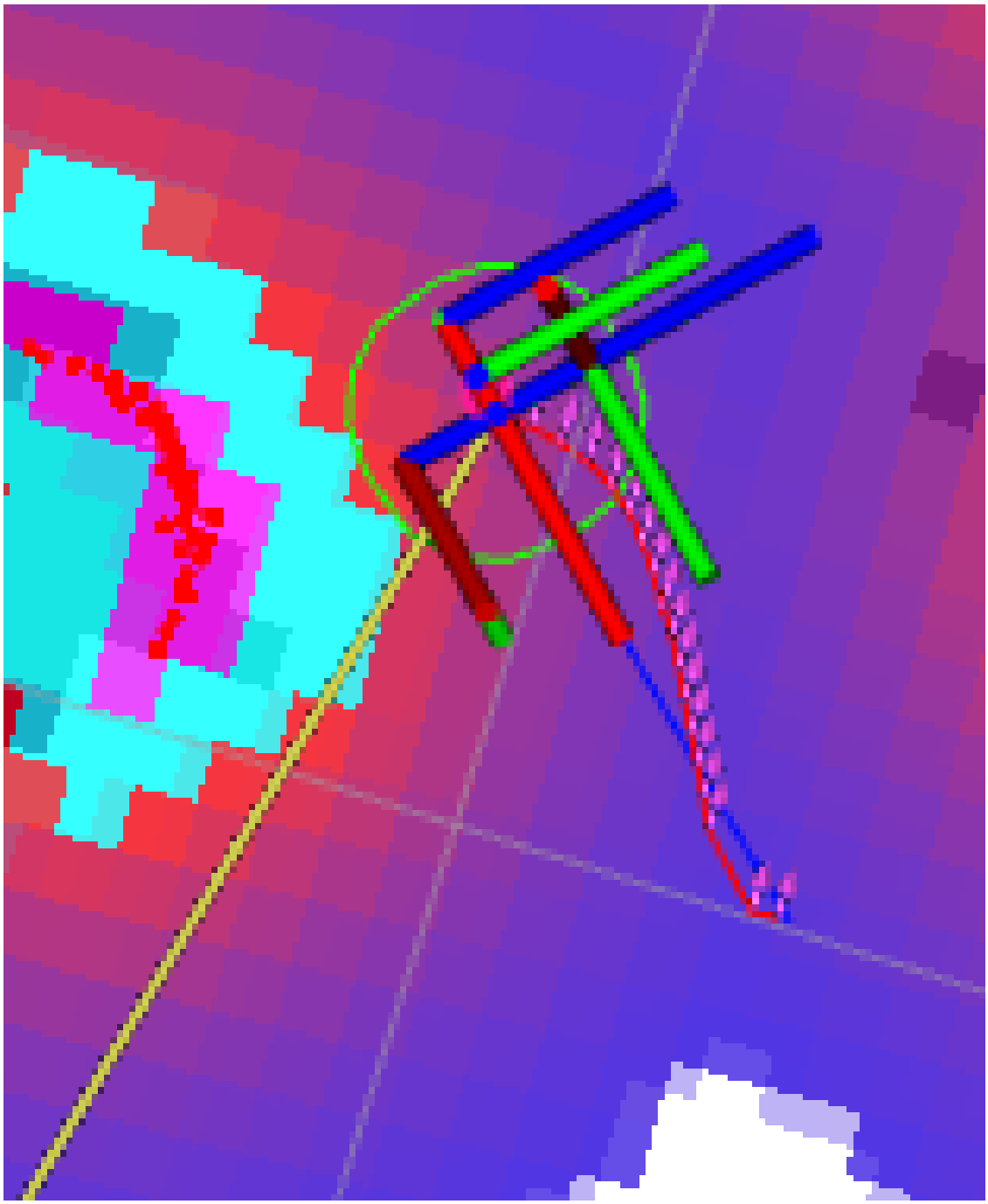

Figure 6: TurtleBot3 executing a multi-point patrol mission via Python script

7 Resource Management with Reinforcement Learning

Beyond navigation on a stable host, I evaluated how ROS 2 SLAM behaves when the operating system is overloaded and applied Reinforcement Learning (RL) to mitigate resource contention.

7.1 Problem and Symptoms

ROS 2 nodes share CPU fairly under Linux CFS, so critical nodes (e.g., `slam_toolbox`) compete equally with background tasks. The scheduler slices CPU time without knowing task importance,

so bursts of background work push SLAM off-core, delaying scans and creating map drift (ghost walls). Injecting CPU hogs (60–100% load) on WSL2 caused delayed scan processing and map drift, effectively breaking navigation.

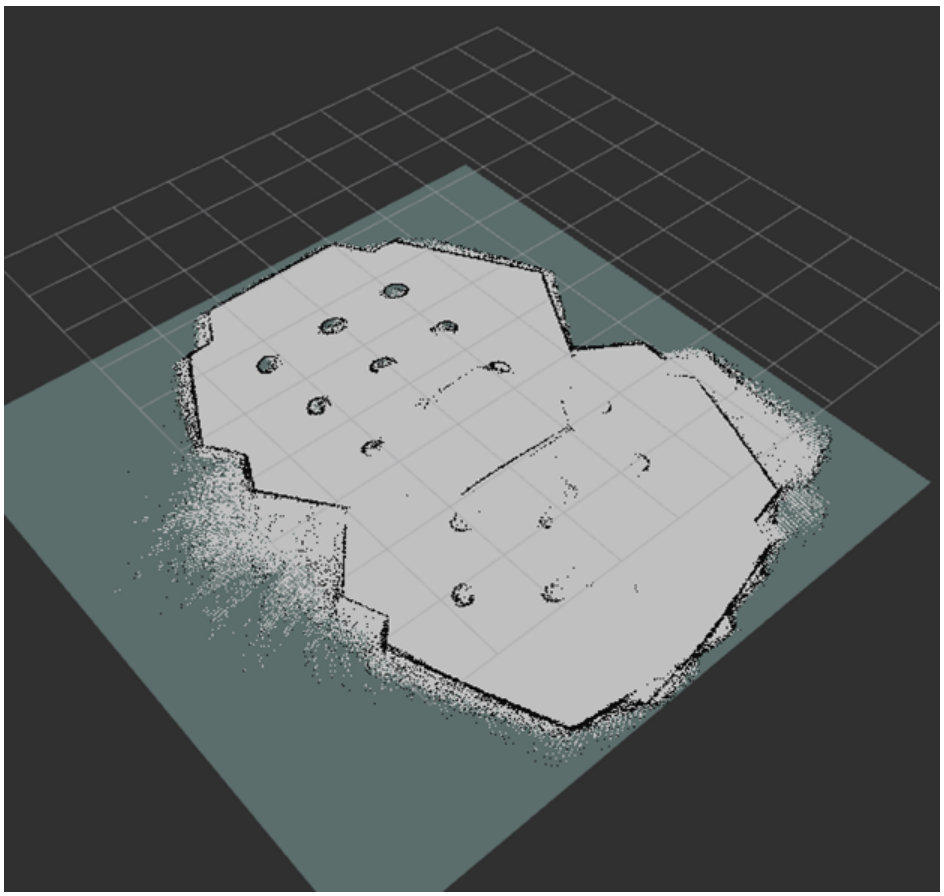

Figure 7: SLAM map drift when CPU contention delays scan processing.

7.2 Experiment Setup

- **Platform:** Windows 11 + WSL2 on i5-9300H, 16GB RAM; VM constrained to 1 vCPU, 4GB; ROS 2 Jazzy, Gazebo Harmonic, TurtleBot3 simulation, SLAM Toolbox (online async), RViz2.
- **SLAM stress config:** `mapper_params_online_async.yaml` set to aggressive values to raise CPU load: `resolution` 0.01, `map_update_interval` 0.1s, `minimum_time_interval` 0.5s, loop closing enabled with fine resolutions (`correlation_search_space_resolution` 0.005, `coarse_angle_resolution` 0.0349), `transform_publish_period` 0.02s.
- **Load Injection:** Synthetic `cpu_hog` (tight compute loop, configurable duty 60–100%) scheduled during 120s runs; metrics logged: `/scan` rate, jitter (std of Δt), CPU usage.
- **Control Space:** Adjust process priorities via `renice`: raise `slam_toolbox` (e.g., nice -5 to -10), lower hogs (e.g., nice 10 to 15). No kernel mods—pure user-space priority tuning.

7.3 RL Approaches

- **Bandit (ε-greedy):** Stateless; learns a single best priority action by updating $Q(a)$ with recent rewards. Fast but blind to changing contention.

- **Q-Learning:** State-aware with three states (Safe/Warning/Critical) defined by jitter; updates $Q(s,a)$ to consider future impact, so it can pre-emptively lower hog priority when jitter rises.

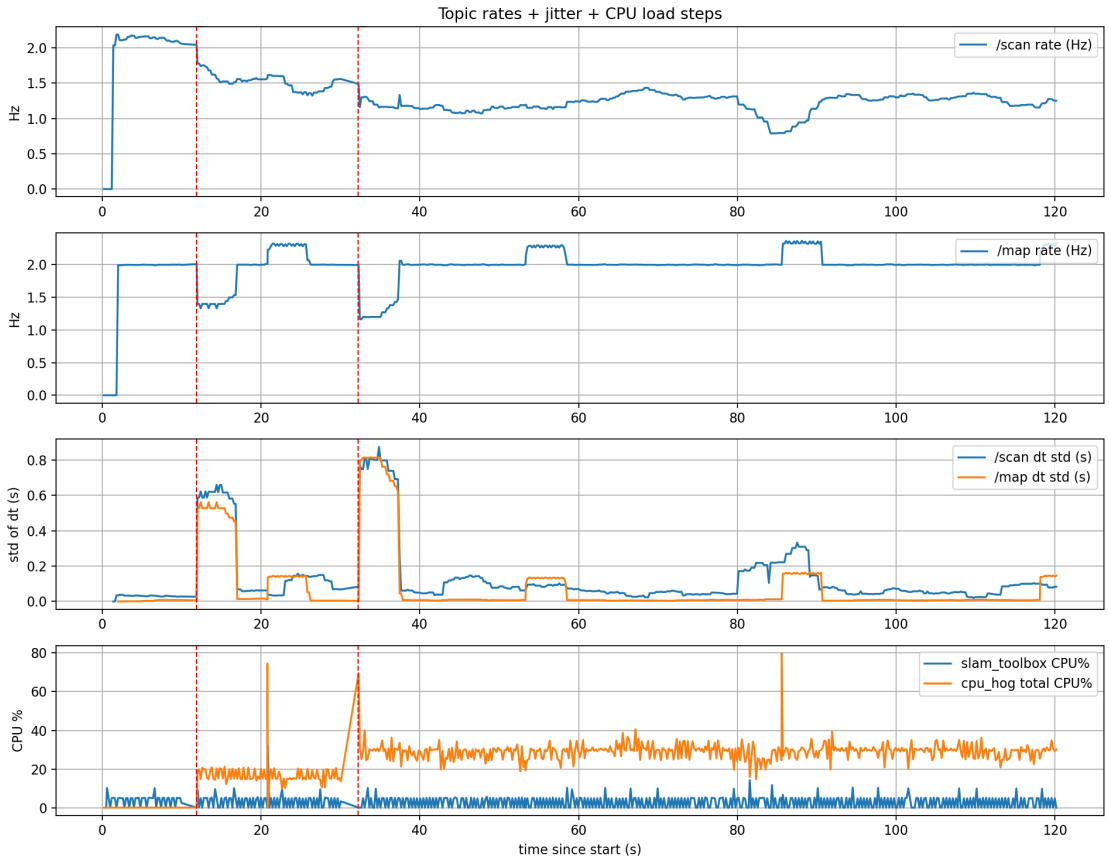

Figure 8: Bandit vs. Baseline: improved but still reactive.

7.4 Key Results

- **Baseline (no control):** Scan rate collapsed to ∼2.5Hz with sub-second jitter spikes; map unusable.
- **Bandit:** Stabilized scan rate ∼4.2Hz but jitter during hog phases remains high, peaking around 0.6 s std(Δt) (Fig. 8); map still noisy.
- **Q-Learning:** Held scan rate ∼4.8Hz; after the initial spike, jitter settles around 0.2–0.3 s std(Δt) (Fig. 9); SLAM keeps more CPU and map stays usable.

Method	Scan Rate (Hz)	Jitter (ms)	Map Quality
Baseline (None)	1.0 ± 0.8	~ 800	Poor (Drift)
Rule-based	1.1 ± 0.5	~ 600	Medium
Bandit (MAB)	1.3 ± 0.2	~ 600	Medium-High
Q-Learning	$\mathbf{1.5 \pm 0.2}$	~ 250	High (Stable)

Table 1: Estimated performance during a 120s contention episode from Figures 8 and 9. Jitter is the std of inter-scan arrival times; lower is better.

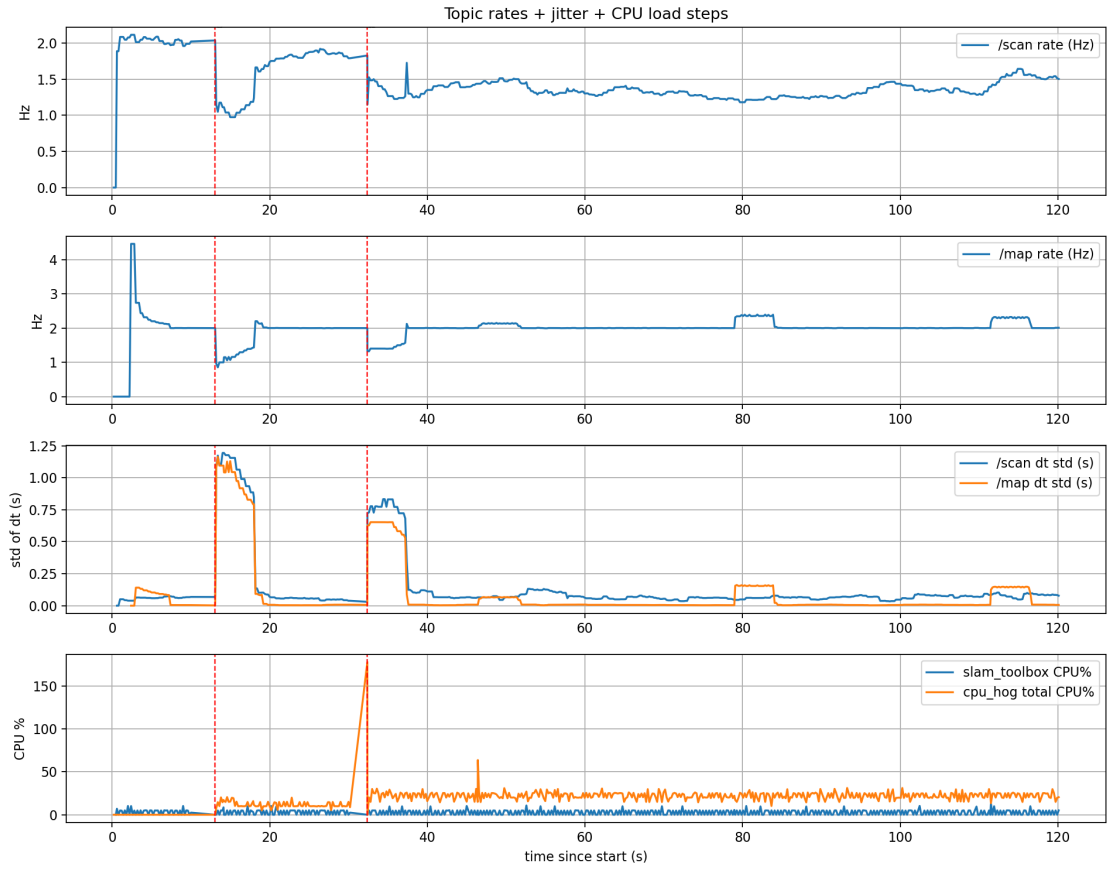

Figure 9: Q-Learning recovers quickly after contention spikes and stabilizes throughput.

7.5 Limitations and Next Steps

- WSL2 and Windows host noise limit determinism; real hardware or RTOS would give cleaner baselines.
- The TurtleBot3 world is relatively simple; in cluttered or real environments, map drift and contention effects would appear sooner.
- RL is reactive; initial spikes still occur before the agent intervenes.
- Future work: integrate policy with Nav2 controller priorities or cgroups, expand state to include SLAM timing/TF delays, and validate sim-to-real transfer.

8 Dynamic Obstacles with Nav2 Navigation

To better approximate real-world navigation (where people and objects move), I added a custom ROS 2 package that spawns and animates dynamic obstacles in Gazebo while the robot navigates to a goal using Nav2.

8.1 Implementation Overview

The package `tb3_nav2_dynamic` provides:

- **Dynamic obstacle node:** `dynamic_obstacles_node.py` spawns three box obstacles and moves them periodically.
- **Integrated launch:** `nav2_dynamic.launch.py` can start Gazebo, Nav2 bringup, RViz2, an initial pose publisher, and the dynamic obstacle node.

The obstacles are created using `ros_gz_sim create` (SDF model) and moved using `ros_gz_sim set_entity_pose`. Each obstacle follows a simple trajectory (line or circle) parameterized by a center point, amplitude, and period. The update loop runs at `update_rate_hz` (default 2 Hz).

8.2 How to Run

Prerequisite: A saved map YAML from the SLAM section (e.g., `$HOME/slam_rl_ws/maps/my_map.yaml`).

Build & Source the Workspace

```
cd ~/slam_rl_ws
colcon build --symlink-install
source install/setup.bash
```

Launch Gazebo + Nav2 + RViz + Dynamic Obstacles

```
export TURTLEBOT3_MODEL=burger
ros2 launch tb3_nav2_dynamic nav2_dynamic.launch.py \
  map:=$HOME/slam_rl_ws/maps/my_map.yaml \
  start_dynamic_obstacles:=true
```

Operation: In RViz2, set **2D Pose Estimate** (if needed) and then send a **Nav2 Goal**. The planner/controller should re-plan around moving boxes while executing the path.

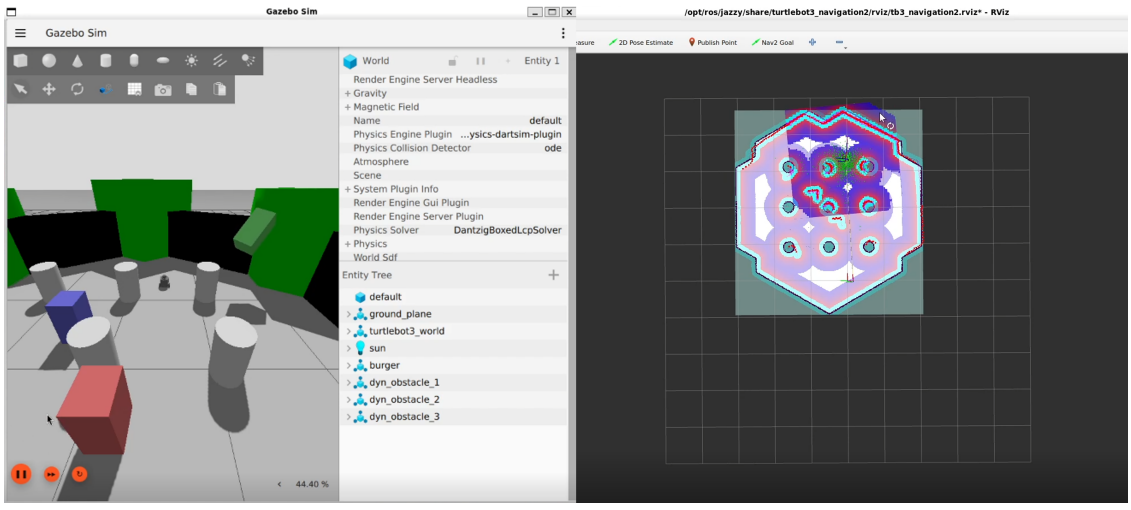

Figure 10: Dynamic obstacles spawned and moved in Gazebo during Nav2 navigation.

8.3 Notes

- If obstacles already exist in the world, set `spawn_obstacles:=false` and only update poses.
- The trajectories and count of obstacles can be modified in `dynamic_obstacles_node.py`.

9 Conclusion

I have completed building the Robotics development environment on WSL with ROS 2 Jazzy and Gazebo Harmonic. Issues regarding installation, environment variable configuration, and user privileges have been thoroughly resolved. The system is now capable of physical simulation, robot control, and stable sensor data acquisition, ready for more complex tasks in the subsequent phases.

Acknowledgements

I would like to express my sincere gratitude to Dr. **Do Quoc Huy** (SOICT, Hanoi University of Science and Technology – HUST) for his guidance and support during this project.

References

- ROS 2 Documentation: https://docs.ros.org/
- ROS–Gazebo Integration (`ros_gz`): https://docs.ros.org/en/ros2_packages/jazzy/api/ros_gz/
- Gazebo Harmonic ROS 2 Integration Guide: https://gazebosim.org/docs/harmonic/ros2_integration/
- Navigation 2 (Nav2) Documentation: https://docs.nav2.org/

A Appendix A: Step-by-Step RViz2 Configuration Guide

This appendix provides a detailed visual guide on configuring RViz2 to visualize the TurtleBot3, its sensor data, and the SLAM process.

A.1 Step 1: Launching RViz2

Open a new terminal and execute the following command to start the visualization tool:

Terminal: Start RViz2

```
ros2 run rviz2 rviz2
```

At startup, the main window will be empty. The first task is to configure the reference frame.

A.2 Step 2: Setting the Fixed Frame

By default, the Fixed Frame might be set to `map` or `world`. For basic telemetry and sensor testing, we switch this to the odometry frame.

1. Locate the **Displays** panel on the left side.
2. Expand **Global Options**.
3. Click on **Fixed Frame** and manually type or select **odom**.

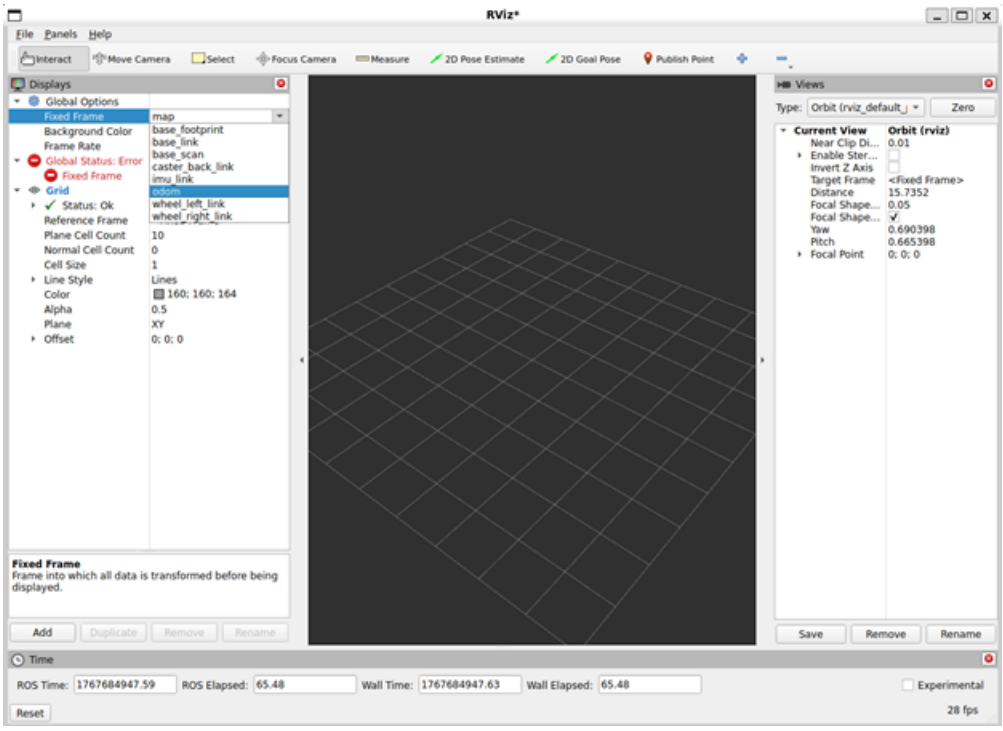

Figure 11: Changing the Fixed Frame to `odom`

A.3 Step 3: Visualizing LiDAR Data (LaserScan)

To see what the robot "sees", we need to add the LaserScan display.

1. Click the **Add** button at the bottom left.
2. Select the **By Topic** tab.
3. Find `/scan` and select **LaserScan**.
4. Click **OK**.

The red dots representing obstacle distances will appear on the screen.

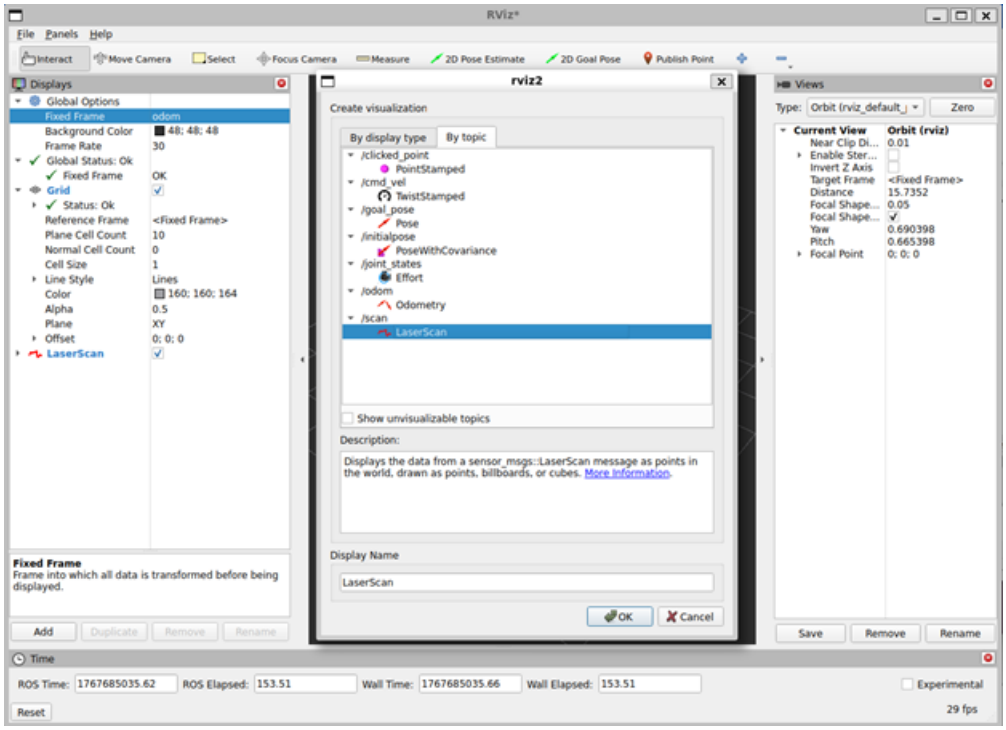

Figure 12: Adding the LaserScan topic to visualize LiDAR data

A.4 Step 4: Visualizing the Robot Model

To visualize the physical structure of the TurtleBot3 Burger:

1. Click the **Add** button again.
2. Select the **By Display Type** tab (or search in the list).
3. Select **RobotModel**.
4. *Optional:* Add **TF** (Transform Framework) to see the coordinate axes of the wheels and sensors.

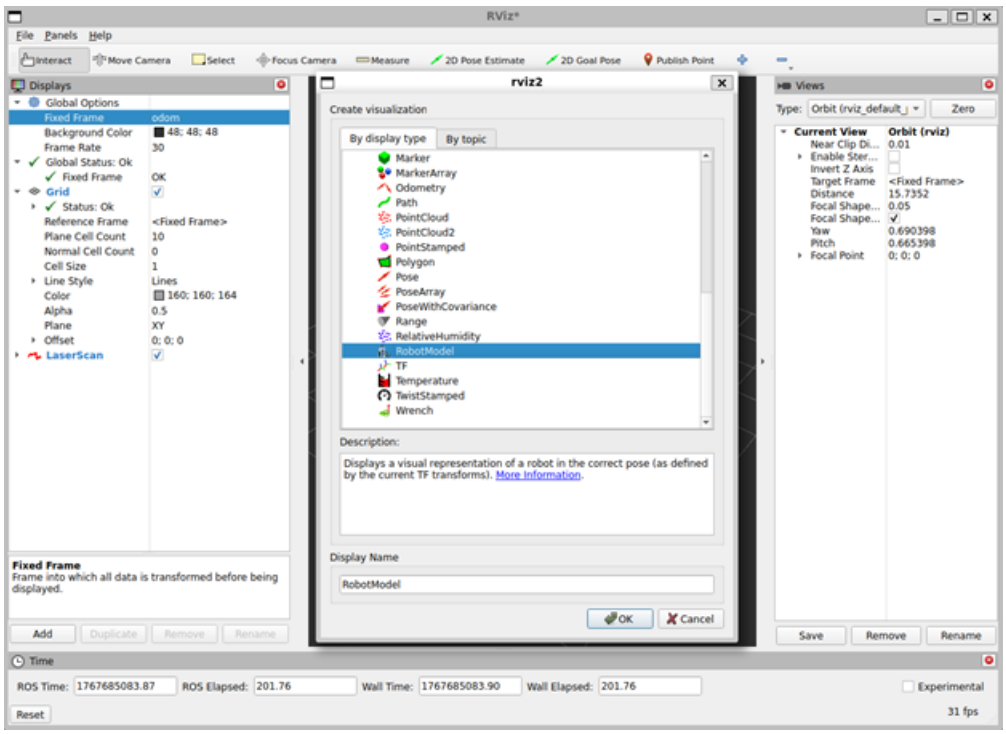

Figure 13: RobotModel added to the visualization

A.5 Step 5: Visualizing SLAM Map (Mapping Process)

When running SLAM (Simultaneous Localization and Mapping), the Fixed Frame must be set back to **map** to correctly overlay the generated map.

1. Change **Fixed Frame** back to **map**.
2. Click **Add** → **By Topic**.
3. Select `/map` → **Map**.
4. Set the **Update Interval** to 1 or 2 seconds for real-time updates.

As the robot moves, the map will transition from unknown (grey) to free space (white) and obstacles (black).

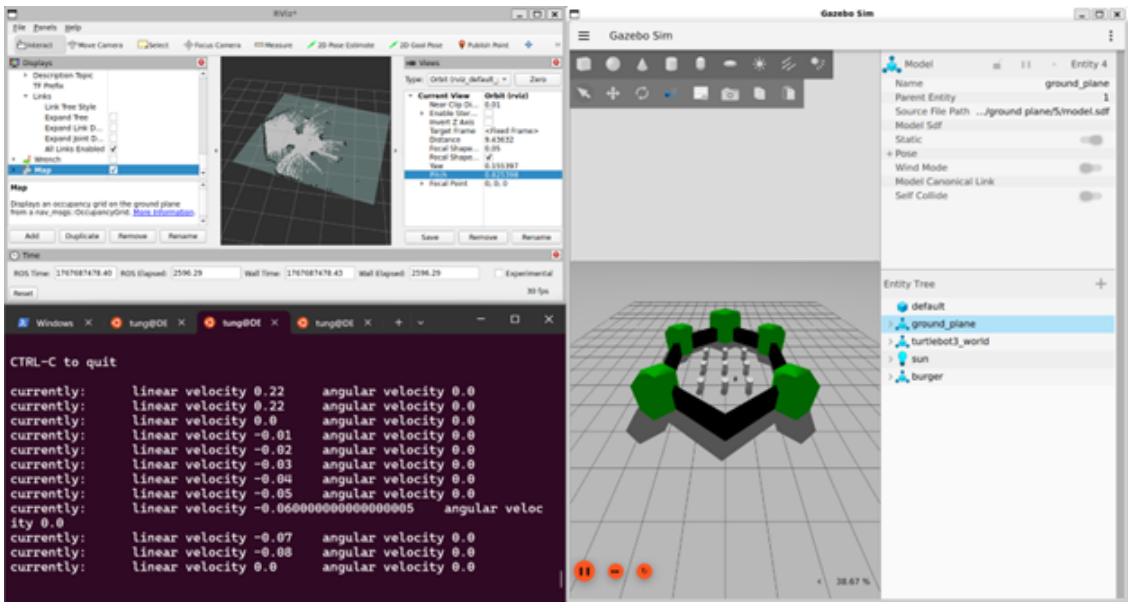

Figure 14: Real-time SLAM process visualizing the map and robot trajectory