Tutorial 6 ROS Programming – Simulation, SLAM and Navigation

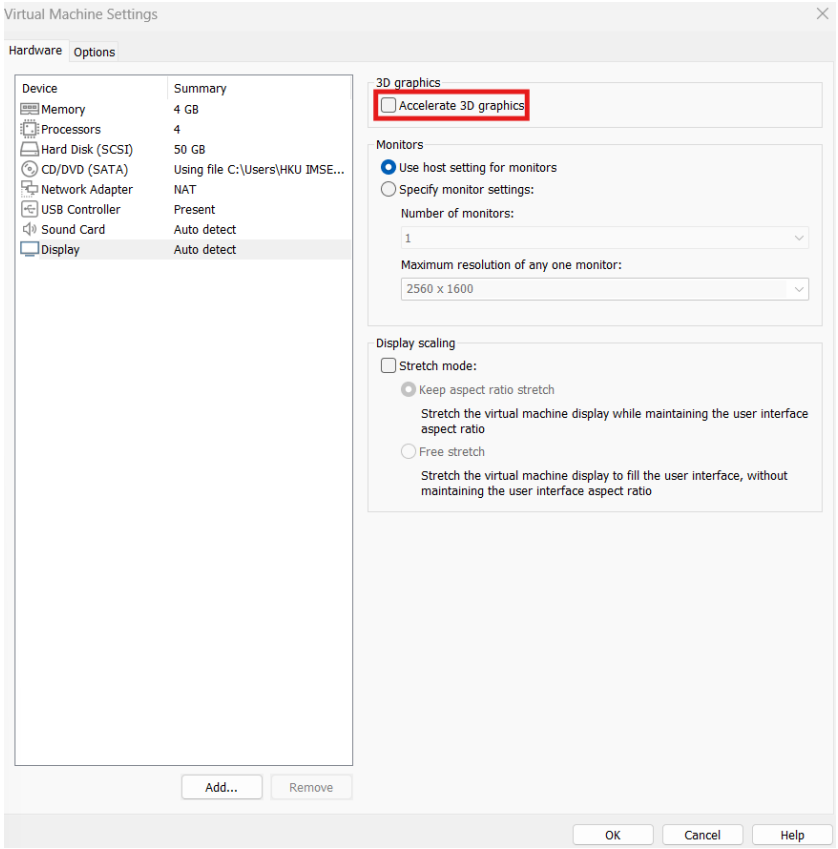
Objectives

The learning objectives are as follows:

- Learn how to use Gazebo to simulate the robot environment
- Learn how to implement SLAM to create a map using Gazebo and RViz
- Learn how to use Nav2 to navigate the robot in the map scanned by SLAM

1 Package Installation

1. Before power on the VM, right click the VM – Settings – Display, disable the "Accelerate 3D graphics".



2. Open a new terminal and install the required package by the following commands:

```
$ sudo apt-get update

$ sudo apt install ros-jazzy-turtlebot3 ros-jazzy-turtlebot3-gazebo ros-
jazzy-turtlebot3-description ros-jazzy-teleop-twist-keyboard ros-jazzy-
```

```
rviz2 ros-jazzy-slam-toolbox ros-jazzy-navigation2 ros-jazzy-nav2-bringup
ros-jazzy-turtlebot3-bringup
```

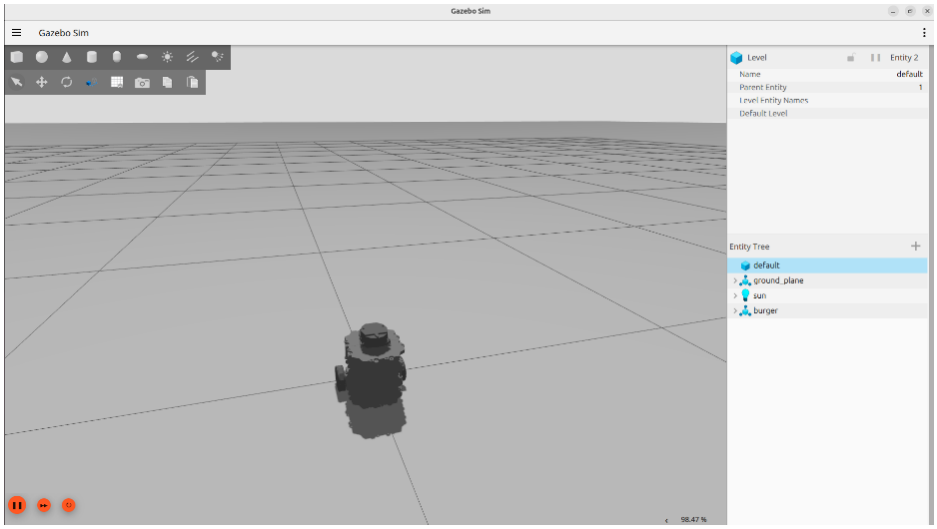
3. Source the ROS2 Jazzy's setup bash file and set the default robot model as "burger":

```
$ source /opt/ros/jazzy/setup.bash

$ export TURTLEBOT3_MODEL=burger
```

4. Try to launch an empty world with a burger turtle robot in Gazebo:

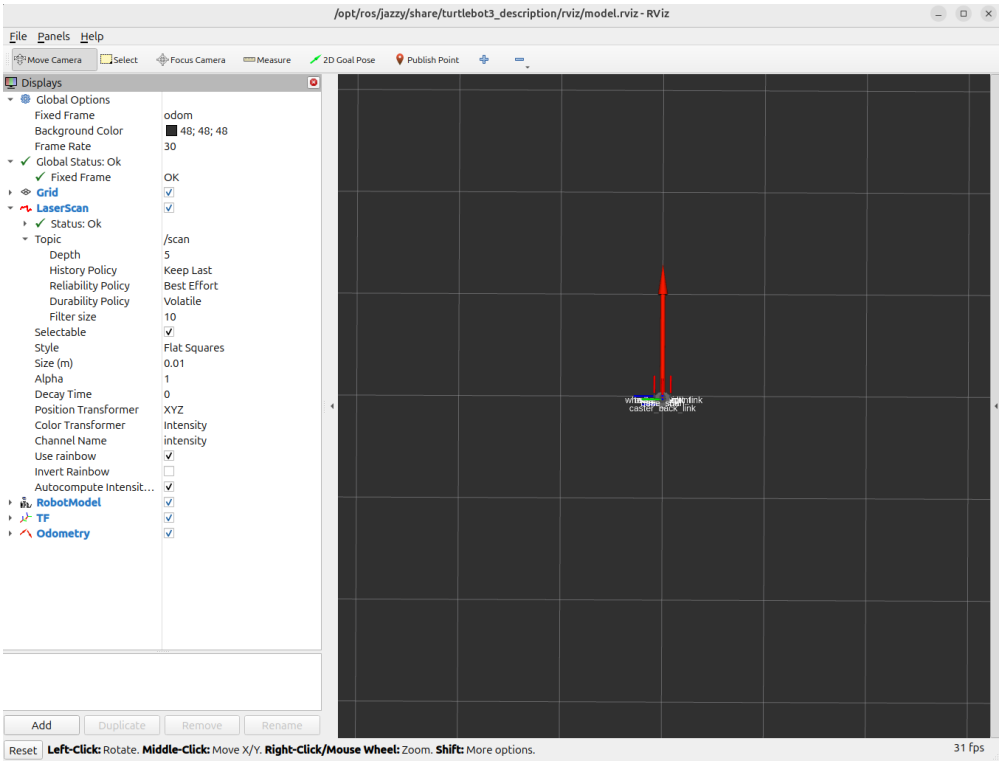
```
$ ros2 launch turtlebot3_gazebo empty_world.launch.py
```



5. Open a new terminal and try to launch the Rviz:

```
$ source /opt/ros/jazzy/setup.bash

$ ros2 launch turtlebot3_bringup rviz2.launch.py
```



6. Open a new terminal and try to use a keyboard controller to control the robot's movement:

```
$ source /opt/ros/jazzy/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ ros2 run turtlebot3_teleop teleop_keyboard
```

2 SLAM in ROS2 Jazzy

What is ***SLAM?***

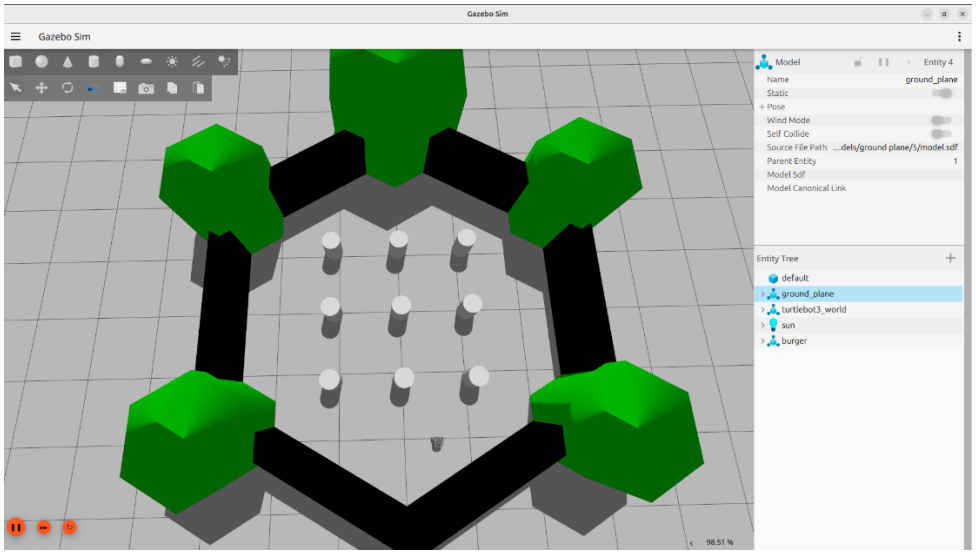
SLAM (Simultaneous Localization and Mapping) is a computational technique used in robotics and computer vision to enable a robot or device to create a map of an unknown environment while simultaneously determining its own location within that environment. It combines sensor data, such as from cameras, LIDAR, or sonar, to estimate the device's position and construct a consistent map in real-time. SLAM is critical for autonomous navigation in applications like self-driving cars, drones, and mobile robots, where GPS is unavailable or unreliable. Key challenges include handling sensor noise, computational complexity, and dynamic environments. Popular approaches include visual SLAM, LIDAR-based SLAM, and graph-based SLAM. In this course, LIDAR-based SLAM will be simulated as an example.

1. Close all the terminals opened previously.
2. Open a new terminal to launch a turtlebot3 world in Gazebo. Restart the Ubuntu if the Gazebo does not launch.

```
$ source /opt/ros/jazzy/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```



3. Open a new terminal and launch the slam toolbox to scan the map:

```
$ ros2 launch slam_toolbox online_async_launch.py
```

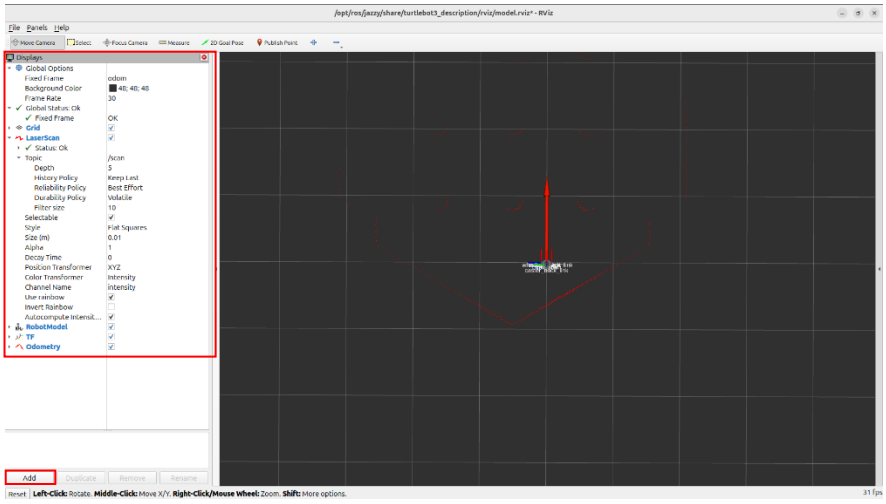
4. Open a new terminal and launch the Rviz to visualize the mapping process:

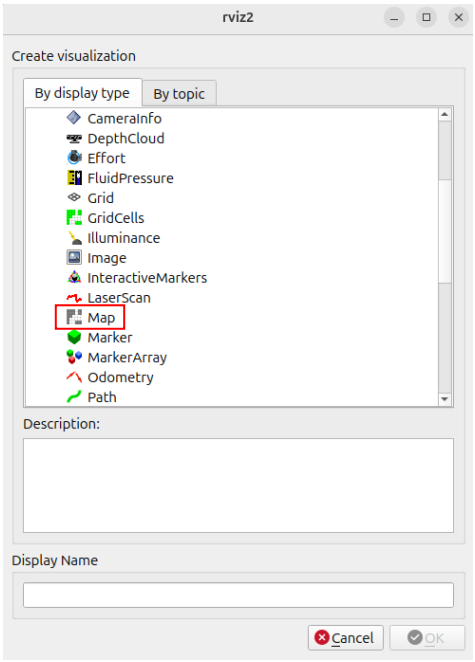
```
$ ros2 launch turtlebot3_bringup rviz2.launch.py
```

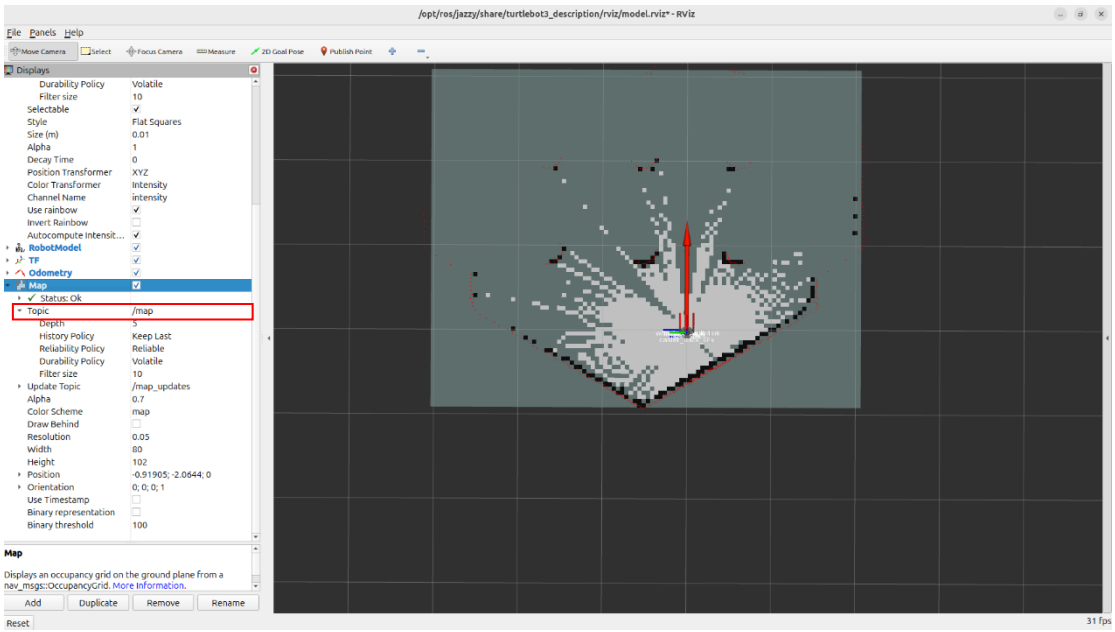
5. Configuration in Rviz:

   In the left Displays menu, ensure the following items are set correctly:

   i. Fixed Frame in Global Options is set as **"Odom";**
   ii. **"LaserScan",** if not exist, add it in By topic;
   iii. **"RobotModel",** if not exist, add it in By display type;
   iv. **"TF"** in By display type; (i, ii, iii, iv shall have been added by default)
   v. Add "Map", add it in By display type
   vi. Set the topic of Map as "/map"



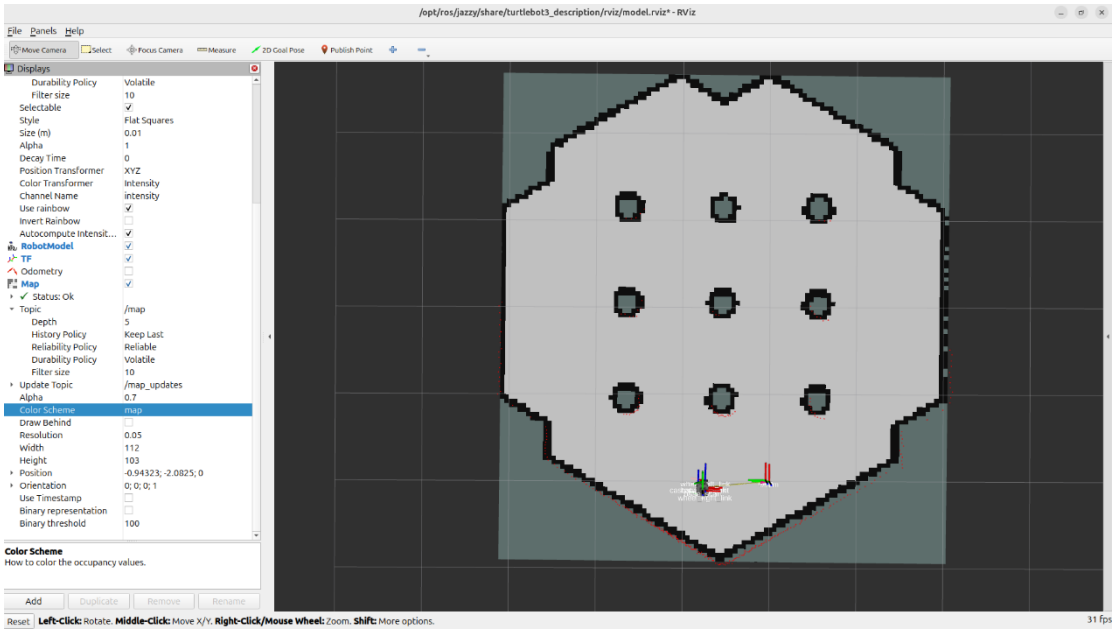

rviz2
Create visualization
By display type
By topic
CameraInfo
DepthCloud
Effort
FluidPressure
Grid
GridCells
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
Description:
Display Name
Cancel
OK



6. Use a keyboard controller to move the robot in the Gazebo world to scan the map:

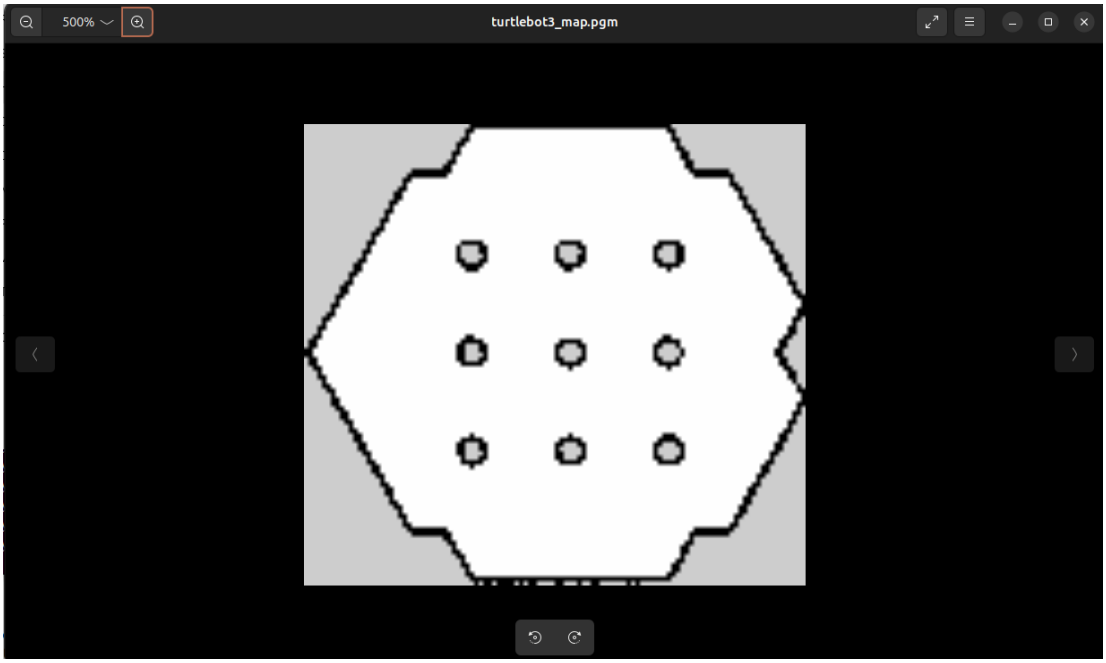
```
$ export TURTLEBOT3_MODEL=burger

$ ros2 run turtlebot3_teleop teleop_keyboard
```



7. Save the map as “turtlebot3_map” in the home folder:

```
$ ros2 run nav2_map_server map_saver_cli -f ~/turtlebot3_map
```



3 Nav2 in ROS2 Jazzy

What is ***Nav2?***

Nav2, or the ROS 2 Navigation Stack, is the next-generation navigation framework for ROS 2, designed to enable autonomous robot navigation in complex environments. Nav2 allows robots to localize themselves, plan paths, and navigate while avoiding obstacles, using sensor data from LIDAR, cameras, or odometry. Key components include:

- AMCL: Adaptive Monte Carlo Localization for estimating the robot's position in a known map.
- Global Planner: Computes optimal paths to a goal using algorithms like A* or Dijkstra's.
- Local Planner: Handles real-time obstacle avoidance and smooth trajectory generation (e.g., using DWA or TEB planners).
- Costmaps: 2D/3D maps for tracking free, occupied, and unknown spaces.

1. Restart Ubuntu.
2. Open a new terminal to launch a turtlebot3 world in Gazebo.

```
$ source /opt/ros/jazzy/setup.bash

$ export TURTLEBOT3_MODEL=burger

$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

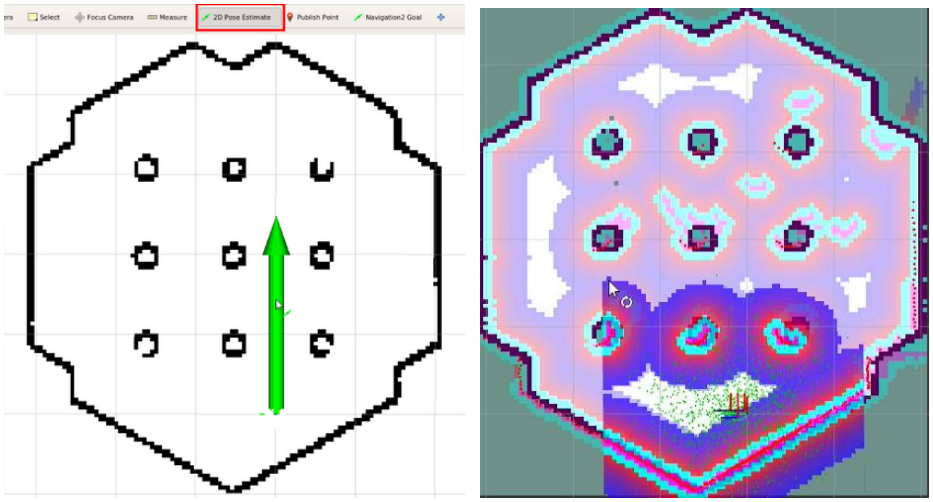
3. Open a new terminal to launch navigation node

```
$ source /opt/ros/jazzy/setup.bash

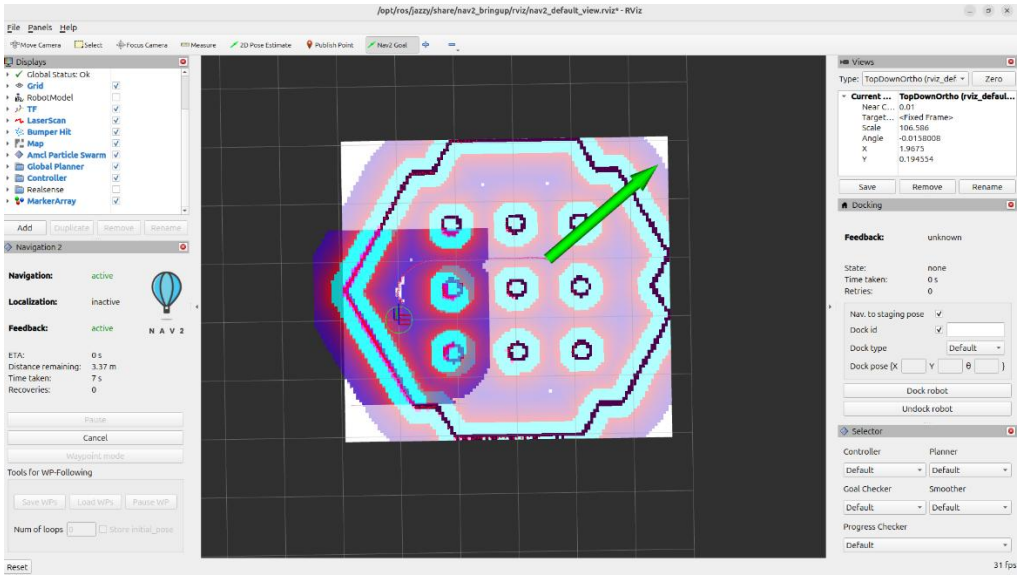
$ export TURTLEBOT3_MODEL=burger

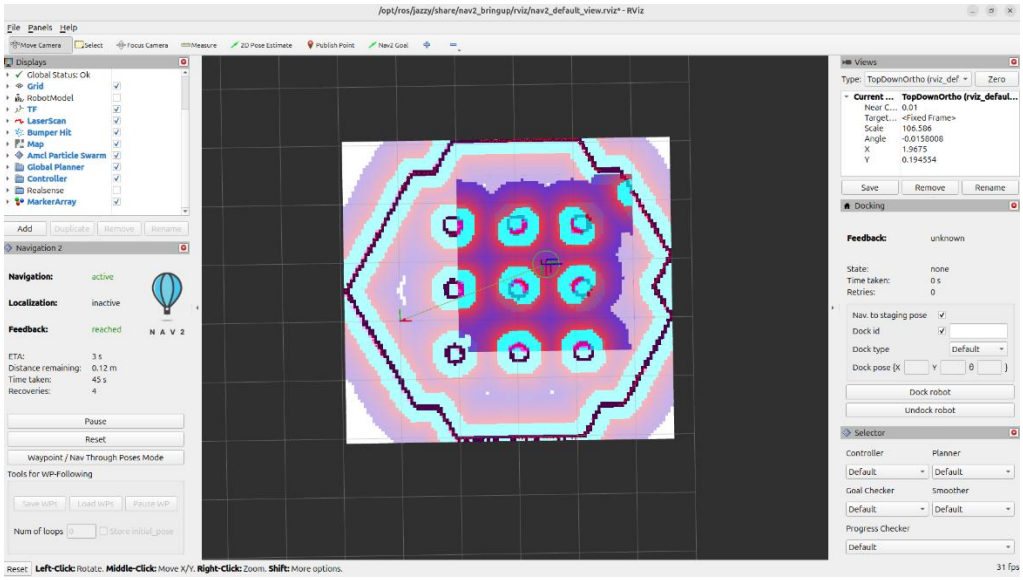
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py
use_sim_time:=True map:=$HOME/turtlebot3_map.yaml
```

4. Initial Pose Estimation must be performed before running Navigation as this process initializes the AMCL parameters that are critical for accurate Navigation. TurtleBot3 has to be correctly located on the map with the LDS sensor data that neatly overlaps the displayed map.
    a. Click the "2D Pose Estimate" button in the RViz2 menu.
    b. Click on the map where the actual robot is located and drag the large green arrow toward the direction where the robot is facing.
    c. Repeat step 1 and 2 until the LDS sensor data is overlaid on the saved map.



5. Click “Nav2 Goal” and long-click in the map and control the direction of the arrow to try navigation of the robot.





6. Nav2 parameters adjustment

Global Costmap:

- Purpose:
  Represents the entire known environment for high-level path planning from the robot's starting position to a goal.
- Scope:
  Covers a large, static area, typically based on a pre-existing map.
- Functionality:
  i. Used by the global planner (e.g., A* or Dijkstra's) to compute an optimal, long-term path to the goal.
  ii. Incorporates static obstacles (e.g., walls, furniture) and can be updated with new sensor data if the environment changes.
  iii. Typically has a lower resolution and updates less frequently to reduce computational load.

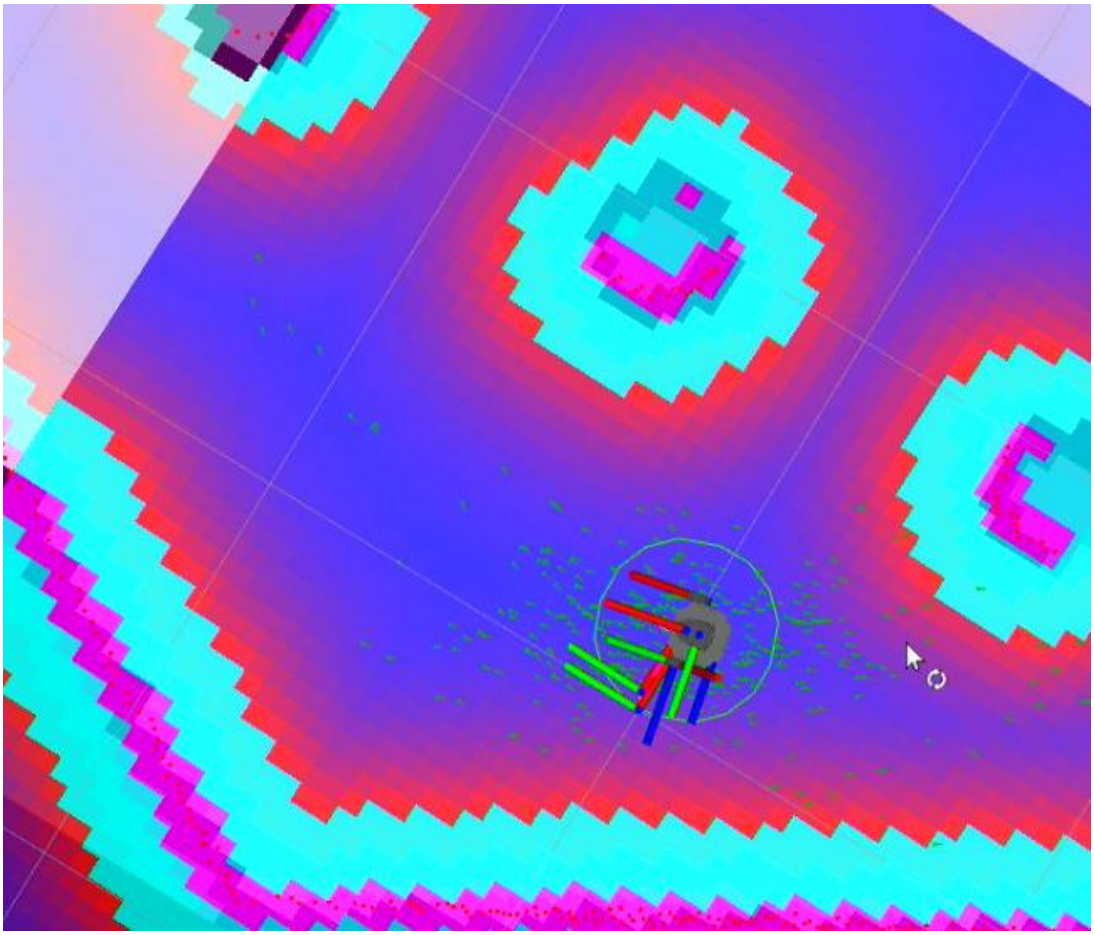
Local Costmap:

- Purpose:
  Represents a smaller, dynamic region around the robot for real-time obstacle avoidance and local trajectory planning.
- Scope:
  Covers a limited area around the robot, centred on the robot's current position.
- Functionality:
  i. Used by the local planner (e.g., DWA or TEB) to generate short-term, collision-free trajectories, adjusting for dynamic obstacles (e.g., people, moving objects).
  ii. Relies heavily on real-time sensor data (e.g., LIDAR) and updates frequently to capture environmental changes.
  iii. Typically has a higher resolution for precise navigation and uses a rolling window that moves with the robot.

Nav2 Parameters:

- **robot_radius**: the size of the robot (the green circle)

```
$ ros2 param set /local_costmap/local_costmap robot_radius 0.2
```



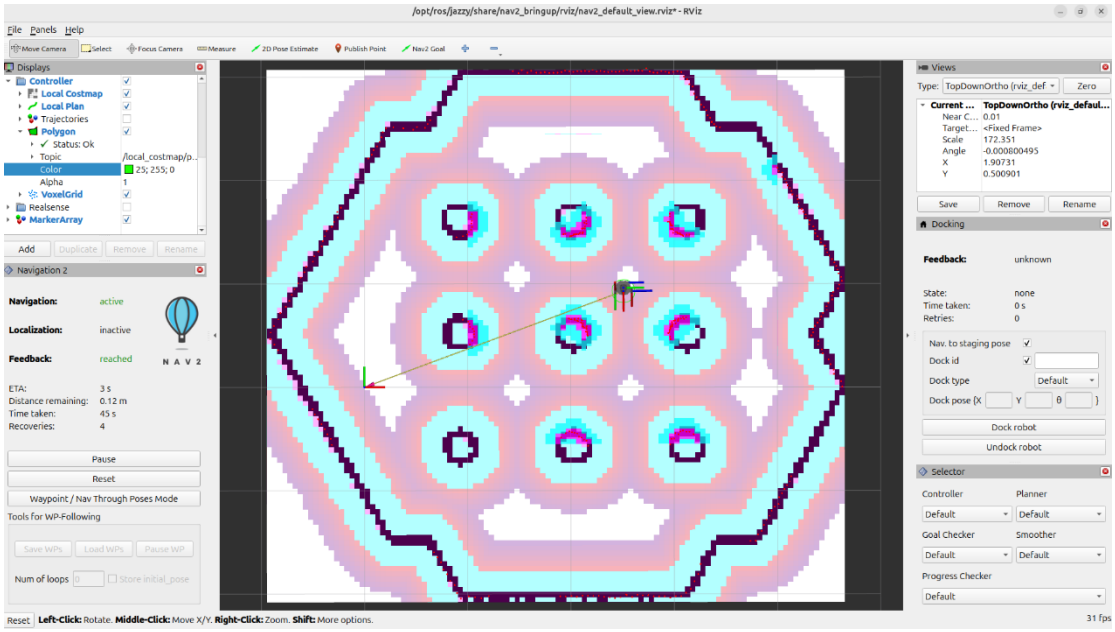
- **inflation radius**: The maximum distance from an obstacle where inflated costs are applied. It is used in the costmap (global/local) to enhance safe navigation by accounting for the robot's physical size and adding a safety buffer around obstacles.

```
$ ros2 param set /local_costmap/local_costmap
inflation_layer.inflation_radius 0.1

#Blue area

$ ros2 param set /global_costmap/global_costmap
inflation_layer.inflation_radius 0.5

#Purple area
```



- **Cost scaling factor:** Determines how quickly the cost decreases with distance. If it is set higher, the robot will avoid obstacles more aggressively. Otherwise, it will get close to obstacles more willingly.

```
$ ros2 param set /local_costmap/local_costmap
inflation_layer.cost_scaling_factor 2.0
```

Acknowledgement

1. Turtlebot – a robot kit with open-source software.

https://www.turtlebot.com/