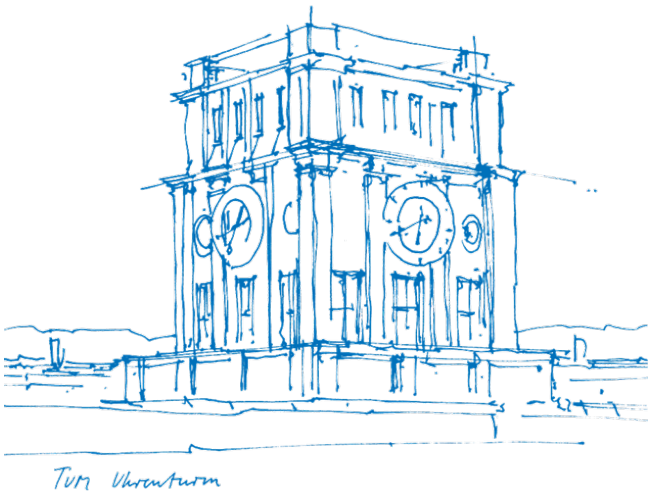
Chair of Computational Modeling and Simulation
TUM School of Computation, Information and Technology
Technical University of Munich

A Complete Guide to Unitree Go1 in ROS 2: From Simulation to Real-World Deployment

Panagiotis Petropoulakis
Mohammad Reza Kolani
Prof. André Borrmann
December 2025



Contents

1 Getting Started **1**
- 1.1 Hardware Components Overview . . . 1
  - 1.1.1 Unitree Go1 Robot . . . 1
  - 1.1.2 External Components of Go1 Robot . . . 3

2 Simulated Environment **5**
- 2.1 Gazebo Base Simulation . . . 5
- 2.2 Navigation and Mapping Overview . . . 7
  - 2.2.1 Coordinate Frames and Transforms . . . 7
  - 2.2.2 Nav2 Configuration . . . 8
  - 2.2.3 Example with TurtleBot3 . . . 8
  - 2.2.4 Installation and Technical Details . . . 9
- 2.3 Gazebo Complete Simulation . . . 10
  - 2.3.1 Building the Map . . . 10
  - 2.3.2 Processing the Map . . . 10
  - 2.3.3 Autonomous Navigation with Nav2 . . . 11
  - 2.3.4 3D SLAM and 2D Navigation . . . 11
  - 2.3.5 Performance Considerations . . . 12

3 Real World Experiments **13**
- 3.1 Set-up Overview . . . 13
- 3.2 Ouster LiDAR Configuration and Connection . . . 13
- 3.3 Robot Network Configuration and Monitoring . . . 15
  - 3.3.1 Ethernet Configuration . . . 15
  - 3.3.2 Remote Desktop Access to the Mini-PC . . . 15
  - 3.3.3 Connecting Additional PCs for ROS Topic Monitoring . . . 17
- 3.4 Navigation Demo . . . 18
- 3.5 Inspection Demo . . . 19

4 Conclusion and Future Work **22**
- 4.1 Conclusion . . . 22
- 4.2 Future Directions . . . 22

Bibliography **24**

1 Getting Started

In this section, we will discuss the specifications of the Unitree Go1 robot (Edu version) [1]. The Go1 is a four-legged, agile robot. We have additionally installed to the robot a range of external hardware components, including a Mini-PC, 3D LiDAR, and Intel RealSense camera. These components enable precise navigation, depth perception, and real-time obstacle avoidance, making the Go1 highly adaptable for various environments and tasks.

1.1 Hardware Components Overview

1.1.1 Unitree Go1 Robot

This section provides an overview of the hardware setup, internal configuration, power requirements, and operational modes for the Unitree Go1 robot.

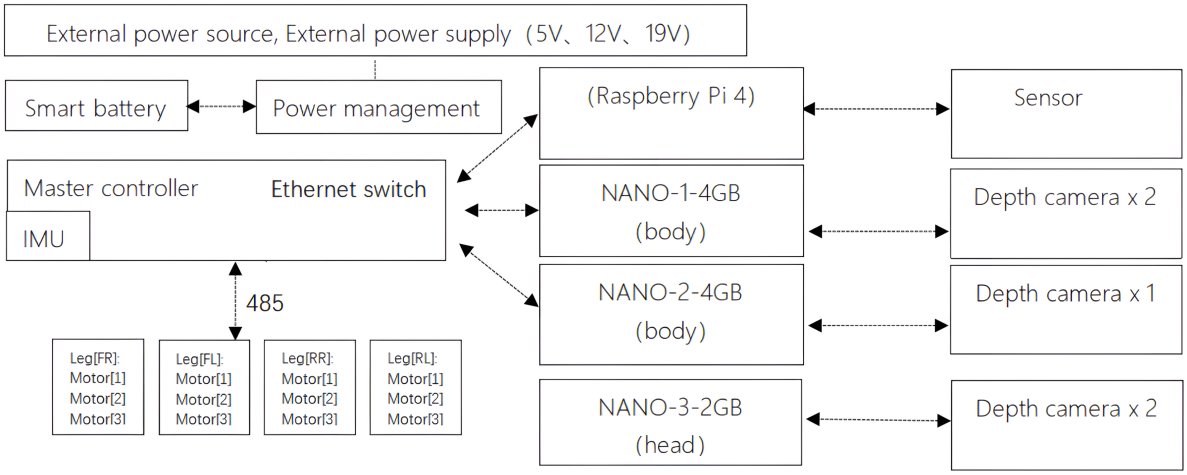


Figure 1.1 Go1 robot hardware architecture and control system overview.

General Specifications

The Go1 robot has a weight of 12 kg and is powered by a battery with a capacity of 6300 mAh and a voltage of 21.6 V. This battery supports up to 1 hour of continuous operation and approximately 1.5 hours in standby mode.

Camera System and Configuration

The Go1 is equipped with five groups of binocular fisheye cameras, distributed across various parts of the robot for optimal environmental awareness:

- **Head Nano** (IP: 192.168.123.13): Equipped with a front-facing camera (device ID=1) and a chin camera (device ID=0).
- **Body Nano** (IP: 192.168.123.14): Equipped with cameras on the left (device ID=0) and right (device ID=1) sides.

- **Main Body Nano** (IP: 192.168.123.15): Equipped with an abdominal camera (device ID=0).

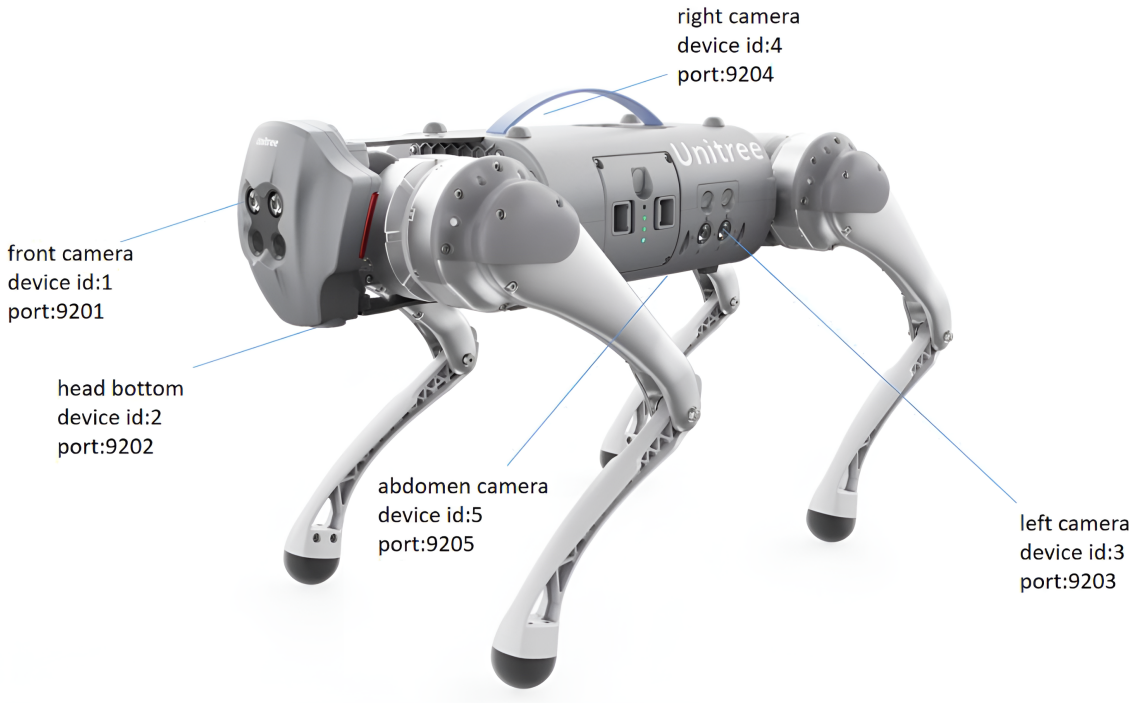


Figure 1.2 Camera setup of the Go1 robot.

The Go1 robot's camera hardware includes an RGB camera and a depth camera, each with specialized capabilities. The RGB camera features a stereo resolution of 1856×800, operates at 30 frames per second (fps), and uses global shutter technology for high-quality image capture with a wide field of view (FOV) of $222°$. The depth camera utilizes stereoscopic technology to measure distances effectively within a range of 10 cm to 85 cm, with a depth FOV of less than $180°$. Its output resolution is adjustable, with a default setting of 464×400 up to 1280×720, and it supports frame rates of up to 30 fps, enabling precise depth perception for navigation and obstacle detection.

Ultrasonic Sensors

The robot also includes three sets of ultrasonic modules for enhanced obstacle detection. An **ultrasonic sensor** is a device that detects objects and measures distances using high-frequency sound waves. It operates by emitting a burst of sound waves, which bounce off objects and return as echoes. By calculating the time it takes for these echoes to return, the sensor determines the distance to the object. Ultrasonic sensors are particularly useful for non-contact measurement, as they can detect objects without physical contact, making them ideal for applications in obstacle detection, level measurement, and industrial automation. They function reliably in various lighting conditions and typically operate within a short-to-medium range.

More precisely, the robot is equipped with:

- **Forward-Facing Ultrasonic Sensor**: Connected to the Head Nano (IP: 192.168.123.13), interface ttyTHS1.

- **Side Ultrasonic Sensors**: Left and right side ultrasonic sensors are connected to the Raspberry Pi (IP: 192.168.123.161).

Important Safety Notes

For optimal performance and safety, it is important to consider environmental and terrain restrictions when operating the robot. Avoid using the robot on low-friction surfaces, such as ice, and on soft or loose ground, as these conditions may lead to instability. On smooth surfaces, exercise caution to prevent slipping. Additionally, the robot should not be used on steps higher than 5 cm or on slopes greater than 25°, as these terrains can impact stability. For the best performance, operate the robot on flat, open areas whenever possible.

1.1.2 External Components of Go1 Robot

In our navigation setup, the primary computation and processing tasks are carried out on an external Mini-PC. This external computer runs **Ubuntu 22.04** and **ROS 2 Humble**, which handle real-time sensor data processing, communication, and control commands for the robot.

More precisely, the middle image (see Figure 1.3) showcases the main computational and sensor components of the robot system:

- **Mini-PC (ASRock 4x4 BOX-4800U)**: A compact Mini-PC running **Ubuntu 22.04** and **ROS 2 Humble**, used for sensor data processing and robot control. This is the core computational unit responsible for running all algorithms and processing data in real-time.
- **Intel RealSense D435i [2]**: The depth camera D435i combines the robust depth sensing capabilities of the D435 with the addition of an inertial measurement unit (IMU). It allows allows to detect obstacles and understand the environment in a three-dimensional space.
- **Ouster OS1-32 LiDAR [3]**: A high-resolution 3D LiDAR sensor used for scanning the environment. It is connected to the Mini-PC via Ethernet, enabling high-speed data transfer for real-time localization and mapping.
- **Power Supply**: Two rechargeable batteries are used to power the Mini-PC and the 3D LiDAR. The LiDAR requires 24V, while the Mini-PC runs on 19V.

To be more exact, the **Intel RealSense Depth Camera D435i** integrates advanced depth sensing, RGB imaging, and IMU tracking within a compact form factor. The stereo depth sensor provides high-resolution depth data at up to 1280×720 resolution, $87°\times58°$ Depth Field of View (FOV), and 90 fps, while maintaining depth accuracy within 2% at 2 meters. Complementing the depth sensor, the RGB camera captures high-quality color images at 1920×1080 resolution, $69° \times 42°$ FOV 30 fps, supporting a field of view optimized for accurate 3D perception. Additionally, the integrated IMU enables the camera to detect motion and rotation across 6 degrees of freedom (6DoF), essential for applications where spatial awareness is crucial.

The **OS1-32 3D LiDAR sensor** delivers exceptional mid-range optical performance, designed to operate effectively in various environmental conditions, including high sunlight (100 klx). For a target with 80% Lambertian reflectivity, the OS1 provides a range of up to 170 meters with a detection probability of over 90%. For darker targets with 10% Lambertian reflectivity, the range reaches 90 meters. The minimum detection range is configurable, with options from 0.0 meters (with defaults of 0.3 or 0.5 meters). The OS1's vertical resolution features 32 channels, and horizontal resolution options of 512, 1024, or 2048. Its rotation rate is also configurable at either 10 or 20 Hz. The sensor's field of view covers a full 360° horizontally and a $42.4° \pm 1.0°$ vertical range ($+21.2°$ to $-21.2°$), providing a broad perspective suited to complex 3D mapping and situational awareness. For precision, the OS1 offers an angular sampling accuracy of $\pm0.01°$ in both vertical and horizontal directions, with a range resolution of 0.1 cm.

In addition to the processing and sensor systems, essential safety and charging components play a crucial role in supporting the robot's operation. The right image (see Figure 1.3) illustrates these components:

- **Charging Station**: This is used to charge the Go1 robot's batteries, ensuring that they remain fully powered and ready for autonomous or long-duration tasks.

- **Emergency Stop (E-Stop)**: A critical safety feature that allows for an immediate stop of the robot's operations. This can be triggered either remotely or from a physical button mounted on the robot.

As far as the **power supply**, when integrating the Mini-PC and 3D LiDAR onto the robot, separate batteries are required for each device. The LiDAR uses a 24V power supply, while the Mini-PC is powered by a 19V battery.

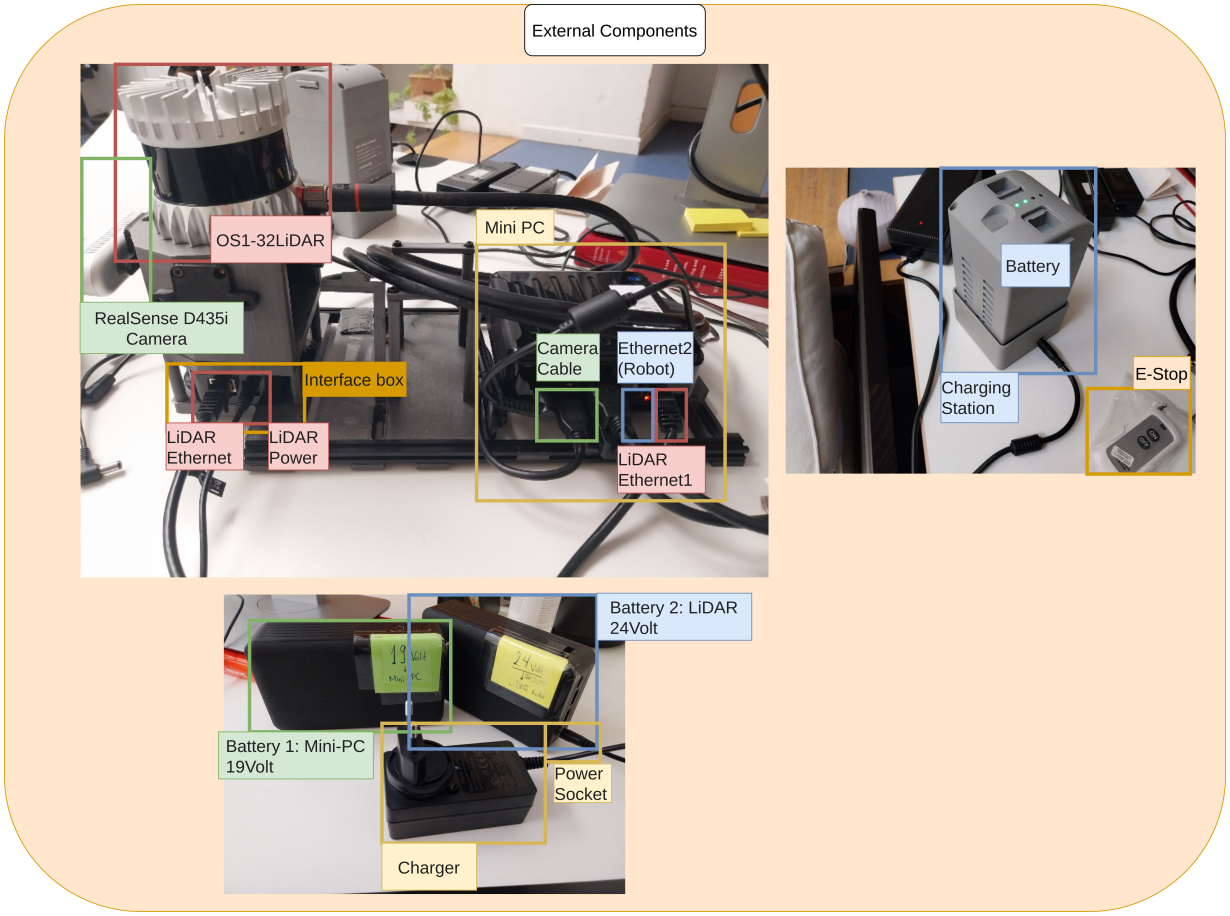


Figure 1.3 Overview of the Go1 robot external setup, including the processing system (Mini-PC), sensor system (RealSense camera, and LiDAR), the charging station with Emergency Stop, and the mounting configuration.

2 Simulated Environment

2.1 Gazebo Base Simulation

To develop the simulation setup for the Go1 robot in ROS 2, we extended and adapted repositories specifically designed for robotic control and simulation in ROS 2. Our primary foundation is the Unitree Go2 ROS 2 repository [4], which builds upon the CHAMP controllers [5] for quadrupedal control.

We updated the necessary descriptions to convert the robot model and its components to ROS 2 compatibility. This included the Ouster OS1-32 LiDAR and Intel RealSense D435i camera, which required specific repository extensions for accurate simulation and sensor performance.

For the integration of the on-robot depth cameras (5 in total), we utilized the gazebo_ros_pkgs repository [6], which provides the necessary plugins to visualize camera outputs in Gazebo. To enable joint control of the robot's legs, we incorporated gazebo_ros2_control [7]. For the Ouster LiDAR, we referred to the ouster_example repository [8] and tailored the URDF parameters to match our setup. Finally, to integrate the RealSense camera in the Gazebo environment, we used the IntelRealSense ROS wrapper [9] along with the realsense_gazebo_plugin [10]. It is important to note that we used the Foxy branch of the realsense_gazebo_plugin, which also works smoothly with ROS 2 Humble.

The odometry provided by the Go2 repository [4] using the CHAMP API [5] is quite noisy, making it less desirable for precise even very local localization. To address this, we switched to using pure LiDAR-based odometry, which, while more accurate, results in lower frame rates. Ideally, we would fuse odometry data with IMU measurements and periodically update them using LiDAR estimates at lower frequencies. This approach would allow the odometry to run at higher frame rates, around 100 Hz, or even higher. Currently, however, when relying solely on an ICP-based LiDAR solution, an odometry rate of more than 20 Hz should not be expected. This is the maximum output rate of the Ouster sensor. For more detailed information about odometry in ROS, consider reviewing the official ROS odometry documentation [11].

The Iterative Closest Point (ICP) algorithm [12] is a widely used method for aligning two point clouds by iteratively refining the transformation (rotation and translation) that minimizes the distance between corresponding points. The primary steps of the ICP algorithm are as follows:

1. **Find Correspondences:** For each point in the source point cloud, identify the closest point in the target point cloud. This step establishes pairs of corresponding points.
2. **Outlier Rejection:** Remove pairs of points that are considered outliers based on pruning techniques, such as normals compatibility or appearance-based matching. This helps in improving the accuracy of the alignment.
3. **Estimate Transformation:** Compute the optimal rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$ that minimize the mean squared error between the corresponding points.
4. **Apply Transformation:** Update the source point cloud by applying the estimated transformation.
5. **Check for Convergence:** Determine if the algorithm has converged based on a predefined criterion (e.g., change in error below a threshold). If not, repeat the process from step 1.

The goal of ICP is to find the rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$ that minimize the following cost function:

$$\min_{\mathbf{R},\mathbf{t}} \sum_{i=1}^{N} \|\mathbf{R}\mathbf{p}_i + \mathbf{t} - \mathbf{q}_i\|^2 \tag{2.1}$$

where:

- $\mathbf{p}_i$ are the points in the source point cloud.
- $\mathbf{q}_i$ are the corresponding points in the target point cloud.
- N is the number of corresponding point pairs.
- $\| \cdot \|$ denotes the Euclidean norm.

The optimal transformation, once computed using Singular Value Decomposition (SVD) or iterative optimization methods like Gauss-Newton on the covariance matrix of the corresponding points, provides the best-fit rotation and translation to align the source point cloud to the target, as shown in Figure 2.1. This transformation not only aligns the two point clouds but can also serve as the odometry information for the robot. Specifically, the computed translation and rotation represent the robot's estimated movement relative to its previous position, making it a key input for tracking the robot's pose during navigation.

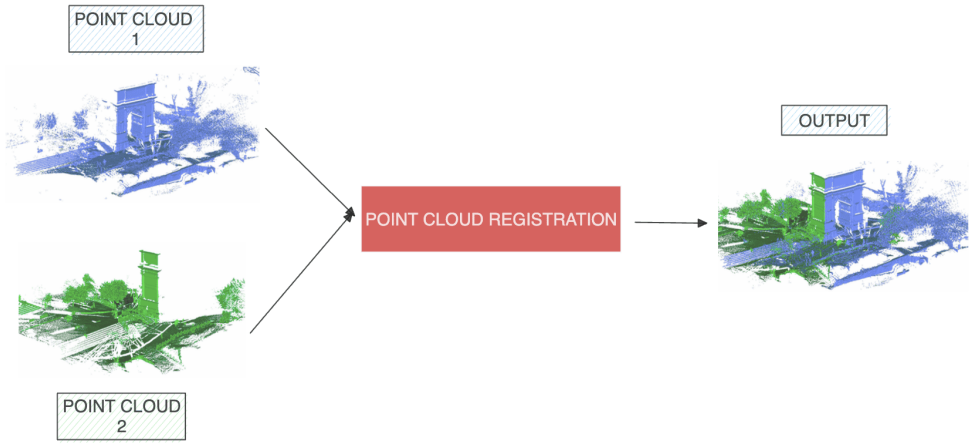


Figure 2.1 Illustration of the Iterative Closest Point (ICP) algorithm aligning two point clouds. Point Cloud 1 (blue) represents the source, while Point Cloud 2 (red) represents the target. The ICP algorithm iteratively refines the transformation to minimize the distance between corresponding points, resulting in the alignment of the two point clouds [13].

Now that we have set up the robot descriptions, Gazebo simulation environment, and odometry estimation pipeline, we are ready to begin the simulation. Run the following commands to spawn the robot description in the Gazebo simulator. After a 20-second delay, the `icp_odometry` node from the `rtabmap_odom` [14] package will be launched, using 3D LiDAR data to estimate the robot's odometry through ICP registration.

```
$ cd unitree_go1/sim
$ rm -rf build/ && rm -rf install/ && rm -rf log/ && clear
$ source /usr/share/gazebo/setup.sh && colcon build
$ source install/setup.bash && ros2 launch go1_config gazebo.launch.py
```

2.1 Start the base Go1 simulation

If you would like to run the simulation in debug mode, simply add the `-d` flag at the end of the launch file command. Additionally, you can use the `rviz:=true` option to visualize the robot and its topics in RViz [15]. By default, the robot description, Ouster lidar point cloud, and RealSense camera image output are displayed.

To launch also the simulation with a different world, you can append the `world:=` argument. You may provide an absolute path or modify the default hardcoded path specified in the `gazebo.launch.py` file. For example:

```
ros2 launch go1_config gazebo.launch.py world:=/home/go1/Desktop/workspace
    /unitree_go1/sim/unitree-go2-ros2/robots/configs/go1_config/worlds/
    complete.world
```

2.2 Launching with a custom world

To control the robot via teleoperation, you can use the `teleop_twist_keyboard` [16] package. First, open a new terminal and run the following command:

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

2.3 Enable teleoperation

2.2 Navigation and Mapping Overview

Now that we have a basic simulation set up, the next step is to enable the robot to navigate autonomously in an unknown environment while building a map. Navigation2 (often referred to as Nav2) [17] is the ROS 2-based navigation stack designed to provide autonomous navigation capabilities for robotic platforms (see Figure 2.2). It handles essential tasks such as localizing the robot within a given map (however, in our case, we will employ a different framework to achieve this), planning paths, and controlling the robot's motion to reach designated goals while avoiding obstacles.

To enable mapping and localization in our system, we utilize RTAB-Map (Real-Time Appearance-Based Mapping) [18], a graph-based Simultaneous Localization and Mapping (SLAM) framework. RTAB-Map employs sensor data such as images, LiDAR, or other modalities to construct maps of the environment while tracking the robot's position in real-time. By detecting loop closures—instances where the robot revisits previously mapped areas—the framework enhances map accuracy and minimizes localization drift. It supports various odometry techniques (visual, LiDAR-based, or combined) to estimate motion between frames and can produce both 2D and 3D occupancy grids. RTAB-Map's robust optimization capabilities make it well-suited for navigating complex environments and performing long-term tasks. Additionally, its modular design ensures compatibility with a wide range of sensors and seamless integration into ROS-based applications.

Before diving into the details of navigation and mapping, it is worth noting a recommended direction for future work: addressing the limitations of LiDAR-based odometry, which can be computationally intensive and slow for high-frequency updates. A popular solution is to integrate state estimation techniques to enhance localization accuracy by incorporating IMU measurements. One approach is to leverage the `robot_localization` package [19] in ROS, which provides non-linear state estimation tailored for robots operating in 3D or 2D environments. This package enables the fusion of data from multiple sensors, including IMU, odometry, and GPS, to produce a more robust and precise estimate of the robot's position and orientation. Such enhancements would mitigate the performance bottlenecks of LiDAR-based methods and significantly improve the reliability of the localization system, especially in dynamic or complex environments.

2.2.1 Coordinate Frames and Transforms

To work correctly, Nav2 relies on a series of coordinate frames and corresponding transforms. Common frames include:

- **map** → **odom**: Provides a globally-consistent reference frame of the environment.
- **odom** → **base_link**: The odometry frame holds locally-consistent pose information of the robot. The `base_link` frame is fixed to the robot's main body, usually at its rotational center.

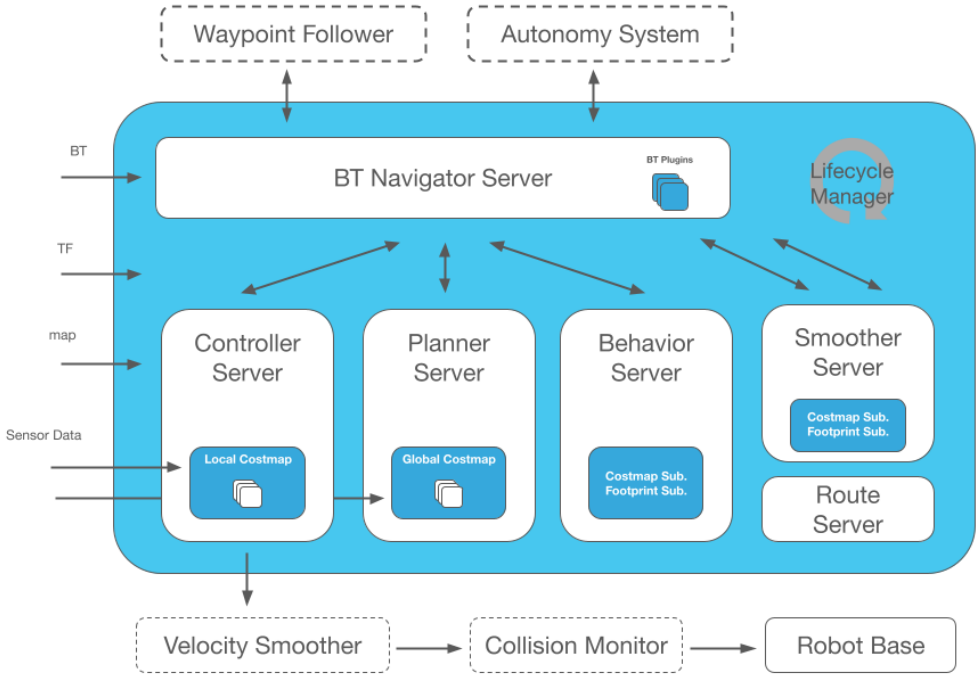


Figure 2.2 Navigation2 Block Diagram [17] illustrating its core functionalities. It supports diverse robot types, enables localization, path planning, collision avoidance, and behavior trees for customizable robot behaviors.

- **base_footprint**: A virtual link representing the robot's projection onto the ground plane. It's primarily used for planning footprints and collision detection, enabling more accurate path planning in Navigation2.

In short, **map** is a global reference frame, **odom** is a local frame that drifts over time but remains continuous, and **base_link** is attached directly to the robot. For obstacle avoidance and navigation computations, Nav2 often uses **base_footprint** as a simpler geometric representation of the robot, typically defined as a polygon with specified vertices or, in simpler cases, just a radius from the center of the reference frame.

2.2.2 Nav2 Configuration

Nav2 is highly configurable, with default parameters and example launch files available in the Nav2 bringup repository [20]. For tuning and improving navigation performance, detailed guidelines can be found in the Navigation Tuning Guide [21] publication. For a practical tutorial on Navigation2, refer to the Navigation2 Tutorial by Robotics Back-End [22].

2.2.3 Example with TurtleBot3

To illustrate how the Navigation2 stack operates, you can use the TurtleBot3 platform [23], a modular and versatile mobile robot designed for ROS 2. TurtleBot3 provides an accessible way to simulate and experiment with navigation concepts, such as mapping, path planning, and autonomous navigation. Here's a step-by-step guide to set up a basic example:

First, **launch the TurtleBot3 Simulation:** begin by launching the Gazebo simulation environment with a predefined TurtleBot3 world:

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

2.4 Run TurtleBot3 world simulation

After that, **build the Map with Cartographer:** use the Cartographer SLAM framework to construct a map of the environment in real-time while the robot navigates via pure teleoperation (use your keyboard):

```
$ ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time
   :=true
```

2.5 Run Cartographer for mapping

The Cartographer processes sensor data such as laser scans and odometry to create a consistent map of the environment.

Following, **save the Generated Map:** once the mapping is complete, save the map to a file for later use during navigation:

```
$ ros2 run nav2_map_server map_saver_cli -f /home/go1/.../my_map
```

2.6 Save the map

Finally, **start the Navigation2 Stack:** launch the Navigation2 stack to enable autonomous navigation with the saved map:

```
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=
   True map:=/home/go1/.../my_map.yaml
```

2.7 Run Nav2 for navigation

In this step, the robot uses the provided map for localization, path planning, and dynamic obstacle avoidance to reach goals. You can transition to the RViz panel, and use the "Nav2 Goal" tool to set navigation targets. The robot will autonomously navigate to the selected locations while avoiding obstacles.

2.2.4 Installation and Technical Details

To get started with Navigation2 and TurtleBot3 support, you first need to install the required ROS 2 packages:

```
$ sudo apt install -y ros-humble-turtlebot3
$ sudo apt install -y ros-humble-navigation2 ros-humble-nav2-bringup
```

2.8 Install Navigation2 and TurtleBot3 Packages

Next, set your TurtleBot3 model. For the previous example, we have used the `waffle` model, a popular variant of TurtleBot3:

```
$ echo "export TURTLEBOT3_MODEL=waffle" >> ~/.bashrc
$ source ~/.bashrc
```

2.9 Set TurtleBot3 Model

The `export` command sets the environment variable `TURTLEBOT3_MODEL` to `waffle`, ensuring that all related configurations use this model. The variable is added to `.bashrc` to make it persistent across terminal sessions. Remember to source the file for the changes to take effect.

When using RViz to visualize and interact with the robot, it is crucial to set the *Fixed Frame* to `map` to avoid unintended rotating effects caused by mismatched reference frames. Follow these steps to configure RViz:

1. Open RViz by running in the terminal: rviz2
2. In the left-hand panel, go to **Global Options**.
3. Set the *Fixed Frame* field to `map`.

ROS 2 also uses the Data Distribution Service (DDS) [24, 25] middleware for communication. The default DDS vendor for ROS 2 Humble is FastDDS, but some users experience better compatibility and fewer issues with CycloneDDS. To switch to CycloneDDS:

```
$ sudo apt install -y ros-humble-rmw-cyclonedds-cpp
$ echo "export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp" >> ~/.bashrc
$ source ~/.bashrc
```

2.10 Set DDS communication

This change can improve stability and performance of Nav2 in some environments.

2.3 Gazebo Complete Simulation

Now that we have an understanding of the basic simulation setup and the functionality of the Navigation2 stack, we can proceed to integrate these concepts for a complete simulation of our robot in Gazebo. The first step in this process is to create a map of the environment using RTAB-Map SLAM [18]. This mapping process will enable the robot to localize itself.

2.3.1 Building the Map

To launch the SLAM node, open a new terminal. Before proceeding, ensure that the basic simulation is running by executing the following command:

```
$ ros2 launch go1_config gazebo.launch.py
```

2.11 Launch Gazebo Simulation

Once the simulation is running, you can launch the SLAM node by executing:

```
$ source install/setup.bash
$ ros2 launch go1_config slam.launch.py restart_map:=true use_rviz:=false
```

2.12 Launch RTAB-Map SLAM

This command initiates the SLAM process, allowing the robot to build a detailed map of its surroundings (using the keyboard teleoperation). Once the map is created and saved, it becomes the foundation for enabling autonomous navigation using the Navigation2 stack.

The expected output in the terminal during the SLAM process might look like this:

```
[rtabmap-1] [INFO] [1733930863.083438453] [rtabmap]: rtabmap (2):
Rate=1.00s, Limit=0.000s, Conversion=0.0004s, RTAB-Map=0.0539s,
Maps update=0.0001s pub=0.0011s (local map=1, WM=1)
[rtabmap-1] [INFO] [1733930865.259253867] [rtabmap]: rtabmap (3):
Rate=1.00s, Limit=0.000s, Conversion=0.0004s, RTAB-Map=0.0453s,
Maps update=0.0001s pub=0.0026s (local map=1, WM=1)
...
```

Upon closing the SLAM program, the generated map is typically saved at the `~/.ros/` folder location.

2.3.2 Processing the Map

To process and refine the map further, you can use the RTAB-Map GUI. The GUI provides tools for exporting the map's point cloud and inspecting its details. To open the RTAB-Map database in the GUI, use the command:

```
$ cd ~/.ros
$ rtabmap rtabmap.db
```

2.13 Open RTAB-Map database in GUI

Within the GUI, you can export the map's point cloud by navigating to `File > Export 3D Clouds`. Ensure that the first option in the export dialog is unchecked to avoid errors such as:

```
"Could create clouds for 1 node(s). You may
want to activate clouds regeneration option."
```

Disabling this option ensures successful export since the scan does not use any RGB-D images.

2.3.3 Autonomous Navigation with Nav2

The next step is to launch the Navigation2 [22] stack for autonomous navigation. Open a new terminal and run:

```
$ source install/setup.bash && ros2 launch go1_config navigate.launch.py
```

2.14 Launch Navigation2

Once RViz is open, use the `Nav2 Goal` tool on the top toolbar to set a navigation goal in the map. The robot will then autonomously navigate to the specified goal while avoiding obstacles in its path. An overview of the complete simulation setup is shown in Figure 2.3.

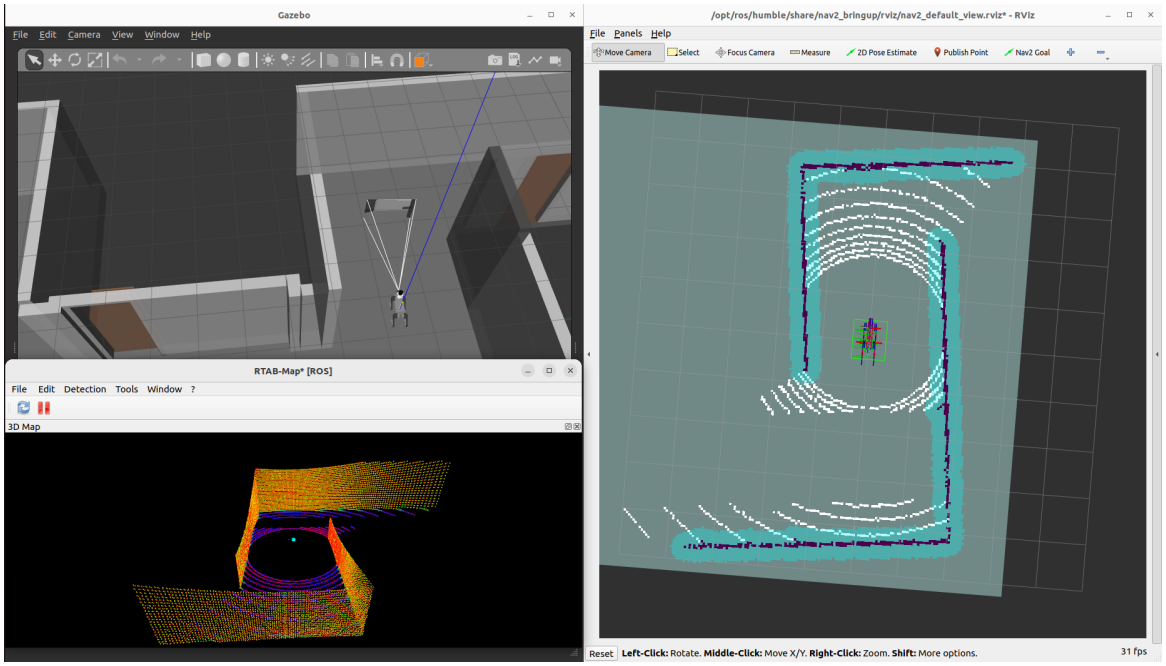


Figure 2.3 Overview of the Gazebo simulation with Navigation2 [22] and RTAB-Map SLAM [18].

2.3.4 3D SLAM and 2D Navigation

In this setup, RTAB-Map [18] handles 3D localization, while AMCL-based localization from Navigation2 [22] is disabled, as shown in Figure 2.4. If you wish to implement a 2D-only solution, consider the following:

- Use a state estimator for odometry that does not rely on ICP for faster performance.
- Employ the `pointcloud_to_laserscan` [26] package to convert the 3D point cloud from the Ouster LiDAR into a 2D laser scan slice for Navigation2.

Although RTAB-Map performs mapping and localization in 3D, it generates a 2D occupancy grid for Navigation2. This grid is used for path planning and goal navigation, ensuring that while the internal map is 3D, the navigation occurs in a computationally efficient 2D space.

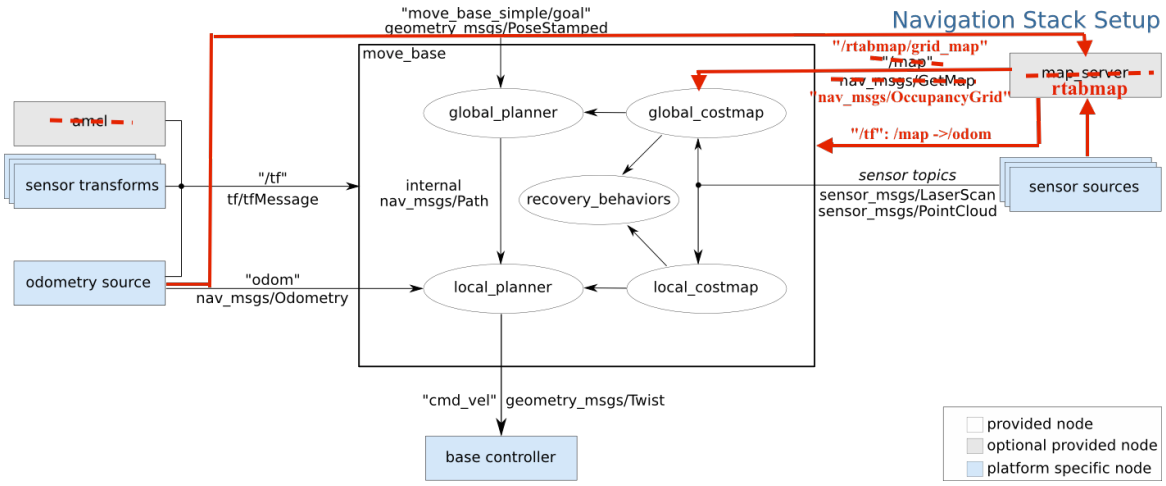


Figure 2.4 Integration of RTAB-Map [18] and Navigation2 [22] for 3D localization and 2D navigation.

2.3.5 Performance Considerations

Running SLAM and Gazebo simultaneously on low-resource machines, such as the Mini-PC mounted on the robot, can be computationally expensive. This may result in the robot struggling to align perfectly with its goals or drifting due to dropped frames. To mitigate these issues:

- Use a more powerful computer to handle the simulation.
- Simplify the simulation environment to reduce computational load.

3 Real World Experiments

3.1 Set-up Overview

During the real-world experiments, the Mini-PC, LiDAR, and Intel RealSense camera were mounted on the Go1 Robot. Figure 3.1 shows the real setup. The LiDAR is connected to the Mini-PC via the `enp2s0f0` Ethernet interface and is powered by a 24V battery. The Mini-PC is powered by a 19V battery and is also connected to the robot via the `enp1s0` Ethernet interface.

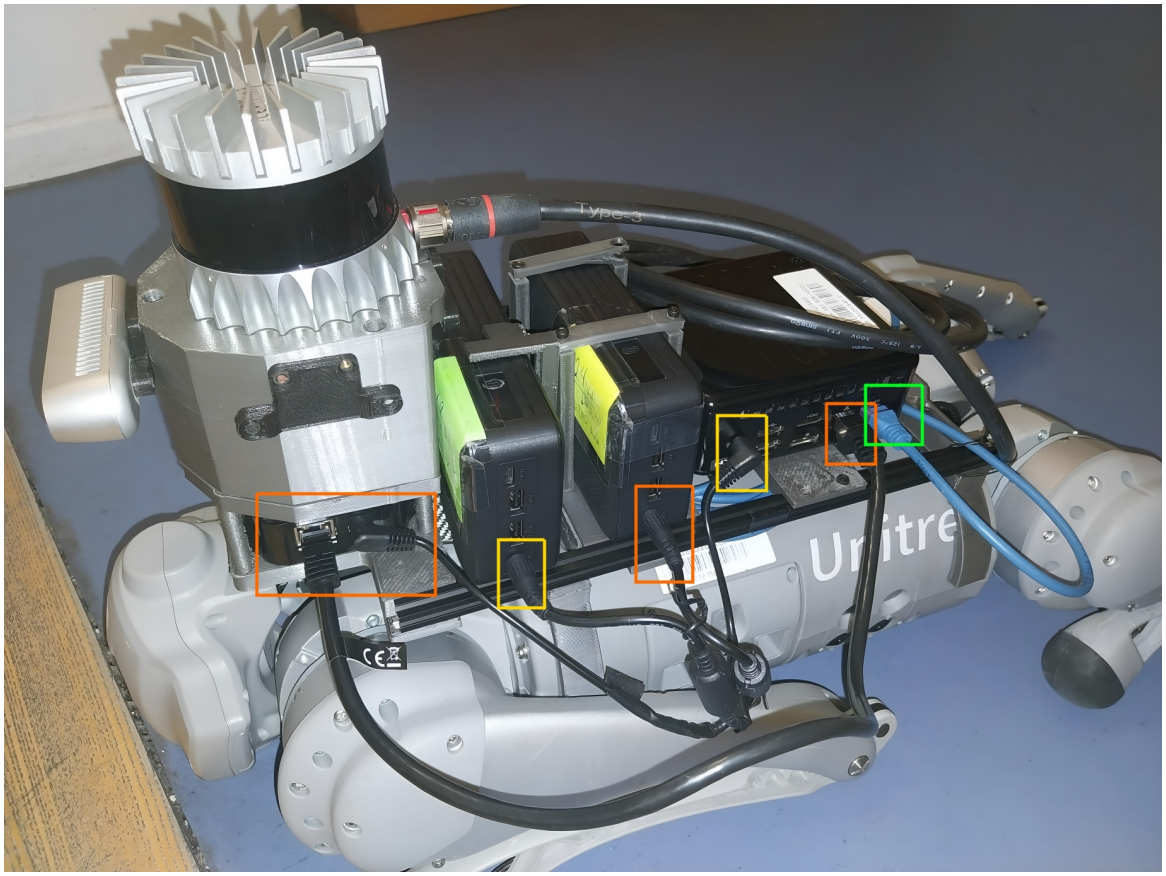


Figure 3.1 The real-world setup of the Go1 Robot. Orange indicates the LiDAR connections, yellow highlights the Mini-PC to battery connections, and the green box shows the Ethernet connection between the Mini-PC and the robot.

3.2 Ouster LiDAR Configuration and Connection

We follow the instructions provided by the Ouster ROS 2 drivers repositories [27, 28]. To set up the Ethernet interface and configure the Ouster LiDAR, we have created a script that automates the network configuration process. The steps involved are as follows:

- **Disconnect the ethernet cable:** Ensure that the Ethernet interface `enp2s0f0` is disconnected before running the script. This prevents any network conflicts during the configuration process.

- **Firewall Configuration:** Disable the firewall to allow uninterrupted communication between the Mini-PC and the LiDAR sensor. For enhanced security, expose only specific ports required for the sensor.
- **Run the initialization script:** Execute the initialization script to configure the network settings.
- **Connect the sensor:** Once you see the prompt message indicating readiness ("Connect the sensor"), connect the Ouster LiDAR sensor to the Ethernet port of the Mini-PC.
- **Script actions in details:**
  - **Flush Existing IP Addresses:** The script flushes any existing IP addresses on the `enp2s0f0` interface to ensure a clean configuration.
  - **Assign Static IP:** Assigns a static IP address (e.g., `10.5.5.1/24`) to the `enp2s0f0` interface.
  - **Bring Up Interface:** Brings up the Ethernet interface and uses `dnsmasq`, a lightweight DHCP and DNS server, to manage the IP address range for the Ouster sensor.

To execute the initialization process, run the following commands:

```
$ cd unitree_go1/real
$ sudo src/scripts/init_ouster_connection.sh
```

3.1 Initialize Ouster Connection

Once the connection is established (as shown in Figure 3.2), you can launch the Ouster LiDAR driver and visualize the data in RViz:

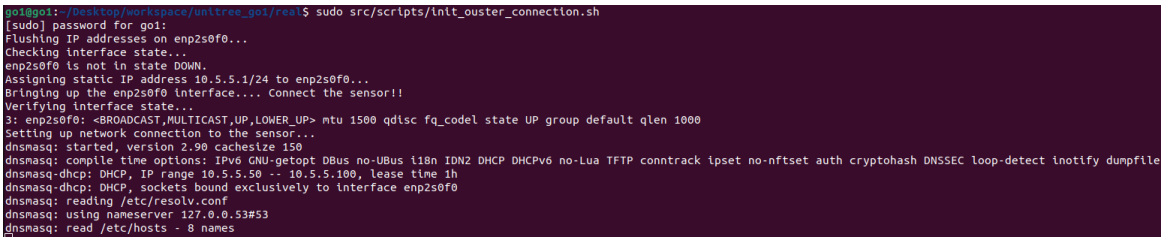


Figure 3.2 Terminal output showing the successful initialization and connection of the Ouster LiDAR sensor.

```
$ cd unitree_go1/real
$ source install/setup.bash && ros2 launch unitree_nav ouster_sensor.
    launch.py viz:=true
```

3.2 Launch Ouster Sensor Node and Visualization

The `ouster_sensor.launch.py` file allows you to override several parameters to customize the LiDAR's operation:

- `lidar_mode`: Specifies the operational mode of the LiDAR. For example, `512x20` sets the LiDAR to run at 20 Hz with a resolution of 512 points per column.
- Additional modes and configurations can be found in the Ouster Modes Documentation.

To support these configurations, edit the `ouster_sensor.launch.py` file accordingly.
To ensure also seamless integration with RTAB-Map [18] (as we will present later) and accurate odometry estimation, we have configured the following parameters:

- `sensor_qos_profile`: This parameter is set to `RELIABLE`, as ICP in RTAB-Map requires this quality of service profile. The default, `BEST_EFFORT`, can lead to errors in ROS communication on the associated topics.

- `timestamp_mode`: This parameter is set to `TIME_FROM_ROS_TIME` to ensure that ROS timestamps are correctly applied. Additionally, `use_system_default_qos:=true` is enabled to synchronize ROS time, preventing potential warnings and ensuring accurate data alignment. Without enabling this option, the `timestamp_mode` setting has no effect.

Once configured, the LiDAR data will be published to the `/ouster/points` topic, which can be utilized by SLAM and navigation pipelines.

Regarding the frames of the Ouster LiDAR, the convention is that the x-axis points outward from the external connector, as illustrated in Figure 3.3. However, this is not the case for key frames like base_link and the frame os_sensor_mount_lidar (the renamed "lidar" frame from our simulation). The transforms must be correct, and in our code, internally, the `os_lidar` and `os_sensor_mount_lidar` frames are set to be identical. To ensure this alignment, we use the following command:

```
$ ros2 run tf2_ros static_transform_publisher 0.0 0.0 -0.036 0.0 0.0 -1.0
   0.0 os_sensor_mount_lidar os_sensor
```

3.3 Set Static Transform for Ouster LiDAR

3.3 Robot Network Configuration and Monitoring

To establish a connection with the robot for real-world experiments and monitor ROS topics across multiple devices, follow the steps below. This section covers configuring the Ethernet connection, accessing the robot via SSH, and connecting additional PCs.

3.3.1 Ethernet Configuration

1. **Set Up Static Ethernet Connection:** On your Linux machine, configure a new static Ethernet connection with the following settings:
   - **IP Address:** `192.168.123.51`
   - **Netmask:** `255.255.255.0`
   - **Connection Method:** Manual
2. **Reconnect Ethernet:**
   - Disconnect the Ethernet cable before applying the new settings.
   - Apply the static IP configuration.
   - Reconnect the Ethernet cable once the settings are applied.
3. **Verify Connection:** Open a terminal and ping the robot to ensure connectivity:

```
    $ ping 192.168.123.161
```

3.4 Ping the Robot

   Successful responses indicate a proper connection.

3.3.2 Remote Desktop Access to the Mini-PC

During experiments, all computing is performed on the Mini-PC. To remotely access and control the Mini-PC, set it up as a Remote Desktop Server using the RDP protocol.

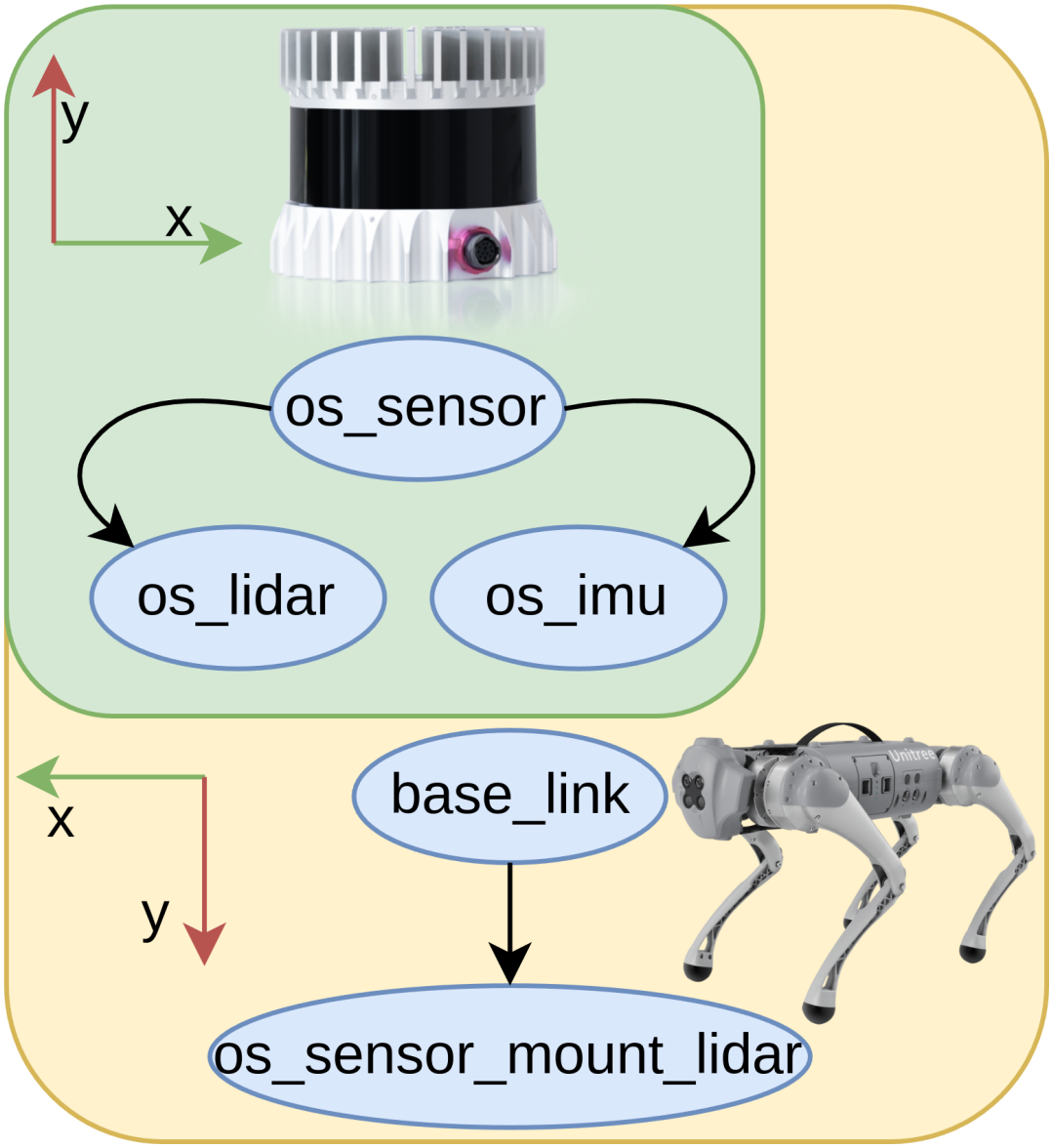


Figure 3.3 Visualizing the transform tree of the Ouster LiDAR sensor. The figure shows the alignment of the `os_sensor_mount_lidar` and `os_sensor` frames.

Installing and Configuring XRDP

Install XRDP and enable the service:

```
$ sudo apt-get install xrdp
$ sudo systemctl enable xrdp
$ sudo systemctl start xrdp
```

3.5 Install and Configure XRDP

Connecting from an External Laptop

On your external laptop, install Remmina, an RDP client:

```
$ sudo apt-get install remmina remmina-plugin-rdp
```

3.6 Install Remmina

Configure the connection as follows:

- **Protocol:** RDP - Remote Desktop Protocol
- **Server:** IP address of the Mini-PC
- **Username:** go1
- **Password:** 1

Network Setup: One option is to configure the laptop to create a hotspot connection, which the Mini-PC will automatically connect to for establishing network communication. **Important:** After the Mini-PC boots, ensure that all Ethernet cables are disconnected to prioritize its connection to the Wi-Fi hotspot.

3.3.3 Connecting Additional PCs for ROS Topic Monitoring

For monitoring ROS topics from another PC without a graphical interface, configure the network and ROS 2 environment as follows:

1. **Set ROS Domain ID:** Ensure all PCs in the ROS network use the same `ROS_DOMAIN_ID`. For example:

```
$ export ROS_DOMAIN_ID=37
```

3.7 Set ROS Domain ID

2. **Configure CycloneDDS:** Create a `cyclonedds.xml` file to specify the network interface for DDS communication:

```
<CycloneDDS>
  <Domain>
    <General>
      <!-- Specify the network interface for communication -->
      <NetworkInterfaceAddress>wlp3s0</NetworkInterfaceAddress>
    </General>
  </Domain>
</CycloneDDS>
```

3.8 CycloneDDS Network Configuration

   Replace `wlp3s0` with the actual network interface name on your PC.

3. **Export CycloneDDS URI:**

```
$ export CYCLONEDDS_URI=file://$HOME/.ros/cyclonedds.xml
```

3.9 Export CycloneDDS URI

4. **Verify ROS Communication:** Once connected to the same network, list the ROS topics to ensure proper communication from your external PC:

```
$ ros2 topic list
```

3.10 List ROS 2 Topics

For isolated experimentation without affecting your host system, we extended a pre-configured Docker file from [29]. You can build and run the Docker container using the following commands:

```
$ cd unitree_go1/Docker
$ ./docker_build.sh && ./docker_run.sh
```

3.11 Build and Run the Docker Container for External Topic Communication

3.4 Navigation Demo

For real-world communication with the robot over UDP, we use the Unitree ROS2 Communication Repository [30], which facilitates real-time communication and control of the robot. For the navigation process, we leverage the Unitree Nav Repository [31].

To launch the Navigation demo, open a new terminal and run the following command:

```
$ cd unitree_go1/real
$ source install/setup.bash && ros2 launch unitree_nav unitree_nav.launch.
    py restart_map:=true use_rtabmapviz:=true localize_only:=false use_rviz
    :=false
```

3.12 Launch the Navigation Demo

The launch file performs the following functions:

- Initializes the RTAB-Map SLAM [18] nodes for simultaneous localization and mapping.
- Loads the robot model and configures the necessary controllers.
- Translates `cmd_vel` commands into high-level movement commands, which are then transmitted via UDP to control the real robot.

If computational resources are limited, you can disable the RTAB-Map visualization panel by setting `use_rtabmapviz:=false` to focus solely on the navigation stack. This improves performance, especially on low-resource systems.

```
$ source install/setup.bash && ros2 launch unitree_nav unitree_nav.launch.
    py restart_map:=true use_rtabmapviz:=false localize_only:=false
    use_rviz:=false
```

3.13 Launch the Navigation Demo Without RTAB-Map Visualization

It is crucial to verify that the required frame transforms are correctly configured before setting any goal positions. The essential transforms include:

- `map` $\rightarrow$ `odom`
- `odom` $\rightarrow$ `base_link`
- `os_sensor_mount_lidar` $\rightarrow$ `os_sensor`

To visualize the frames and confirm their alignment, run the following command:

```
$ ros2 run tf2_tools view_frames
```

3.14 View Frame Transforms

An example of the frame setup is available in the file `frames_nav_demo.pdf` located in the parent directory of the `unitree_go1/` repository.

Moreover, if you wish to use the LiDAR for data collection while controlling the robot via a joystick, run the following command:

```
$ cd unitree_go1/real
$ source install/setup.bash && ros2 launch unitree_nav
    unitree_nav_only_slam.launch.py restart_map:=true use_rtabmapviz:=true
    localize_only:=false use_rviz:=false
```

3.15 Data Collection Demo

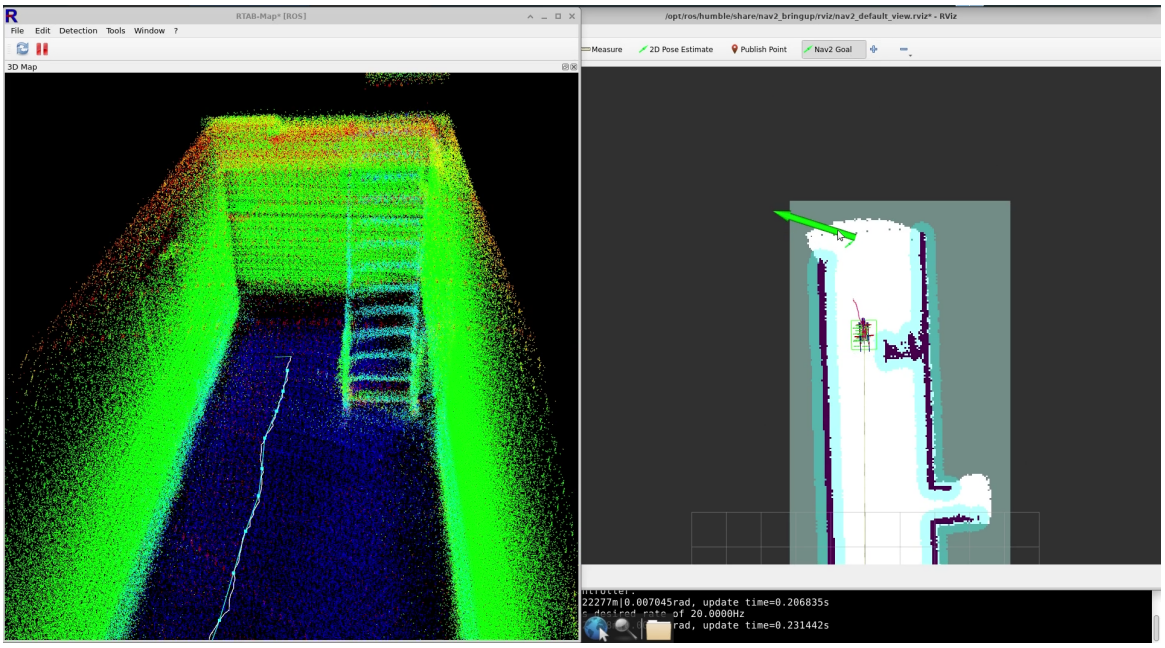


Figure 3.4 Navigation Demo Illustration: Left panel shows RTAB-Map [18] performing SLAM for localization and mapping. Right panel displays RViz with Navigation2 [22], where the robot is navigating toward a defined goal.

The map generated during this process will be saved under `~/.ros/`.

In Figure 3.4, an overview of the running programs during the experiment is shown. The left panel displays the RTAB-Map [18] interface, performing simultaneous localization and mapping (SLAM). The right panel shows RViz with Navigation2 [22], where a navigation goal is set, and the robot autonomously moves toward the target.

Observations During Experiments: During the experiments, several observations were made to ensure successful operation. First, the environment must contain enough distinguishable objects; otherwise, the ICP-based odometry fails, preventing the robot from localizing itself within the map. Additionally, when using Navigation2, it is crucial to extract a 2D slice from the 3D LiDAR point cloud. Issues such as RViz displaying obstacles in front of the robot, even when none exist in reality, are often caused by reflections from the floor or ceiling. Most errors encountered stem from Navigation2 inadvertently reading the entire 3D point cloud. To address this, parameters should be adjusted to utilize only a 2D slice. While RTAB-Map includes techniques for removing floor and ceiling data, Navigation2 directly consumes sensor topic data without applying such filters.

3.5 Inspection Demo

Building on the Navigation Demo, our repository includes an inspection module that enables the Go1 robot to autonomously navigate to predefined points, perform inspections, and update its goals based on recognized text labels. This functionality is supported by the Unitree Inspection Repository [32].

For visual inspections, we integrate a RealSense camera using the Intel RealSense ROS2 Repository [9]. Follow the steps below to set up and launch the RealSense camera.

First, connect the RealSense camera to the Mini-PC and launch its driver with the following command:

```
$ cd unitree_go1/real
$ source install/setup.bash && ros2 launch unitree_nav realsense.launch.py
```

3.16 Launch RealSense Camera

The RealSense camera operates in the default resolution of `640x480@30Hz`.

To align the RealSense camera frames with the robot's URDF, we publish (internally) a static transform between the realsense_camera_link and camera_link frames:

```
$ ros2 run tf2_ros static_transform_publisher 0.0 0.0 0.0 0.0 0.0 1.0 0.0
   realsense_camera_link camera_link
```

3.17 Set Static Transform for RealSense Camera

Ensure also the frame setup is identical to the frames_inspection_demo.pdf located in the parent directory of the unitree_go1/ repository.

Finally, **to launch the Autonomous Inspection demo**, execute:

```
$ cd unitree_go1/real
$ source install/setup.bash && ros2 launch unitree_inspection
   autonomous_inspection.launch.py restart_map:=true use_rtabmapviz:=false
    localize_only:=false use_rviz:=false
```

3.18 Launch Inspection Demo

This command initializes all nodes required for the inspection demo, allowing the Go1 to navigate its map, avoid obstacles, and perform inspections at predefined locations.

Once the inspection demo is active, command the robot to navigate to a specific inspection point (e.g., labeled "000") using the following service call:

```
$ ros2 service call /go_to_inspection_point unitree_inspection_interfaces/
   srv/GoToInspectionPoint "{inspection_point: '000'}"
```

3.19 Command Go1 to an Inspection Point

Behavior: At the specified inspection point, the Go1 performs an inspection by sweeping its head up and down to recognize a text label. Based on the recognized text, it updates its goal to the next inspection point and continues the process until it reaches all the points.

To modify inspection points, edit the environment_params.yaml file located at:
unitree_go1/real/src/unitree_inspection/unitree_inspection/config/

The configuration format is as follows:

```
/**:
  ros__parameters:
    inspection_points:
      text:   [ "000" ]
      x:      [  2.00 ]
      y:      [  0.000 ]
      theta:  [  0.000 ]
```

3.20 Inspection Points Configuration

Each inspection point is defined by a text label and its corresponding 2D pose in the `map` frame.

In Figure 3.5, the inspection demo setup is shown. The left panel illustrates the Navigation2 [22] RViz interface with the robot navigating toward inspection point "000". The right panel shows the RealSense camera's visualization at the inspection point, where the robot successfully detects the text label in the image.

Observations During Experiments:

- Text recognition performance is limited at far distances, requiring close proximity for accurate detection.
- Ensure sufficient lighting and stable mounting of the camera for consistent results.

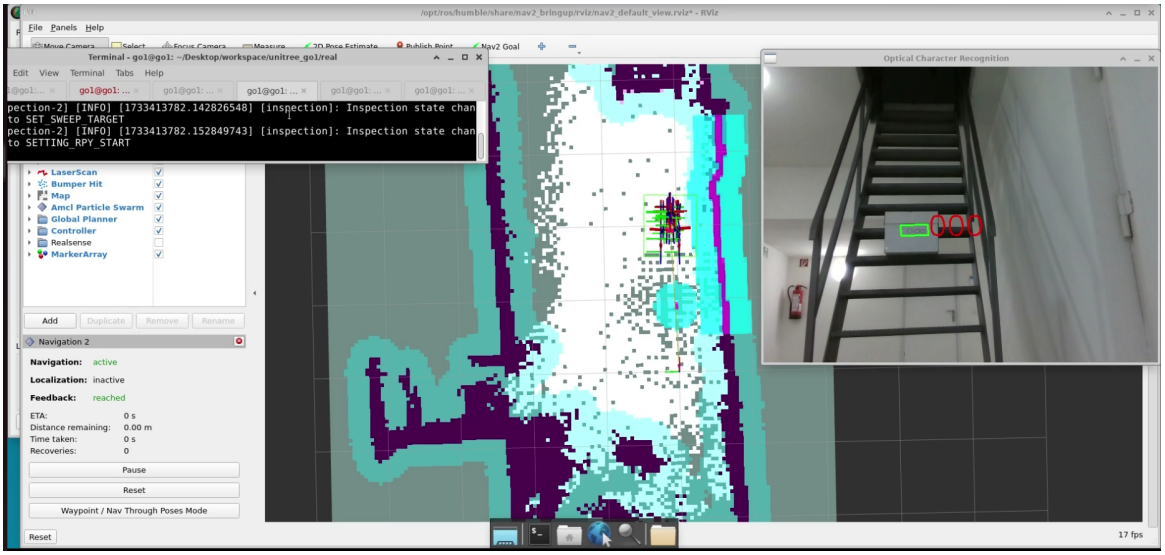


Figure 3.5 Inspection Demo Illustration: Left: RViz interface from Navigation2 [22] navigation to inspection point "000". Right: RealSense camera visualization confirming text recognition at the inspection point.

4 Conclusion and Future Work

4.1 Conclusion

In this work, we successfully implemented and extended the Unitree Go1 robot platform using ROS 2. We developed a complete simulation environment, integrating key components such as a 3D LiDAR and an Intel RealSense depth camera. This setup provided a comprehensive overview of ROS 2 functionalities on the Go1, including robot descriptions, sensor integration, communication protocols, and network configurations for experimental use.

Real-world experiments further demonstrated the capabilities of our system. These included a **navigation demo**, where the robot autonomously navigated to user-defined goals, and an **inspection demo**, which allowed the robot to visit predefined points in a map and perform text-based inspections using the RealSense camera. These experiments validated the system's ability to perform tasks in real environments while highlighting areas for improvement.

4.2 Future Directions

While our current implementation provides a robust foundation for the Go1 in ROS 2, there are several areas for future enhancement:

- **Advanced Controllers for Complex Terrains:** The current control framework is optimized for flat surfaces. For navigating uneven terrain, advanced controllers such as Free Gait [33] can be explored. These controllers are specifically designed for legged robots and provide enhanced stability and adaptability.
- **Optimized Path Planning for Legged Robots:** The ROS 2 Navigation Stack (Nav2) is primarily designed for mobile robots and may result in less smooth paths for quadrupeds. Implementing specialized path planners like the ART Planner [34] could significantly improve the path planning performance for legged robots.
- **SLAM with Dynamic Obstacles:** While RTAB-Map SLAM [18] implementation works well in static environments, it struggles with dynamic obstacles. Addressing this limitation requires integrating algorithms capable of differentiating between static and dynamic objects to prevent unintended map updates.
- **Inspection Accuracy:** Improving the text recognition and object detection accuracy of the inspection module would enhance its reliability. This could involve leveraging foundational models, such as pre-trained vision transformers (e.g., DINO [35] or CLIP [36]), to provide robust and adaptable recognition capabilities. Fine-tuning these models with domain-specific data could further enhance accuracy and adaptability to inspection-specific tasks.
- **Enhanced Metrics and Testing:** Future experiments should incorporate additional metrics to evaluate system performance. Stability tests on diverse terrains and dynamic environments would provide a better understanding of the robot's operational limits.
- **Collaboration and External Device Integration:** The current communication setup supports external laptop integration for monitoring and control, with a typical Wi-Fi range of approximately 10–15 meters. However, to enable the robot to inspect larger areas, such as an entire construction site,

this range is insufficient. Ideally, the robot should operate seamlessly across the entire site without losing connectivity. Achieving this would require either deploying high-gain antennas to extend the communication range or establishing a dedicated wireless network infrastructure throughout the construction site. For example, a mesh network of access points could ensure reliable coverage across large areas, enabling real-time data transfer and collaboration between multiple robots and devices.

- **Hardware Scalability for Processing Demands:** The current Mini-PC setup is sufficient for simple demonstrations and lightweight processing tasks. However, for computationally intensive applications, such as advanced object detection, real-time 3D mapping, or collaborative multi-robot systems, an upgrade to more powerful platforms is recommended. Systems like NVIDIA Jetson AGX Orin or Xavier, equipped with dedicated NVIDIA GPUs, offer significantly higher processing power and are optimized for AI workloads. These platforms enable the use of deep learning models in real-time.

Bibliography

[1] Quadruped Robotics. *GO1 Tutorials*. Accessed: 2024-11-08. 2024. URL: https://www.docs.quadruped.de/projects/go1/html/operation.html.

[2] Intel. *Intel® RealSense™ Product Family D400 Series Datasheet*. Revision 019, October 2024. 2024.

[3] Ouster. *OS1 Hardware User Manual for Rev C OS1 Sensors*. Accessed: 2024-11-08. July 2022. URL: https://data.ouster.io/downloads/hardware-user-manual/hardware-user-manual-revd-os1.pdf.

[4] Anuj Jain. *Unitree Go2 ROS 2 Integration*. Accessed: 2024-11-08. 2024. URL: https://github.com/anujjain-dev/unitree-go2-ros2.

[5] CHAMP Developers. *CHAMP: Open Source Quadruped Controller for ROS 2*. Accessed: 2024-11-08. 2024. URL: https://github.com/chvmp/champ/tree/ros2.

[6] ROS Simulation Team. *gazebo_ros_pkgs: Wrappers, tools and additional APIs for using ROS with Gazebo*. Accessed: 2024-11-08. 2024. URL: https://github.com/ros-simulation/gazebo_ros_pkgs.

[7] ROS Controls Team. *gazebo_ros2_control: Wrappers, tools and additional APIs for using ros2_control with Gazebo Classic*. Accessed: 2024-11-08. 2024. URL: https://github.com/ros-controls/gazebo_ros2_control/tree/master.

[8] Wil Selby. *ouster_example: Sample code for Ouster OS-1 sensor interfacing and visualization with ROS*. Accessed: 2024-11-08. 2019. URL: https://github.com/wilselby/ouster_example/tree/e9f0b74a536a747826fe14ba60db29696a82cb06.

[9] Intel RealSense Team. *realsense_ros: ROS Wrapper for Intel® RealSense™ Cameras*. Accessed: 2024-11-08. 2024. URL: https://github.com/IntelRealSense/realsense-ros/tree/ros2-master.

[10] PAL Robotics. *realsense_gazebo_plugin: Gazebo ROS plugin for Intel RealSense D435 Camera*. Accessed: 2024-11-08. 2024. URL: https://github.com/pal-robotics/realsense_gazebo_plugin/tree/foxy-devel.

[11] ROS Wiki. *Navigation Stack: Robot Setup - Odometry*. https://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom. Accessed: 2024-12-12. 2024.

[12] Aleksandr V. Segal, Dirk Hähnel, and Sebastian Thrun. "Generalized-ICP". In: *Robotics: Science and Systems*. 2009. URL: https://api.semanticscholar.org/CorpusID:231748613.

[13] Think Autonomous. *Point Cloud Registration*. https://www.thinkautonomous.ai/blog/point-cloud-registration/. Accessed: 2024-12-12. 2024.

[14] ROS Wiki. *RTAB-Map Odometry Package*. https://wiki.ros.org/rtabmap_odom. Accessed: 2024-12-12. 2024.

[15] ROS Wiki. *RViz: 3D Visualization Tool for ROS*. https://wiki.ros.org/rviz. Accessed: 2024-12-12. 2024.

[16] *teleop_twist_keyboard*. https://wiki.ros.org/teleop_twist_keyboard. Accessed: 2024-12-12.

[17] Steven Macenski et al. "The Marathon 2: A Navigation System". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[18] Mathieu Labbé and François Michaud. “RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation”. In: *Journal of Field Robotics* 36 (2018), pp. 416–446. URL: https://api.semanticscholar.org/CorpusID:83459364.

[19] Tom Moore and Contributors. *robot_localization - Non-linear state estimation for robots in ROS*. https://github.com/cra-ros-pkg/robot_localization. Accessed: 2024-12-12.

[20] ROS Navigation2 Team. *Nav2 Default Parameters*. https://github.com/ros-navigation/navigation2/blob/galactic/nav2_bringup/bringup/params/nav2_params.yaml. Accessed: 2024-12-12. 2024.

[21] Kaiyu Zheng. “ROS Navigation Tuning Guide”. In: *ArXiv* abs/1706.09068 (2017). URL: https://api.semanticscholar.org/CorpusID:25474950.

[22] Robotics Back-End. *Navigation2 Tutorial*. https://www.youtube.com/watch?v=idQb2pB-h2Q&t=635s. Accessed: 2024-12-12. 2024.

[23] ROBOTIS. *TurtleBot3 GitHub Repository*. https://github.com/ROBOTIS-GIT/turtlebot3. [Online; accessed 12-December-2024]. 2024.

[24] Open Robotics. *ROS on DDS*. https://design.ros2.org/articles/ros_on_dds.html. [Online; accessed 12-December-2024]. 2024.

[25] Open Robotics. *About Different Middleware Vendors*. https://docs.ros.org/en/foxy/Concepts/About-Different-Middleware-Vendors.html. [Online; accessed 12-December-2024]. 2024.

[26] ROS. *pointcloud_to_laserscan Package*. https://wiki.ros.org/pointcloud_to_laserscan. Accessed: 2024-12-12. 2024.

[27] ROS 2 Ouster Drivers Maintainers. *ROS 2 Ouster Drivers*. https://github.com/ros-drivers/ros2_ouster_drivers. [Online; accessed 12-December-2024]. 2024.

[28] Inc. Ouster. *Ouster ROS Drivers*. https://github.com/ouster-lidar/ouster-ros. [Online; accessed 12-December-2024]. 2024.

[29] Shashank Goyal. *ROS 2 Humble GUI Docker Container: A Step-by-Step Guide*. https://shashank-goyal-blogs.medium.com/ros2-humble-gui-docker-container-a-step-by-step-guide-c541b73fe141. Accessed: 2024-12-16. 2024.

[30] Katie Hughes. *Unitree ROS2 Communication Repository*. https://github.com/katie-hughes/unitree_ros2. Accessed: 2024-12-16. 2024.

[31] Nick Morales. *Unitree Navigation Repository*. https://github.com/ngmor/unitree_nav. Accessed: 2024-12-16. 2024.

[32] Nick Morales. *Unitree Inspection Repository*. https://github.com/ngmor/unitree_inspection. Accessed: 2024-12-16. 2024.

[33] Péter Fankhauser et al. “Free Gait – An Architecture for the Versatile Control of Legged Robots”. In: *IEEE-RAS International Conference on Humanoid Robots*. 2016.

[34] Lorenz Wellhausen and Marco Hutter. “ArtPlanner: Robust Legged Robot Navigation in the Field”. In: *Field Robotics*. 2023.

[35] Mathilde Caron et al. “Emerging Properties in Self-Supervised Vision Transformers”. In: *Proceedings of the International Conference on Computer Vision (ICCV)*. 2021.

[36] Alec Radford et al. “Learning Transferable Visual Models From Natural Language Supervision”. In: *International Conference on Machine Learning*. 2021. URL: https://api.semanticscholar.org/CorpusID:231591445.