Mapping

In this section, we will explore the process of mapping using a TurtleBot3 robot in a 3D simulation environment. Mapping is a crucial capability in robotics, allowing robots to create and navigate within an accurate representation of their environment.

For this tutorial, we will:

- Use the **TurtleBot3 robot** for mapping tasks.
- Employ **Gazebo**, a high-fidelity 3D simulation environment, for realistic mapping simulation.

We will be using **ROS 2 Humble** on **Ubuntu 22.04**. Make sure you have the required packages installed before proceeding.

Note: The tutorial follows the guidelines provided in the TurtleBot3 Manual. Ensure you select the correct ROS distro (Humble) on the website for accurate instructions.

Install Requirements (PC Setup)

Follow these steps to set up your environment:

1. **Install Gazebo and TurtleBot3 Packages**

   Gazebo:

```
$ sudo apt install ros-humble-gazebo-*
```

2. **Cartographer** (SLAM method):

```
$ sudo apt install ros-humble-cartographer
$ sudo apt install ros-humble-cartographer-ros
```

3. **Navigation2** (for future navigation tasks):

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

4. **TurtleBot3 Packages**:

```
$ source ~/.bashrc
$ sudo apt install ros-humble-turtlebot3-msgs
$ sudo apt install ros-humble-turtlebot3
```

Configure Environment Variables

1. Add the TurtleBot3 ROS domain ID to your `.bashrc` file:

```
$ echo 'export ROS_DOMAIN_ID=30 #TURTLEBOT3' >> ~/.bashrc
```

2. Resolve Gazebo issues by sourcing its setup script:

```
$ echo 'source /usr/share/gazebo/setup.sh' >> ~/.bashrc
source ~/.bashrc
```

Gazebo Simulation Setup

Install the Simulation Package

1. Clone the **TurtleBot3 Simulation Package** into your workspace:

```
$ cd ~/ros2_ws/src/
$ git clone -b humble-devel
https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
```

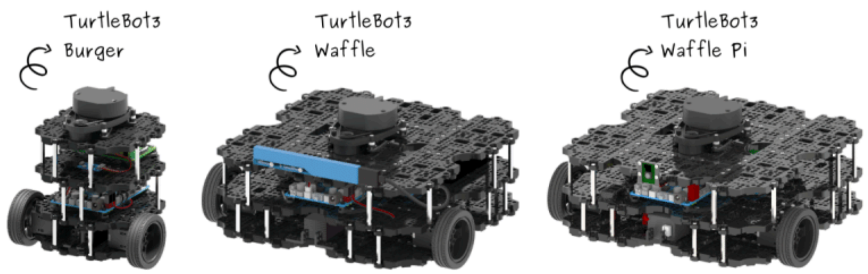
2. Build the workspace:

```
$ cd ~/ros2_ws
$ colcon build --symlink-install
$ source ~/.bashrc
```

Launch the Simulation World

1. Set the TurtleBot3 model to the **burger**:

```
$ echo 'export TURTLEBOT3_MODEL=burger' >> ~/.bashrc
$ source ~/.bashrc
```



2. Launch the Gazebo simulation:

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

Move the Robot

Control the TurtleBot3 robot using the teleoperation node:

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

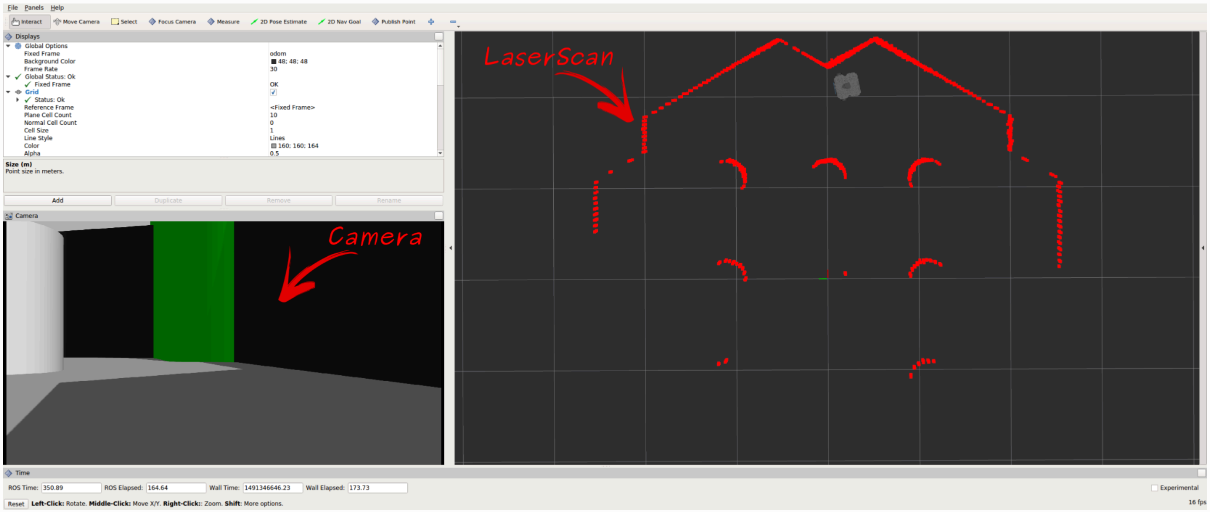
Visualize Simulation Data in RViz2

To visualize sensor data from the robot:

1. Run the RViz2 launch file:

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

2. Observe how the robot perceives its environment using its LiDAR sensor.



SLAM Simulation

What is SLAM?

Simultaneous Localization and Mapping **(SLAM)** is a technique where the robot simultaneously maps its environment while estimating its location. TurtleBot3 uses Cartographer as its SLAM method.

Steps to Perform SLAM:

1. **Launch the Simulation World**:

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

2. **Run the SLAM Node**:

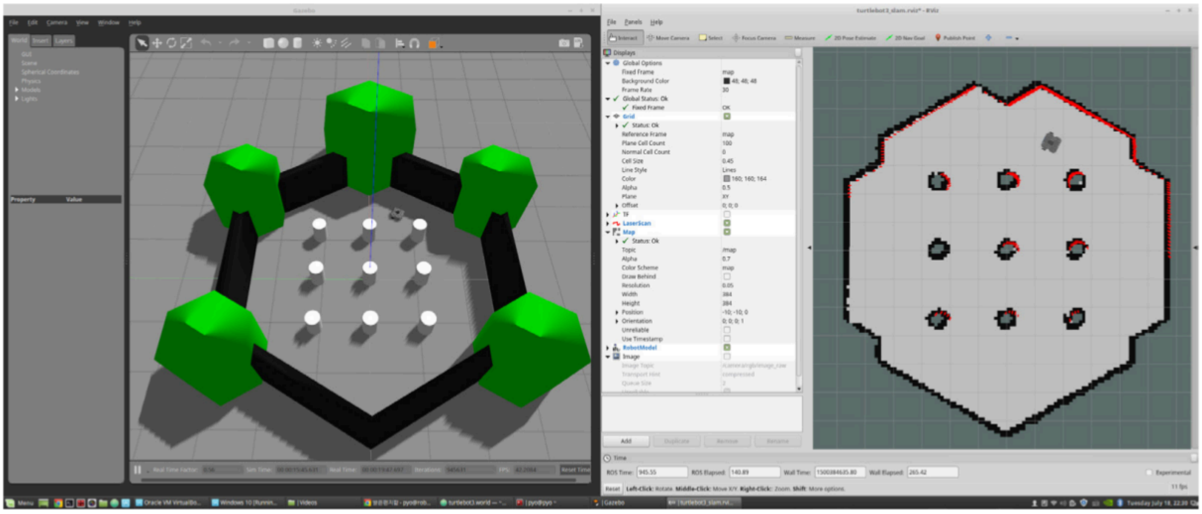
```
$ ros2 launch turtlebot3_cartographer cartographer.launch.py
use_sim_time:=True
```

3. **Control the Robot**: Use the teleoperation node to move the robot and build the map:

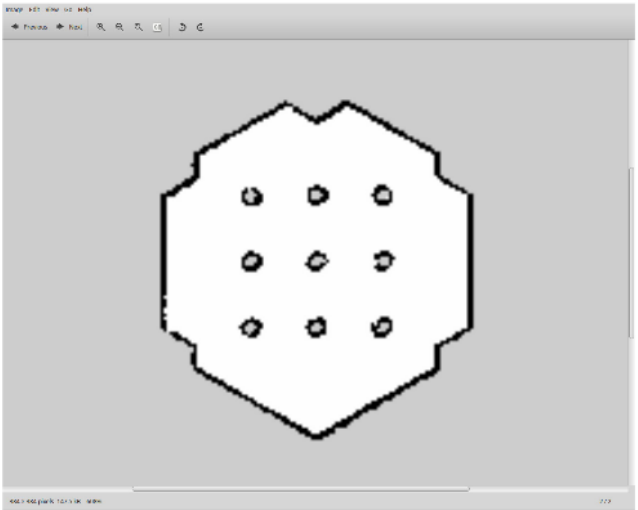
```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

4. **Save the Map**: After mapping the environment, save the generated map:

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



The map will be saved as ***map.pgm*** and ***map.yaml*** in your home directory.



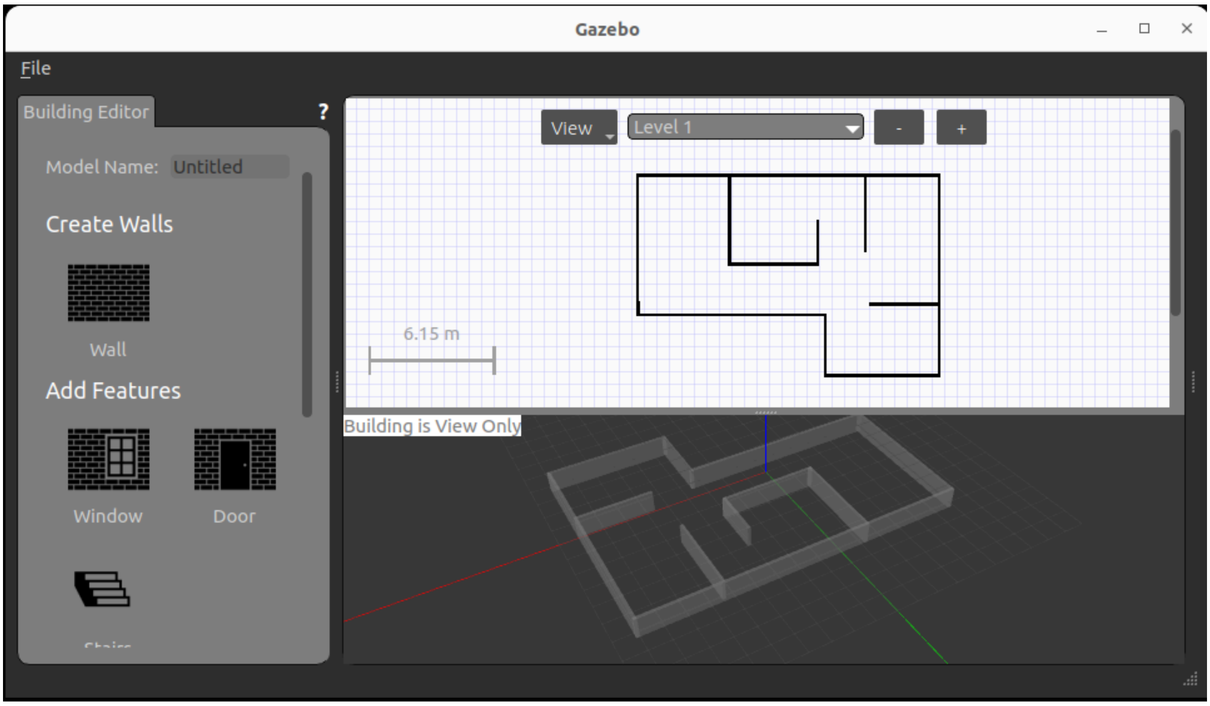
Build a Custom Environment

Create a Custom World in Gazebo

1. Open Gazebo:

```
$ gazebo
```

2. Use the **Building Editor**:
   - Go to the **Edit** tab and select **Building Editor**.
   - Draw walls and design a custom environment (e.g., 15x15 meters).
   - Add internal walls to make the environment moderately complex (avoid a maze structure).



3. Save the environment as a `.world` file:
   - From the **File** menu, click **Save**.
   - Name the file with a `.world` extension, e.g., `my_custom_world.world`.
4. Place the `.world` file in:

```
~/ros2_ws/src/turtlebot3_simulations/turtlebot3_gazebo/worlds
```

Launch the Simulation in Your Custom World

1. Modify the `turtlebot3_world.launch.py` file:
   - Update the `world` variable to your custom world file:

```
world = os.path.join(
    get_package_share_directory('turtlebot3_gazebo'),
    'worlds',
    'my_custom_world.world'
)
```

   - Adjust the robot's initial position if necessary:

```
x_pose = LaunchConfiguration('x_pose', default='0.0')
y_pose = LaunchConfiguration('y_pose', default='0.0')
```

2. Build the workspace and source it:

```
cd ~/ros2_ws
colcon build --symlink-install
source ~/.bashrc
```

3. Simulate with the modified or new launch file:

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

Notes

- If you create a new launch file, replace the file name in the commands with your new launch file name.
- Ensure that all paths and environment variables are correctly configured.

Creating an Autonomous Navigation Node for Mapping

In this tutorial, we will create a TurtlebotMappingNode to autonomously navigate the robot during the mapping process. This node will use LiDAR data to detect obstacles and publish navigation commands to the ***/cmd_vel*** topic, allowing the robot to explore and map the environment without user intervention.

Step 1: Understanding the Required Topics and Messages

1. **Topic for Navigation**:
   - The TurtleBot3 receives navigation commands via the `/cmd_vel` topic.
   - Confirm the topic by running:

     ```
     $ ros2 topic list
     ```

   - Verify the message type:

     ```
     $ ros2 topic info /cmd_vel
     ```

     The message type is `geometry_msgs/msg/Twist` [more info].

2. **Topic for Obstacle Detection**:
   - The TurtleBot3 publishes LiDAR data to the `/scan` topic.
   - Confirm the message structure using:

     ```
     $ ros2 interface show sensor_msgs/msg/LaserScan
     ```

   The `LaserScan` message includes key attributes [more info]:

   - **ranges**: Distance measurements at 360-degree angles.
   - **angle_min / angle_max**: Minimum and maximum angles of the scan.
   - **range_min / range_max**: Minimum and maximum measurable distances.

Step 2: Creating the Mapping Node

1. **Create a New Node**:
    - Add a new file `mapping.py` to your package:

```
$ cd ~/ros2_ws/src/my_robot_controller/my_robot_controller
$ touch mapping.py
$ chmod +x mapping.py
```

2. **Add the Following Code**:

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

class TurtlebotMappingNode(Node):
    def __init__(self):
        super().__init__("mapping_node")
        self.get_logger().info("Mapping Node has started.")

        # Publisher to send movement commands
        self._pose_publisher = self.create_publisher(
            Twist, "/cmd_vel", 10
        )

        # Subscriber to receive LiDAR data
        self._scan_listener = self.create_subscription(
            LaserScan, "/scan", self.robot_controller, 10
        )

    def robot_controller(self, scan: LaserScan):
        cmd = Twist()
        # Define the width of the range for obstacle detection
        a = 2

        # Extract directional distances
        self._front = min(scan.ranges[:a+1] + scan.ranges[-a:])
        self._left = min(scan.ranges[90-a:90+a+1])
        self._right = min(scan.ranges[270-a:270+a+1])
```

```python
        # Navigation logic based on obstacle detection
        if self._front < 1.0:  # Obstacle ahead
            if self._right < self._left:
                cmd.linear.x = 0.05
                cmd.angular.z = 0.5  # Turn left
            else:
                cmd.linear.x = 0.05
                cmd.angular.z = -0.5  # Turn right
        else:
            cmd.linear.x = 0.3
            cmd.angular.z = 0.0  # Move forward

        # Publish the command
        self._pose_publisher.publish(cmd)

def main(args=None):
    rclpy.init(args=args)
    node = TurtlebotMappingNode()
    rclpy.spin(node)
    rclpy.shutdown()
```

IMPORTANT Explanation

This Python script implements a TurtleBot3 mapping node that autonomously navigates the environment by:

1. **Subscribing** to the LiDAR (`/scan`) topic to detect obstacles.
2. **Publishing** velocity commands to the `/cmd_vel` topic to control the robot's motion.

The logic ensures the robot avoids obstacles and explores the environment for mapping purposes.

1. **Import Statements**

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
```

- `rclpy`: ROS 2 client library for Python.
- `Node`: Base class for creating ROS 2 nodes.
- `Twist`: Message type for publishing velocity commands (linear and angular velocities).
- `LaserScan`: Message type for subscribing to LiDAR data.

2. **Node Initialization**

```
class TurtlebotMappingNode(Node):

   def __init__(self):
       super().__init__("mapping_node")
       self.get_logger().info("our controller is started")

       # Publisher for movement commands
       self._pose_publisher = self.create_publisher(
           Twist, "/cmd_vel", 10)

       # Subscriber for LiDAR data
       self._scan_listener = self.create_subscription(
           LaserScan, "/scan", self.robot_controller, 10)
```

➔ **Node Name**:
  - The node is named `"mapping_node"` when initialized.
  - This is important for debugging and identifying the node in the ROS system.

➔ **Publisher**:
  - **Topic**: `/cmd_vel`
  - **Message Type**: `Twist`
  - **Purpose**: Sends velocity commands to control the robot.

➔ **Subscriber**:
  - **Topic**: `/scan`
  - **Message Type**: `LaserScan`
  - **Purpose**: Reads LiDAR data to identify obstacles.
  - **Callback Function**: `robot_controller` processes incoming data.

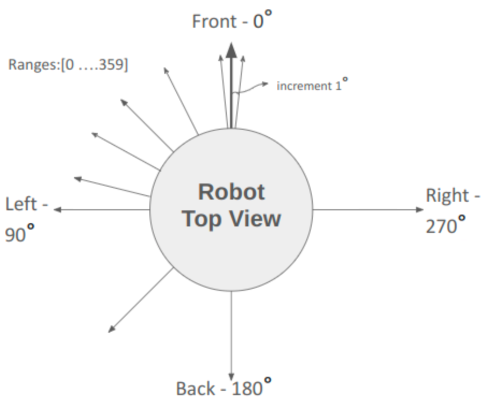
➔ **Logging**:
  - A message logs the initialization of the node: `"our controller is started"`.

3. **Callback Function: `robot_controller`**

```
def robot_controller(self, scan: LaserScan):
        cmd = Twist()
        a = 2  # Number of readings in each direction to consider

        # Extract minimum distances in different directions
        self._front = min(scan.ranges[:a+1] + scan.ranges[-a:])
        self._left = min(scan.ranges[90-a:90+a+1])
        self._back = min(scan.ranges[180-a:180+a+1])
        self._right = min(scan.ranges[270-a:270+a+1])
```

- **Input**:
    - `scan`: A `LaserScan` message containing distance readings around the robot.
- **Define Variables**:
    - `a`: Increases the number of LiDAR readings to consider for averaging (smoothing).
    - Example:
        - **`_front`**: Minimum distance in the forward direction (averages readings near the front).
        - **`_left`**: Minimum distance to the left (angles ~90°).
        - **`_back`**: Minimum distance to the back (angles ~180°).
        - **`_right`**: Minimum distance to the right (angles ~270°).



- **LiDAR Data**:
    - `scan.ranges`: List of distance readings for 360 degrees.
    - Uses slicing to extract relevant angles for each direction.

4. **Obstacle Avoidance Logic**

```
if self._front < 1.0:
           if self._right < self._left:
               cmd.linear.x = 0.05
               cmd.angular.z = 0.5  # Turn left
           else:
               cmd.linear.x = 0.05
               cmd.angular.z = -0.5  # Turn right
       else:
           cmd.linear.x = 0.3
           cmd.angular.z = 0.0  # Move forward
```

- **Obstacle Detection**:
    - If an object is closer than 1 meter (`self._front < 1.0`), the robot must avoid it.
- **Turning Decision**:
    - Compares distances to the right and left (`self._right < self._left`).
    - Turns left or right based on which direction has more room.
- **Default Movement**:
    - If the path is clear (`self._front >= 1.0`), the robot moves forward:
        - `cmd.linear.x = 0.3`: Forward velocity.
        - `cmd.angular.z = 0.0`: No rotation.

5. **Publishing Velocity Commands**

```
self._pose_publisher.publish(cmd)
```

- Publishes the `Twist` message to the `/cmd_vel` topic to control the robot's motion.

DONE

Step 3: Update the Package

1. **Add Dependencies**:

Open the `package.xml` file and add:

```
<depend>geometry_msgs</depend>
<depend>sensor_msgs</depend>
```

2. **Register the Node**:

In the `setup.py` file, add the `mapping` node under `console_scripts`:

```
'console_scripts': [
    "test_node = my_robot_controller.my_first_node:main",
    "draw_circle = my_robot_controller.draw_circle:main",
    "pose_sub = my_robot_controller.pose_subscriber:main",
    "mapping = my_robot_controller.mapping:main",
],
```

3. **Build and Source the Workspace**:

```
$ cd ~/ros2_ws
$ colcon build --symlink-install
$ source ~/.bashrc
```

Step 4: Running the Mapping Process

1. **Launch TurtleBot3 Simulation**:

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

   Replace `turtlebot3_world.launch.py` with your customized launch file if applicable.

2. **Launch SLAM**:

   ```
   ros2 launch turtlebot3_cartographer cartographer.launch.py
   use_sim_time:=True
   ```

3. **Run the Mapping Node**:

   ```
   ros2 run my_robot_controller mapping
   ```

4. **Save the Map**:

   After the environment is fully mapped:

   ```
   ros2 run nav2_map_server map_saver_cli -f ~/map
   ```

Step 5: Creating a Launch File for Mapping

To simplify the process, create a launch file that starts all required nodes:

1. **Create a Launch Folder**:

```
$ mkdir ~/ros2_ws/src/my_robot_controller/launch
$ touch
~/ros2_ws/src/my_robot_controller/launch/start_mapping.launch.py
```

2. **Add the Launch File Code**:

```
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
PythonLaunchDescriptionSource
from launch.substitutions import PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
    return LaunchDescription([
        # Launch TurtleBot3 Simulation
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([
                PathJoinSubstitution([
                    FindPackageShare('turtlebot3_gazebo'),
                    'launch',
                    'turtlebot3_world.launch.py'
                ])
            ]),
        ),
        # Launch SLAM
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([
                PathJoinSubstitution([
                    FindPackageShare('turtlebot3_cartographer'),
                    'launch',
                    'cartographer.launch.py'
                ])
            ]),
            launch_arguments={'use_sim_time': 'True'}.items(),
        ),
        # Launch Autonomous Mapping Node
```

```
        Node(
            package='my_robot_controller',
            executable='mapping',
            name='control'
        )
    ])
```

Explanation

Imports:

```
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
PythonLaunchDescriptionSource
from launch.substitutions import PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare
```

1. **`LaunchDescription`**:
   - Defines a list of actions (nodes and included launch files) to execute.
2. **`Node`**:
   - Used to define individual ROS 2 nodes to be launched.
3. **`IncludeLaunchDescription`**:
   - Allows you to include other launch files.
4. **`PythonLaunchDescriptionSource`**:
   - Specifies the source of a launch file written in Python.
5. **`PathJoinSubstitution`**:
   - Constructs file paths dynamically by joining components.
6. **`FindPackageShare`**:
   - Finds the installed directory of a package to locate files.

Function: `generate_launch_description:`

This function defines and returns the list of actions (launch files and nodes) to execute.

Including the TurtleBot3 Simulation:

```
IncludeLaunchDescription(
    PythonLaunchDescriptionSource([
        PathJoinSubstitution([
            FindPackageShare('turtlebot3_gazebo'),
            'launch',
            'turtlebot3_world.launch.py'
        ])
    ]),
),
```

1. **What it Does**:
   - Includes the `turtlebot3_world.launch.py` file from the `turtlebot3_gazebo` package.
   - Starts the Gazebo simulation environment with the TurtleBot3 robot in the default world.
2. **How It Works**:
   - `FindPackageShare('turtlebot3_gazebo')`: Locates the installed `turtlebot3_gazebo` package.
   - `PathJoinSubstitution([...])`: Constructs the full path to the `turtlebot3_world.launch.py` file inside the `launch` folder.

Including the Cartographer SLAM Node:

```
IncludeLaunchDescription(
    PythonLaunchDescriptionSource([
        PathJoinSubstitution([
            FindPackageShare('turtlebot3_cartographer'),
            'launch',
            'cartographer.launch.py'
        ])
    ]),
    launch_arguments={
        'use_sim_tim': 'True'
    }.items()
),
```

1. **What it Does**:
   - Includes the `cartographer.launch.py` file from the `turtlebot3_cartographer` package.

- Starts the Cartographer SLAM node to generate a map of the environment.

2. **Arguments**:
   - `use_sim_tim='True'`:
     - Enables simulated time (`use_sim_time`), synchronizing SLAM with the Gazebo simulation.
3. **How It Works**:
   - Similar to the Gazebo inclusion, it uses `FindPackageShare` and `PathJoinSubstitution` to locate the `cartographer.launch.py` file.

Launching the Custom Mapping Node

```
Node(
    package='my_robot_controller',
    executable='mapping',
    name='control'
)
```

1. **What it Does**:
   - Launches the custom mapping node (`mapping.py`) from the `my_robot_controller` package.
   - The `mapping` node is responsible for:
     - Subscribing to the LiDAR data on `/scan`.
     - Publishing velocity commands to `/cmd_vel` for autonomous exploration.
2. **Parameters**:
   - **`package`**: The package containing the node (`my_robot_controller`).
   - **`executable`**: The name of the executable to run (`mapping`).
   - **`name`**: The node's name in the ROS graph (`control`).

Assembling the Launch Description:

```
return LaunchDescription([
    IncludeLaunchDescription(...),  # Gazebo simulation
    IncludeLaunchDescription(...),  # Cartographer SLAM
    Node(...)                        # Custom mapping node
])
```

1. Combines all actions (simulation, SLAM, mapping node) into a single launch description.
2. Executes them sequentially when the launch file is run.

DONE

3. **Build and Run**:

   Build and source the workspace:

```
$ colcon build --symlink-install
$ source ~/.bashrc
```

   Run the launch file:

```
$ ros2 launch my_robot_controller start_mapping.launch.py
```

Notes

- If you customized the TurtleBot3 simulation or created a new SLAM launch file, update the paths in the launch file accordingly.
- Refer to the ROS 2 Launch System Documentation for further details on creating launch files.

This tutorial guides you through automating the mapping process and simplifies operations using a launch file.