Introduction to Mobile Robotics

Practical Work 10: Nav2 Controller

Zhi Yan

ENSTA, Institut Polytechnique de Paris

1 Objective

In previous sessions, you used the Nav2 Controller (local planner) with default settings. You likely noticed that the robot sometimes moves hesitantly, gets too close to walls, or oscillates. The goal of this session is to master the **Controller Server**. You will:

1. Visualize the **Local Costmap** and understand how the robot "sees" obstacles.
2. Tune the **Controller** to change the robot's behavior (speed, path following, and prediction horizon).
3. Introduce **dynamic obstacles** and observe avoidance.
4. Observe **Recovery Behaviors** in action.
5. Implement a **Waypoint Follower** to simulate a security patrol loop.

2 Prerequisites

- A working Nav2 installation.
- The `my_map_pkg` you created in Practical Work 9 (containing your maps and params).

3 Experiment A: Visualizing the Local Brain

The Controller does not use the static map but the **Local Costmap**. Let's see it.

1. **Use the default Behavior Tree:** We need to undo the update to our `nav2_params.yaml` copy in Practical Work 9:

   ```
   # COMMENT OUT THE LINE BELOW!
   # default_nav_to_pose_bt_xml: "/home/[USER]/ros2_ws/src/my_map_pkg/
     behavior_trees/spin.xml"
   ```

   Resetting the global planner back to NavFn is not mandatory.

2. **Launch Simulation & Nav2:** Use the multi-terminal setup from Practical Work 9 (i.e., Gazebo + Localization + Navigation + RViz).

   ```
   # T1: Gazebo
   export TURTLEBOT3_MODEL=burger
   ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
   ```

```
# T2: Localization
ros2 launch nav2_bringup localization_launch.py use_sim_time:=True map
    :=/path/to/my_map.yaml

# T3: Navigation (ensure you are using YOUR params file from Practical
     Work 9)
ros2 launch nav2_bringup navigation_launch.py use_sim_time:=True
    params_file:=/path/to/nav2_params.yaml

# T4: RViz
ros2 launch nav2_bringup rviz_launch.py
```

3. **Localize the robot:** Use "2D Pose Estimate" to set the robot's position.
4. **Configure RViz:**
   - Uncheck the "Map" display.
   - Uncheck the "Global Costmap" inside of "Global Planner".
   - Change "Color Scheme" to "Costmap" (if it is not the case) inside of "Local Costmap" inside of "Controller".
5. **Observation:** Drive the robot (send a Goal). Observe the square box around the robot.
   - This is the **Rolling Window**.
   - Notice that as the robot moves, the window moves.
   - Notice that walls in this window come from **live LaserScan data**, not the saved map.

4 Experiment B: Tuning the Controller

The default values for the parameters in `nav2_params.yaml` are actually quite compatible with `turtlebot3_world`. However, for learning purposes, we will still try to change some of them to understand their effects. Find the `controller_server` section in your customized `nav2_params.yaml`. Note that ROS 2 Humble uses the DWB controller by default, while ROS 2 Jazzy uses the MPPI controller by default.

1. **Speed tuning:** The design of `nav2_params.yaml` makes increasing the robot's speed more complex than decreasing it. To focus on understanding our lesson, we will only experience the scenario of slowing the robot down. For **Humble**, set `max_vel_x` from `0.26` to `0.13`. For **Jazzy**, set `vx_max` from `0.5` to `0.22`. You need restart the navigation launch (T3) file for the changes to take effect.
   - **Observe:** The robot's movement speed should have slowed down.
   - **Revert:** Set them back to default values for the next task.
2. **Modify path following (scoring):** Recall you the three cost calculation parameters:
   - `PathDist`: How much should I stick to the global path?
   - `GoalDist`: How much should I aim directly at the goal?
   - `OccDist`: How much should I stay away from the obstacle?

   Task 2.1: For **Humble**, set `GoalDist.scale` from `24.0` to `5.0`. For **Jazzy**, set `cost_weight` inside `GoalCritic` from `5.0` to `1.0`. Relaunch T3 (navigation).

- **Observe:** The robot should now try to travel precisely along the global path.
- **Revert:** Set them back to default values for the next task.

Task 2.2: For **Humble**, set `PathDist.scale` from `32.0` to `6.0`. For **Jazzy**, set `cost_weight` inside `PathFollowCritic` from `5.0` to `1.0`. Relaunch T3 (navigation).

- **Observe:** The robot should now try to "cut corners" and drive straight to the goal, ignoring the specific curve of the global path (unless obstacles block it).
- **Revert:** Set them back to default values for the next task.

Task 2.3: For **Humble**, set `BaseObstacle.scale` from `0.02` to `0.04`. For **Jazzy**, set `cost_weight` inside `CostCritic` from `3.81` to `8.0`. Relaunch T3 (navigation).

- **Observe:** The robot becomes "timid", trying to stay away from obstacles. Visually, if we observe the local costmap, the robot should try to move along the purple areas (low cost), whereas previously it dared to enter the red ones (high cost).
- **Revert:** Set them back to default values for the next task.

3. **The prediction horizon:** This controls how far into the future the planner looks.
Task 3.1: For **Humble**, set `sim_time` from `1.7` to `0.5`. For **Jazzy**, set `time_steps` from `56` to `16`. Relaunch T3 (navigation).

   - **Observe:** The robot becomes "shortsighted". Send it towards a wall at high speed. It will likely react too late and panic stop or crash.

   Task 3.2: For **Humble**, set `sim_time` to `4.0`. For **Jazzy**, set `time_steps` to `86`. Relaunch T3 (navigation).

   - **Observe:** The robot becomes "farsighted". It will start turning meters before the corner. However, this is computationally heavy.
   - **Revert:** Set them back to default values for the next experiment.

5 Experiment C: Dynamic Obstacles

1. Send the robot on a long path.
2. In Gazebo, drop an obstacle (e.g., a box) directly on its path.
3. **Observe:**
   - The Local Costmap updates immediately with the obstacle (magenta/cyan blob).
   - The Global Path (red line) might NOT change immediately.
   - The Local Planner should swerve around it.
4. **Question:** Does the robot stop, go around, or hit it? Why?

6 Experiment D: Recovery Behaviors

What happens if the robot gets confused?

1. Trap the robot in Gazebo (surround it with blocks).
2. Give a goal outside the trap.

3. Watch the terminal (T3) output for "Running spin" and "Running backup".
4. Observe the robot's behavior in Gazebo and Rviz:
    - It should rotate in place (trying to update its costmap with new sensor data).
    - If that fails, it might try to drive backwards.
    - If all fails, it will abort the mission.

7 Experiment E: Waypoint Following (Patrol)

One common task for mobile robots is patrolling (security, inspection). We can easily achieve this leveraging Nav2's "Waypoint" mode in Rviz:

1. On the left side of RViz, look for a button named "Waypoint / Nav Through Poses Mode" and click it.
2. In the top toolbar of RViz, look for "Nav2 Goal" and simply use this button multiple times to set your waypoints.
3. On the left side of RViz, click "Start Waypoint Following" or "Start Nav Through Poses" (blend the motion, i.e., no stop at the waypoints) and the robot should start following the waypoints.

Pretty cool right? Let's go a step further and see how to implement patrolling by writing a program. Create a file `patrol.py`:

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseStamped
from nav2_simple_commander.robot_navigator import BasicNavigator,
    TaskResult

def main():
    rclpy.init()
    navigator = BasicNavigator()

    # Define waypoints.
    # Tip: Use "Publish Point" in RViz to see coordinates.
    w1 = [1.47, 2.28]  # Point A
    w2 = [3.78, 1.09]  # Point B
    w3 = [2.57, -1.26]  # Point C

    goal_poses = []
    for wp in [w1, w2, w3]:
        pose = PoseStamped()
        pose.header.frame_id = 'map'
        pose.header.stamp = navigator.get_clock().now().to_msg()
        pose.pose.position.x = wp[0]
        pose.pose.position.y = wp[1]
        pose.pose.orientation.w = 1.0 # No rotation
        goal_poses.append(pose)

    print("Starting patrol...")
    navigator.followWaypoints(goal_poses)

    while not navigator.isTaskComplete():
        pass
```

```
    result = navigator.getResult()
    if result == TaskResult.SUCCEEDED:
        print('Patrol complete!')
    else:
        print('Patrol failed!')

    rclpy.shutdown()

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

Run it: `python3 patrol.py`, and you will see the robot visiting each point in sequence.

Congratulations! You have now completed the full navigation implementation.