Introduction to Mobile Robotics

Practical Work 6: Maze Solving

Zhi Yan

ENSTA, Institut Polytechnique de Paris

1 Objective

The goal of this session is to use sensor data to implement a classic robotics algorithm: wall-following. You will write a Python (or C++ if you prefer) ROS 2 node that reads LiDAR data (`/scan` topic) and publishes velocity commands (`/cmd_vel` topic) to navigate a robot through a world with obstacles, effectively solving a maze. This is your first step in moving from teleoperation to autonomous behavior.

2 Prerequisites

- A working ROS 2 Humble installation with Gazebo and TurtleBot3 packages (from Practical Work 4).
- Your ROS 2 workspace (`~/ros2_ws`) setup from Practical Work 3.
- Familiarity with creating ROS 2 packages and nodes in Python or C++.

3 The Wall-Following Algorithm

We will implement the "right-hand rule" algorithm. It's a simple yet effective way to solve any maze that is "simply connected" (no loops, or rather, where all walls are connected together). The logic is as follows:

1. Keep your right hand on a wall and just walk forward.
2. If the wall ends (e.g., an intersection), you turn right to find it again.
3. If you walk into a corner, you'll turn left until you can move forward again, all while keeping your hand on the wall.

For our robot, this translates to using its LiDAR to check three key regions: the front, the front-right, and the right. Based on the distances measured in these regions, we will decide whether to turn left, turn right, or go straight.

4 Creating the Package and Node

1. Open a terminal and navigate to your workspace's source directory.

```
cd ~/ros2_ws/src
```

2. Create a new Python package for this project.

```
ros2 pkg create --build-type ament_python maze_solver_pkg
```

3. Create a Python file for our node.

```
cd maze_solver_pkg/maze_solver_pkg
touch wall_follower_node.py
```

5 Implementing the Wall Follower Node

Open `wall_follower_node.py` in a text editor, add the following code, and study it carefully. The comments explain how the logic is implemented.

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
import numpy as np

class WallFollowerNode(Node):
    def __init__(self):
        super().__init__('wall_follower_node')
        # Create a subscriber to the /scan topic
        self.subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.scan_callback,
            10)
        # Create a publisher to the /cmd_vel topic
        self.publisher = self.create_publisher(Twist, '/cmd_vel', 10)

        # This timer will call the control_loop method every 0.1 seconds
        self.timer = self.create_timer(0.1, self.control_loop)

        self.latest_scan = None
        self.get_logger().info('Wall Follower Node has been started.')

    def scan_callback(self, msg):
        # This function is called every time a new scan is received.
        # We just store the latest scan data for the control loop to use.
        self.latest_scan = msg

    def control_loop(self):
        if self.latest_scan is None:
            self.get_logger().info('Waiting for scan data...')
            return

        # The TurtleBot3 LiDAR gives 360 readings.
        # We define regions of interest from the ‘ranges‘ array.
        # 0 is front, 90 is left, 270 is right.

        # Let's define our regions more robustly by taking an average
        # over a small arc of readings to avoid noise.
        # The ranges are counter-clockwise.

        # Front region: average of readings from -15 to +15 degrees
```

```
        front_arc = np.concatenate((self.latest_scan.ranges[0:16], self.
    latest_scan.ranges[345:360]))

        # Right region: average of readings from 260 to 280 degrees
        right_arc = self.latest_scan.ranges[260:281]

        # Ignore 'inf' values which mean no reading
        dist_front = np.mean([r for r in front_arc if r != float('inf')])
        dist_right = np.mean([r for r in right_arc if r != float('inf')])

        self.get_logger().info(f'Dist Front: {dist_front:.2f}m, Dist Right:
     {dist_right:.2f}m')

        # Create a Twist message to publish velocity commands
        twist = Twist()

        # --- WALL FOLLOWING LOGIC ---
        # We want to maintain a distance of about 0.3m from the right wall.
        desired_dist = 0.3

        # Case 1: Obstacle is too close in front. Must turn left to avoid
    collision.
        if dist_front < 0.4:
            self.get_logger().warn('Obstacle ahead! Turning left.')
            twist.linear.x = 0.05  # Move forward slowly
            twist.angular.z = 0.5   # Turn left

        # Case 2: Wall on the right is too far or gone. Turn right to find
    it.
        elif dist_right > desired_dist + 0.15:
            self.get_logger().info('Wall is far. Turning right.')
            twist.linear.x = 0.15  # Move forward
            twist.angular.z = -0.4  # Turn right

        # Case 3: Wall on the right is too close. Turn left to move away.
        elif dist_right < desired_dist - 0.05:
            self.get_logger().info('Wall is close. Turning left.')
            twist.linear.x = 0.15  # Move forward
            twist.angular.z = 0.4   # Turn left

        # Case 4: Perfect distance. Go straight.
        else:
            self.get_logger().info('Following wall. Going straight.')
            twist.linear.x = 0.2  # Full speed ahead!
            twist.angular.z = 0.0   # No turning

        # Publish the command
        self.publisher.publish(twist)


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

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

6 Configuring the Package

1. **Edit `setup.py`**: Open `~/ros2_ws/src/maze_solver_pkg/setup.py` and add the entry point for our node so `ros2 run` can find it. The `'console_scripts'` section should look like this:

```
'console_scripts': [
    'wall_follower = maze_solver_pkg.wall_follower_node:main',
],
```

2. **Edit `package.xml`**: Add dependencies for the message types we are using. Add these lines inside the `<package>` tags in `~/ros2_ws/src/maze_solver_pkg/package.xml`:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>geometry_msgs</exec_depend>
<exec_depend>sensor_msgs</exec_depend>
```

7 Build, Launch, and Test

1. **Build the package:** Navigate to your workspace root and run `colcon build`.

```
cd ~/ros2_ws
colcon build --packages-select maze_solver_pkg
```

2. **Source the workspace:**

```
source install/setup.bash
```

3. **In Terminal 1**, launch a Gazebo world. The TurtleBot3 world is a good testing ground, but you might need to add simple shapes (spheres, cubes) to make it look like a maze. For example, filling the center creates a loop maze. For more inspiration, please refer to: `https://github.com/yzrobot/mrs_testbed/tree/master/MORSE`

```
# Don't forget to set the model
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

   Wait for Gazebo to fully load.

4. **In Terminal 2** (remember to source!), run your wall-following node:

```
source ~/ros2_ws/install/setup.bash
ros2 run maze_solver_pkg wall_follower
```

8 Observe and Analyze

Your TurtleBot3 should now begin to move on its own! It should find the nearest wall to its right and begin to follow it around the perimeter of the room.

- Watch the terminal output from your node. Does it match the robot's behavior in Gazebo?
- What happens when the robot reaches a sharp corner?
- Can your robot solve the maze you designed?
- Can you improve the algorithm? For example, are the turning speeds too fast or too slow?

- What is a major limitation of this simple algorithm?

At the end, please share your maze and improvements with your classmates!

Congratulations! You have just implemented your first autonomous robot navigation algorithm.