ENAE450: Robotics Programming

Final Competition: Team MERRIS

Micah Rao, Isabella Schroeder, Rayan Elias

June 5, 2026

1 Introduction

For the ENAE450 Final Competition, our team built a fully autonomous navigation stack and put it to the test on a physical TurtleBot 4 navigating a box-built maze. The goal is to get the robot from the Start Zone to the Finish Zone, spin 180 degrees, and stop without any human interaction. We competed in two formats: a pre-mapped race where we had a 3-minute teleoperation window to scout and save a map beforehand, and a novel environment race where the robot had to figure things out completely blind. We built upon the TurtleBot 4's onboard 2D LiDAR as the primary sensor, with computation split between the robot's Raspberry Pi 4 and a host PC talking to it over Fast DDS on the RAL lab network. This report walks through how we built it, how it performed, and what we would do differently given another shot at it.

2 Hardware

The platform used for the competition was the TurtleBot 4, a differential-drive mobile robot built around a Roomba Create 3 base. Key hardware components relevant to navigation included:

- **RPLiDAR:** Primary sensor, providing 2D laser scans used for both mapping and navigation.
- **Raspberry Pi 4:** Onboard compute, handling low-level ROS 2 communication and interfacing with the Create 3 base for odometry and velocity commands.
- **Create 3 base:** Provided wheel odometry and executed velocity commands sent over `/cmd_vel`.

The TurtleBot 4 ships with the ROS 2 software that we have been learning about in ENAE450. Higher-level processing was offloaded to a host PC connected to the TurtleBot 4 over the RAL lab network using Fast DDS as the ROS 2 middleware, with the two machines sharing a single ROS 2 graph. No modifications were made to the TurtleBot 4 hardware itself, as per competition rules. A picture of the robot in the lab is shown in Figure 1.

Figure 1: The TurtleBot 4 platform used during the competition in the IDEA Factory RAL laboratory.

2.1 Methods

Our methodology was straightforward for both race formats. For the pre-mapped race, we drove the robot through the maze during the 3-minute teleoperation window, saved the resulting map using `map_saver_cli`, and then relied on Nav2 with AMCL localization to navigate to a goal point at the maze exit. For the novel environment race, the robot used a right-hand wall-following algorithm to trace the maze walls and find its way out. Our fallback plan, if wall-following failed, was to switch to online SLAM to build a map on the fly and navigate out from there.

2.2 Results

The hardware itself was not a major source of issues during our runs. The robot moved consistently, the LiDAR scans were clean enough to use without filtering, and the Create 3 base responded reliably to velocity commands. The one hardware limitation worth noting was battery life. After roughly an hour of testing, several robots began losing charge, which made back-to-back testing sessions difficult to plan around. Some units held their charge better than others, so this was somewhat unpredictable on a given lab day.

3 Software Architecture

Our ROS 2 stack was split across two machines: the TurtleBot 4's Raspberry Pi 4 handled sensor data and low-level control, while the host PC ran the heavier navigation nodes.

The key topics flowing through the system were `/scan` (LiDAR), `/odom` (wheel odometry), `/tf` (coordinate transforms), and `/cmd_vel` (velocity commands to the base). For the pre-mapped race, the stack consisted of three main components running on the host PC: a localization node using AMCL, the Nav2 navigation stack with its built-in path planner, and a goal-publishing node that sent the maze exit as a `NavigateToPose` action. For the novel environment race, we replaced the localization and planning nodes with a custom wall-following node that subscribed to `/scan` and published directly to `/cmd_vel`.

The launch file for the novel-environment race brought up the wall-follower node with a single `Node` declaration. Listing 1 shows the complete file; it was intentionally minimal so that Nav2, SLAM Toolbox, and AMCL could be brought up separately on competition day without needing to restart the custom node.

Listing 1: Launch file for the novel-environment wall-follower node (`run_wall_follower.launch.py`).

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='maze_navigation',
            executable='wall_follower',
            name='wall_follower',
            parameters=[
                {'use_sim_time': True},
            ]
        )
    ])
```

4 Navigation Approach

4.1 Pre-Mapped Race

For the pre-mapped race, the process broke down into three sequential steps. First, during the 3-minute teleoperation window, we drove the robot through the maze using keyboard teleoperation while SLAM Toolbox built a map in the background. Once coverage looked sufficient, we saved the map with `map_saver_cli`. Second, at the start of the official run, we launched AMCL localization against the saved map and brought up the Nav2 stack. Third, we published a single goal pose at the maze exit, and Nav2 planned and executed the path autonomously.

One key tuning step was adjusting the `inflation_radius` (wall thickness) parameter in the costmap. Our initial value caused the planner to treat the narrow maze corridors as

impassable, so the robot would stop instead of proceeding. Reducing this parameter close to zero allowed the planner to find a path through, though it also reduced the safety margin near walls.

The Nav2 goal was dispatched via a `NavigateToPose` action client. Listing 2 shows the relevant portion of the goal-sending node, which constructs a `PoseStamped` target and sends it asynchronously so the node can continue monitoring the map and odometry topics while Nav2 executes the plan.

Listing 2: Goal dispatch using the `NavigateToPose` action client (`nav2_maze_solver.py`).

```
from nav2_msgs.action import NavigateToPose
from geometry_msgs.msg import Point, PoseStamped, Pose
from rclpy.action import ActionClient

# Inside MazeSolverNode.__init__:
self.nav_client = ActionClient(self, NavigateToPose, 'NavigateToPose
    ')

# Inside map_callback -- fires when the map topic updates:
def map_callback(self, msg: OccupancyGrid):
    if self.current_goal is None or self.current_goal.done():
        goal = NavigateToPose.Goal()

        pose_s = PoseStamped()
        pose_s.header.stamp = self.get_clock().now().to_msg()
        pose_s.header.frame_id = 'base_link'
        pose_s.pose = Pose(
            position=Point(x=targ_x, y=targ_y),
            orientation=self.robot_orientation
        )
        goal.pose = pose_s

        self.current_goal = self.nav_client.send_goal_async(goal)
```

4.2 Novel Environment Race

For the novel environment race, we implemented a left-hand wall-following algorithm. The robot subscribes to `/scan` and uses the LiDAR readings on its left side and front to decide how to move: if the left wall is clear it turns left, if it detects an obstacle ahead it turns right, and otherwise it drives forward. This approach is not guaranteed to solve every maze topology, but it works reliably on simply-connected mazes where all walls are connected to the outer boundary.

State Machine

The wall-follower is structured as a finite state machine with four states: `STOPPED`, `DRIVING`, `TURNING`, and `DONE`. The `timer_callback` runs at 20 Hz, checks the current state, computes

a velocity command, and publishes to `/cmd_vel`. The finish-line condition (open space in all directions, detected via a full-scan moving average) transitions the robot into the `DONE` state. Listing 3 shows the full callback.

Listing 3: 20 Hz control loop implementing the wall-follower state machine (`wall_follower.py`).

```
def timer_callback(self):
    if not self.have_odom or not self.scan_started:
        self.scan_started = self.scan_handler.left_dist() != 0.0
        return

    cmd = TwistStamped()
    cmd.header.stamp = self.get_clock().now().to_msg()
    cmd.header.frame_id = 'base_link'

    left  = self.scan_handler.left_avg()
    right = self.scan_handler.right_avg()
    front = self.scan_handler.front.get_average()

    # State transitions
    if self.state != State.TURNING and self.state != State.DONE:
        if left > self.wall_present * 1.5:          # gap on the
           left -> turn left
            self.state = State.TURNING
            self.heading_direction = 1
            self.turn_yaw = self.robot_yaw
        elif front <= self.wall_present:             # wall ahead ->
            turn right
            self.state = State.TURNING
            self.heading_direction = -1
            self.turn_yaw = self.robot_yaw
        else:
            self.state = State.DRIVING

    # Velocity commands per state
    if self.state == State.STOPPED:
        cmd.twist.linear.x  = 0.0
        cmd.twist.angular.z = 0.0

    elif self.state == State.DRIVING:
        linear, turn = self.drive_balanced(right <= self.
           wall_present)
        cmd.twist.linear.x  = linear
        cmd.twist.angular.z = turn

    elif self.state == State.TURNING:
        turn_dist = abs(wrap_to_pi(self.robot_yaw - self.turn_yaw))
        if front > self.wall_present * 2 and turn_dist > math.pi /
```

```
            3:
40              cmd.twist.linear.x  = self.max_forward_speed / 2
41              cmd.twist.angular.z = 0.0
42              if left <= self.wall_present:
43                  self.error_sum = 0.0
44                  self.state = State.DRIVING
45          else:
46              cmd.twist.linear.x  = 0.0
47              cmd.twist.angular.z = (self.heading_direction
48                                     * self.max_angular_speed / 2)
49
50      elif self.state == State.DONE:
51          cmd.twist.linear.x  = 0.0
52          cmd.twist.angular.z = 0.0
53
54      # Finish condition: total average scan distance exceeds
            threshold
55      if self.scan_handler.total.get_average() >= 2.5:
56          self.turn_yaw = self.robot_yaw + math.pi
57          self.state = State.DONE
58
59      self.cmd_pub.publish(cmd)
```

LiDAR Sector Parsing

Raw `/scan` messages contain a flat array of range readings indexed by angle. Rather than processing every ray at control-loop frequency, we partitioned the scan into named sectors (front, left, right, back) using angular thresholds computed from the message's `angle_min` and `angle_increment` fields. Each sector fed its readings into a `SensorMA` sliding-window moving-average filter (Listing 4) to suppress single-ray noise. Listing 5 shows the sector-routing logic inside `ScanHandler`.

Listing 4: Sliding-window moving-average filter used to smooth per-sector LiDAR readings (`moving_average.py`).

```
1  class SensorMA:
2      def __init__(self, sample_size: int):
3          self.sample_size   = sample_size
4          self.running_sum   = 0.0
5          self.samples_array = [0.0] * self.sample_size
6          self.oldest_value  = 0
7
8      def add_reading(self, reading: float):
9          self.running_sum += reading
10         self.running_sum -= self.samples_array[self.oldest_value]
11         self.samples_array[self.oldest_value] = reading
12         self.oldest_value = (self.oldest_value + 1) % self.
               sample_size
```

```
13
14      def get_average(self) -> float:
15          return self.running_sum / self.sample_size
```

Listing 5: LiDAR scan callback routing rays to named sectors by angle (`scan_handler.py`).

```
1  def scan_callback(self, msg: LaserScan):
2      for i, d in enumerate(msg.ranges):
3          # Reject invalid readings
4          if math.isinf(d) or math.isnan(d) or d <= 0.0:
5              continue
6          if d < msg.range_min or d > msg.range_max:
7              continue
8
9          theta = wrap_to_pi(msg.angle_min + i * msg.angle_increment)
10
11         # Front sector  (around -pi/2 in ROS body frame)
12         if (-math.pi/2 - self.width/2) <= theta <= (-math.pi/2 +
               self.width/2):
13             self.front.add_reading(d)
14
15         # Right sectors (near +/-pi)
16         if theta <= -math.pi + self.width / 2:
17             self.right_1.add_reading(d)
18         if theta >= math.pi - self.width / 2:
19             self.right_2.add_reading(d)
20
21         # Back sector   (around +pi/2)
22         if (math.pi/2 - self.width/2) <= theta <= (math.pi/2 + self.
               width/2):
23             self.back.add_reading(d)
24
25         # Left sectors  (near 0)
26         if -self.width/2 <= theta <= 0:
27             self.left_1.add_reading(d)
28         if 0 <= theta <= self.width/2:
29             self.left_2.add_reading(d)
30
31         self.total.add_reading(d)
```

5 Results

5.1 Race Performance

Due to time constraints and shared access to the TurtleBots in the lab, Professor Conroy recommended we prioritize one race format over completing both. As a result, we focused

our official attempts on the pre-mapped race and were not able to record a timed novel environment run.

Table 1: Official race times and penalty summary.

Race Type	Run	Raw Time (s)	Collisions	Interventions	Completed Maze
Pre-Mapped	1	10	0	1	Unsuccessful (1/4 completed)
	2	95	0	1	Unsuccessful (stopped at exit)
Novel Env.	0	57*	0	1	Unsuccessful
	1	42	1	0	"Successful" (no end 180)
	2	50*	0	0	Unsuccessful
	3	46	1	0	Unsuccessful

* Time taken from rosbag, not necessarily indicative of actual runtime

5.2 Top-Down Run Plot

The top-down trajectory shown in Figure 2 was generated by extracting `/odom` position data from the MCAP ROS 2 bag recorded during our best official run. The x and y odometry fields were plotted directly, giving a top-down view of the robot's path through the maze relative to its starting position.

Listing 6 shows the bag-extraction script that reads the MCAP recording and writes `/odom` and `/cmd_vel` messages to CSV files. Listing 7 shows the plotting script that reads the resulting `odom.csv` and renders the top-down trajectory.

Listing 6: Extracting `/odom` and `/cmd_vel` from the MCAP bag to CSV (`export_odom_cmdvel_signals.py`).

```
from rosbags.highlevel import AnyReader
import csv
from pathlib import Path

with open('odom.csv', 'w', newline='') as odom_file, \
     open('cmd_vel.csv', 'w', newline='') as cmd_file:

    odom_writer = csv.writer(odom_file)
    cmd_writer  = csv.writer(cmd_file)

    odom_writer.writerow([
        'timestamp_ns', 'frame_id', 'child_frame_id',
        'pos_x', 'pos_y', 'pos_z',
        'ori_x', 'ori_y', 'ori_z', 'ori_w',
        'lin_x', 'lin_y', 'lin_z',
        'ang_x', 'ang_y', 'ang_z',
    ])
    cmd_writer.writerow([
        'timestamp_ns', 'frame_id',
```

```
        'linear_x', 'linear_y', 'linear_z',
        'angular_x', 'angular_y', 'angular_z',
    ])

    with AnyReader([bag_dir]) as reader:
        for connection, timestamp, rawdata in reader.messages():
            topic = connection.topic
            if topic not in (f'/{namespace}/odom', f'/{namespace}/
               cmd_vel'):
                continue
            msg = reader.deserialize(rawdata, connection.msgtype)

            if topic == f'/{namespace}/odom':
                pose  = msg.pose.pose
                twist = msg.twist.twist
                odom_writer.writerow([
                    timestamp, msg.header.frame_id, msg.
                       child_frame_id,
                    pose.position.x, pose.position.y, pose.position.
                       z,
                    pose.orientation.x, pose.orientation.y,
                    pose.orientation.z, pose.orientation.w,
                    twist.linear.x,  twist.linear.y,  twist.linear.z
                       ,
                    twist.angular.x, twist.angular.y, twist.angular.
                       z,
                ])
            elif topic == f'/{namespace}/cmd_vel':
                twist = msg.twist
                cmd_writer.writerow([
                    timestamp, msg.header.frame_id,
                    twist.linear.x,  twist.linear.y,  twist.linear.z
                       ,
                    twist.angular.x, twist.angular.y, twist.angular.
                       z,
                ])
```

Listing 7: Top-down path plot from exported odometry CSV (`plot_navigation.py`).

```
import csv, math
import matplotlib.pyplot as plt

t_odom, position = [], []

with open('./odom.csv', newline='') as f:
    reader = csv.DictReader(f)
    for row in reader:
        t_odom.append(int(row['timestamp_ns']) * 1e-9)
```

```
        position.append((float(row['pos_x']), float(row['pos_y'])))

t0      = t_odom[0]
t_odom = [t - t0 for t in t_odom]

xs = [x for x, _ in position]
ys = [y for _, y in position]

plt.figure()
plt.title('Path')
plt.xlabel('X position (m)')
plt.ylabel('Y position (m)')

for i in range(len(t_odom)):
    plt.plot(xs[i], ys[i], '-o', color='blue', markersize=1,
             scalex=False, scaley=False)

plt.xlim(-5, 5)
plt.ylim(5, 15)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

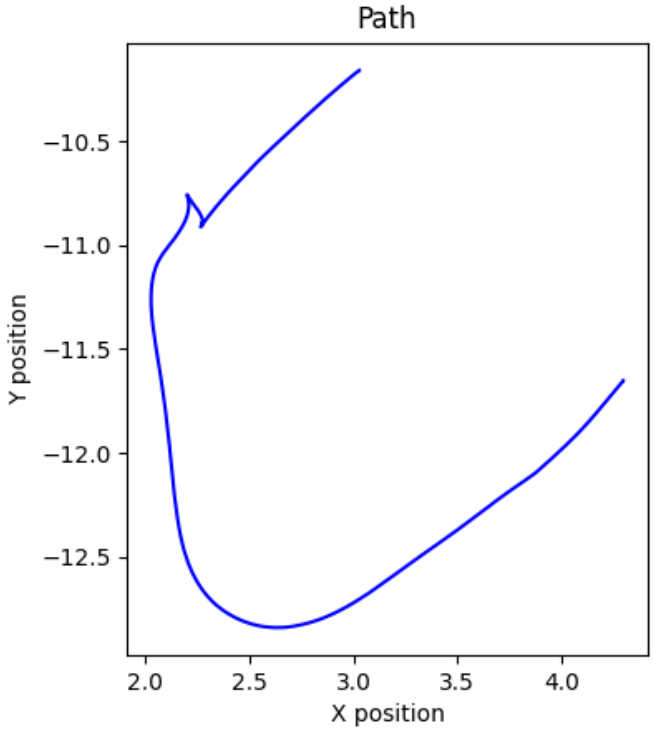

Figure 2: Top-down trajectory of the TurtleBot 4 during the best official run.

5.3 Exploration Run Analysis

When working on our exploration runs, 3 out of 4 runs resulted in the turtlebot being stuck in a loop between the 3 highlighted corners in figure 3.

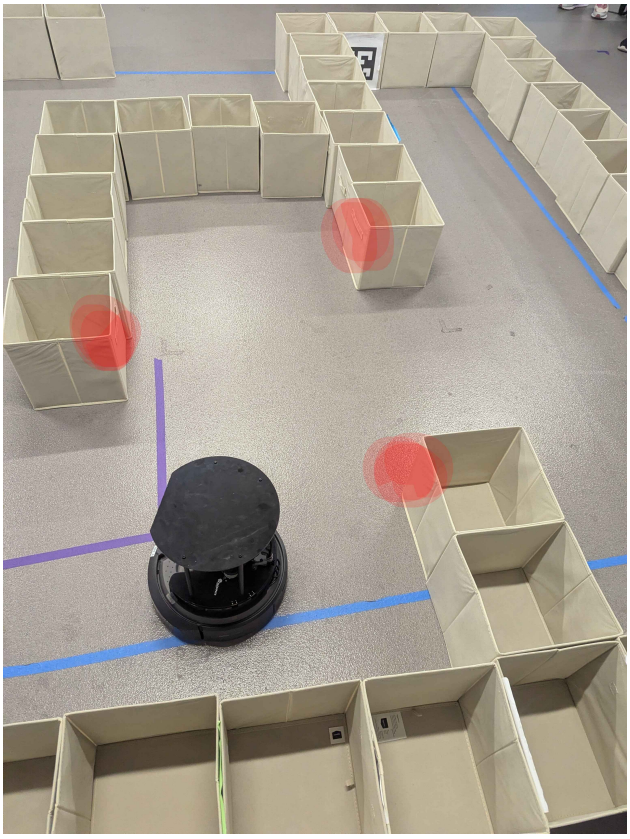

Figure 3: Area where turtlebot began looping

Being between all 3 corners was an edge case in our logic that consistently caused the turtlebot to turn around and go back the way it came. Figures 4, 5, and 6 show the three exploration runs where this "looping" occurred.

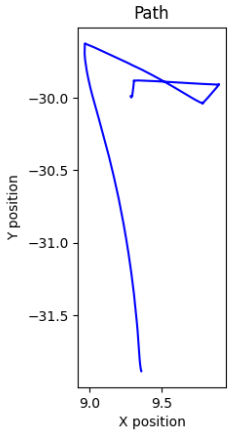

Figure 4: Exploration 0

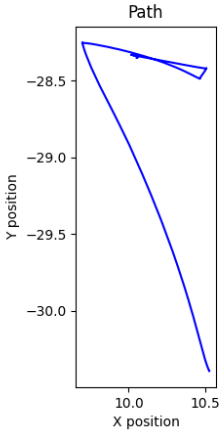

Figure 5: Exploration 2

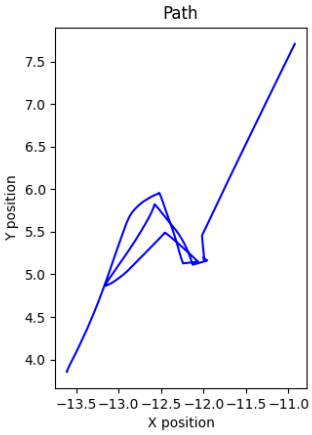

Figure 6: Exploration 3

For figure 6, the odometry position shoots off at the end because we forgot to end the ros bag directly after the run. Our best guess as to the reasoning for the looping is as follows:

1. The turtlebot enters the triangle of corners from the right hand side of figure 3.
2. As the turtlebot passes the threshold, it detects that there is no wall on the left and then turns left.
3. The turtlebot crawls forward until it once again can detect a wall on the left.
4. The turtlebot detects the left wall and starts moving forward, but since every turn is not a perfect 90°, it is angled slightly away from the wall
5. Since the turtlebot is angled away from the wall, once it moves forward it suddenly thinks there is no wall next to it and begins turning left again
6. The left turn logic requires there to be empty space in front of the turtlebot, causing it to turn $\approx 180°$ and go back the way it came.

Finally, figure 7 shows our best exploration run, which made it to the end of the maze but bumped into the side of the exit and failed to turn 180° after exiting.

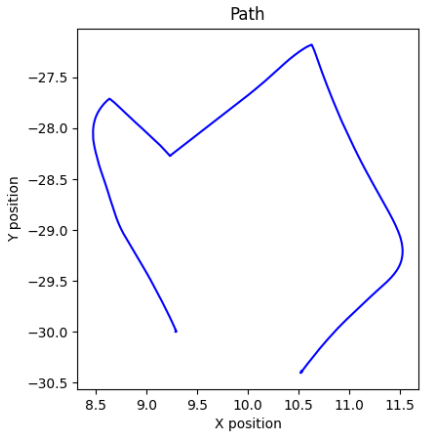

Figure 7: Best exploration run

6 Retrospective

6.1 What Worked

One thing that genuinely helped during development was building a small LiDAR diagnostic tool early on. Drawing on what we learned about subscribers, listeners, and nodes earlier in ENAE450, we set up a node that printed live LiDAR readings when the robot was next to a wall. This made it easy to figure out which direction the robot considered "front" and how far it thought it was from an obstacle, information that was critical when tuning the wall-following logic. Listing 8 shows the diagnostic node's timer callback, which printed the computed right-side distance at 20 Hz so we could verify sector assignments interactively.

Listing 8: LiDAR diagnostic node used to verify sector assignments during tuning (`scan_test.py`).

```python
class ScanTestNode(Node):
    def __init__(self):
        super().__init__('scan_test_node')
        self.scan_sub = self.create_subscription(
            LaserScan, 'scan', self.scan_callback,
               qos_profile_sensor_data
        )
        sample_size = 30
        self.front_MA_1 = SensorMA(sample_size)
        self.front_MA_2 = SensorMA(sample_size)
        self.right_MA_1 = SensorMA(sample_size)
        self.right_MA_2 = SensorMA(sample_size)
        timer_period = 1.0 / 20.0
        self.timer = self.create_timer(timer_period, self.
           timer_callback)

    def get_right_distance(self) -> float:
        a = self.right_MA_1.get_average()
        b = self.right_MA_2.get_average()
        theta = math.pi / 16
        alpha = math.atan2((a * math.cos(theta) - b), (a * math.sin(
           theta)))
        return b * math.cos(alpha)

    def timer_callback(self):
        """Print right-side distance at 20 Hz for live tuning."""
        print(f"Right: {self.get_right_distance():.3f}, "
              f"{self.right_MA_1.get_average():.3f}, "
              f"{self.right_MA_2.get_average():.3f}")
```

On the navigation side, Nav2 consistently generated and visualized a planned path to the goal, even when that path turned out to be wrong. Having the visual feedback in RViz made it much easier to diagnose what the planner was thinking and why it was choosing a particular route. For the wall-following race, the robot handled turning angles and basic obstacle avoidance correctly throughout testing, which gave us confidence in the core algorithm.

6.2 What Did Not Work

The most frustrating issue during the pre-mapped race was a recurring action server error that prevented the robot from reading the navigation goal. We tried adjusting the startup order of localization and Nav2, but the error persisted across multiple attempts. The only reliable fix was a full WSL restart, which worked most of the time but cost us valuable time during the competition. The costmap inflation parameter caused problems that we did not fully resolve in time. Reducing the wall thickness value to near zero fixed the issue in our

test environment, but the same problem resurfaced in the real maze, the robot stopped just before the exit because it still believed there was an obstacle blocking the way. In hindsight, we needed more time to tune the costmap against the actual competition maze geometry rather than our test setup.

6.3 What We Would Change

Given more time, the first thing we would fix is the costmap configuration. The inflation radius needs to be tuned against the real maze, not just the test environment, and ideally we would set up a parameter file that can be adjusted quickly on competition day without editing source files. Beyond that, we would explore a more robust maze-solving algorithm. A junction-tracking approach, where the robot explicitly identifies and records decision points, would handle more complex topologies than pure right-hand wall following. We also would have liked to consolidate our startup process into a single launch file that brings up localization, Nav2, and the goal publisher in the correct order automatically, eliminating the manual sequencing that caused issues during the competition.

7 Team Contribution Summary

Table 2: Team contribution summary.

Team Member	Primary Contributions
Micah Rao	Algorithm design and implementation
Isabella Schroeder	Operations, design logic, and testing
Rayan Elias	Documentation and facilitation

8 Conclusion

Our team made it through most of the pre-mapped maze across two official runs, but ultimately fell short of a clean finish due to costmap tuning issues that stopped the robot at the exit. The experience highlighted something that is easy to underestimate: getting individual ROS 2 nodes to work in isolation is very different from getting them to work together reliably under competition conditions. Node startup order, parameter values, and network timing all interact in ways that are hard to predict without testing in the actual environment. If we were to do this again, we would spend more time on integration testing in the real space and less time on component-level tuning in simulation. The competition was a good reminder that robotics is, above all, a systems problem.

Works Cited (APA):

TurtleBot 4. Clearpath Robotics. (2025, November 18). https://clearpathrobotics.com/turtlebot-4/

Conroy, J. (n.d.). *ENAE450_S26*. GitLab Enterprise Edition. https://code.umd.edu/conroyj/enae450_s26

Lau, K. (2025, April 6). *Learning ROS2 with a simple wall-following robot.* Medium. ://medium.com/@kinran_lau/learning-ros2-with-a-simple-wall-following-robot-55537d6ce084

Che, X., Zhang, Z., Sun, Y., Li, Y., Li, C. (2022). A wall-following navigation method for autonomous driving based on lidar in tunnel scenes. In *Proceedings of the 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)* (pp. 594–598). IEEE. https://doi.org/10.1109/CSCWD54268.2022.9776288