TurtleBot3 Virtual Obstacle Mapping & Localization

From Simulation to Real-Time Autonomous Navigation

Muhammad Ureed Hussain & Ghaith Chamaa
Under Supervision of Prof. Joaquin Rodriguez & Raphaël Duverne

Université Bourgogne Europe

December 15, 2025

Overview

1. Project Objectives
2. Part 1: Perception (Lane Detection)
3. Part 2: Mapping & Localization
4. Navigation
5. Conclusion

Introduction & Objectives

This project develops a complete autonomous driving pipeline for the TurtleBot3, transitioning from a simulated Gazebo environment to real-world hardware.

Two Main Components

1. **Part 1: Lane Detection & Following**:
   - Camera calibration (Intrinsic/Extrinsic).
   - Accurate lane segmentation (Yellow/White lines).
2. **Part 2: Mapping & Localization**:
   - Converting lane perception into a **Virtual Occupancy Grid**.
   - Using SLAM for localization and Nav2 for path planning inside the virtual corridor.

Reference: The pipeline strictly follows and extends the official TurtleBot3 Autonomous Driving E-Manual.

Part 1: Camera Intrinsic Calibration

Methodology:

1. Used the standard ROS 2 camera calibration package with a checkerboard pattern.
2. Executed the `intrinsic_camera_calibration.launch.py` (from the Autonomous Driving E-Manual).

Output:

- Obtained Compressed, Raw, and Rectified image topics necessary for accurate processing.

Figure 1: Calibrating with Checkerboard Pattern.

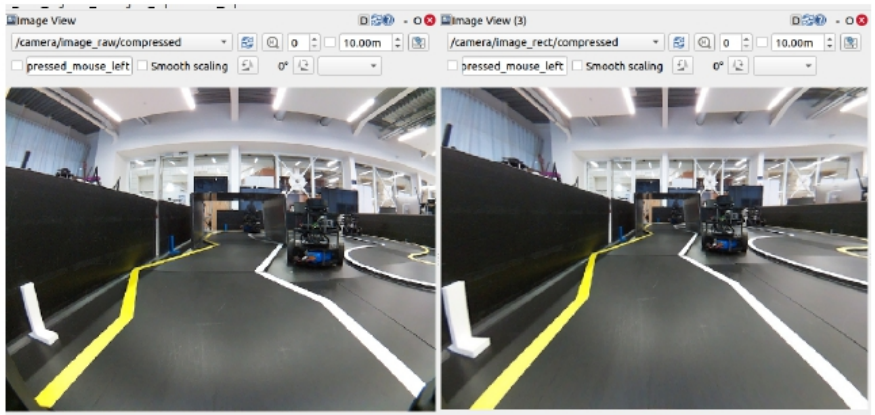


Figure 2: Comparison: Raw vs. Rectified Image.

Part 1: Extrinsic Calibration

Objective: Transform the camera perspective to a "Bird's Eye View" (Ground Projection).

The Challenge

In the standard package, the projected red trapezoid did not align perfectly with the ground plane in our setup.

Solution:

- Added X-offsets and Y-offsets to `image_projection.py`.
- Tuned parameters via `rqt` dynamic reconfigure until the trapezoid matched the lane width.

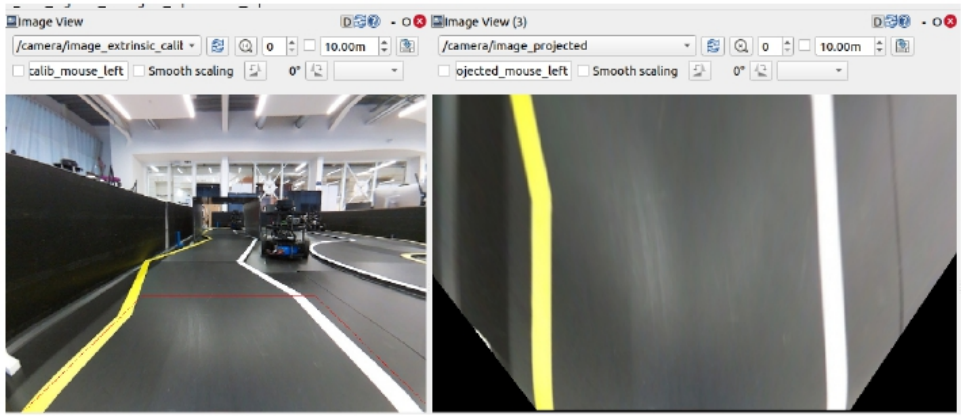


Figure 3: Tuning the projection trapezoid.

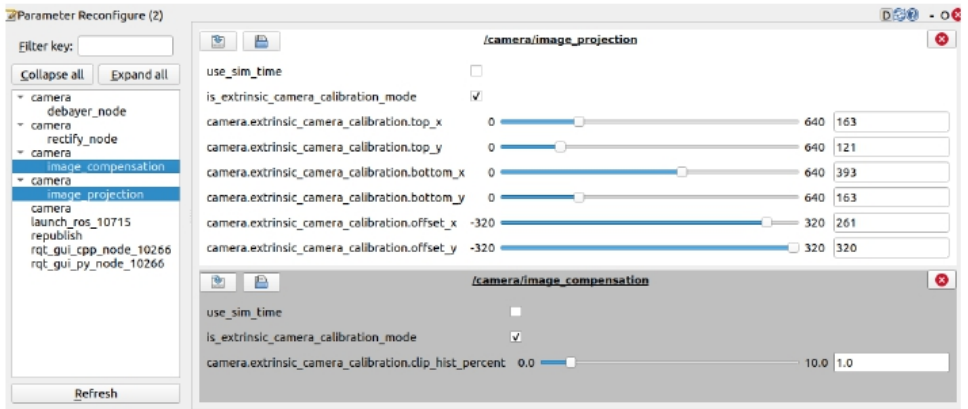


Figure 4: Adjusting the Red Trapezoid for Projection.

Part 1: Lane Detection Calibration

To segment the lanes reliably, we convert images to HSV color space and apply thresholding.

- **Yellow Line**: Represents the Left boundary.
- **White Line**: Represents the Right boundary.
- **Calibration**: Sliders are adjusted in real-time to isolate the lane markers from lighting noise.

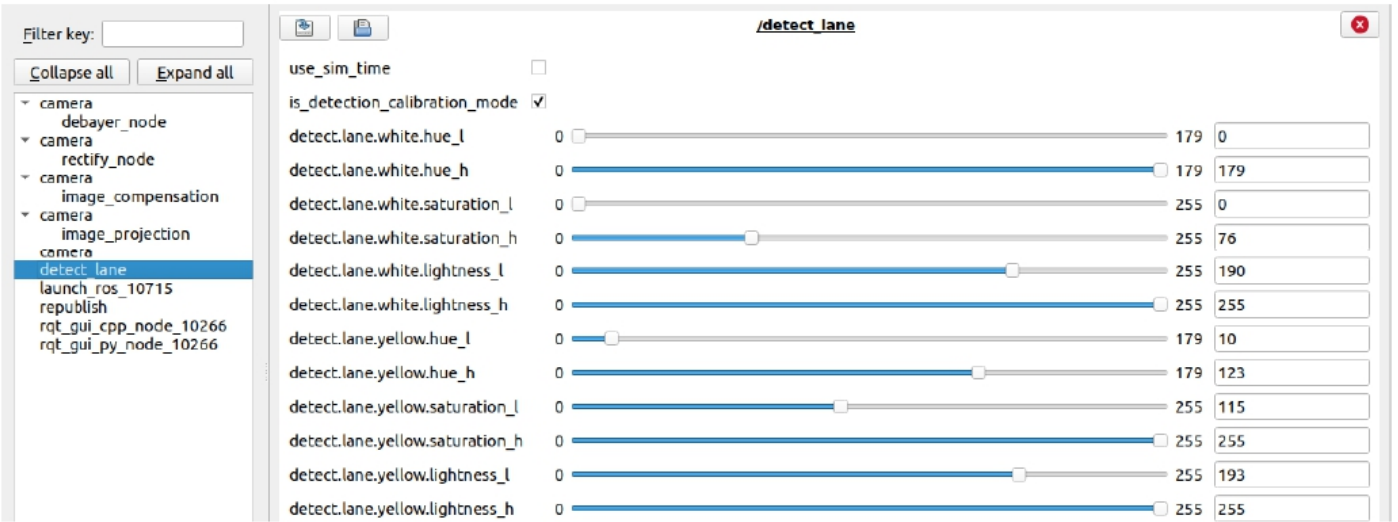


Figure 5: Threshold tuning for lane segmentation.

Part 1: Lane Detection Masks

To segment the lanes reliably, we convert images to HSV color space and apply thresholding.

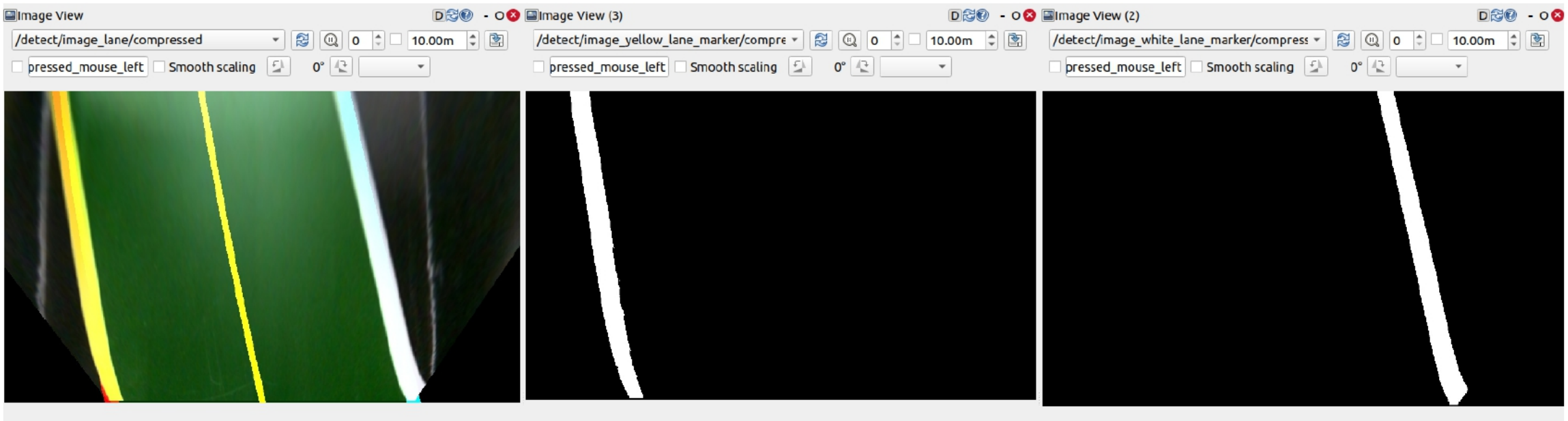


Figure 6: Lane Segmentation Pipeline Results:
Combined Detection, Yellow Mask (Left), White Mask (Right)

Control Logic: Lateral Error & Trajectory Tracking

1. Lateral Error Estimation (e): The control objective is to align the detected lane centroid (x_{lane}) with the camera's optical center (x_{center}).

$$e(t) = x_{lane} - x_{center}$$

2. PD Controller (Angular Velocity): A Proportional-Derivative controller minimizes $e(t)$ by adjusting the robot's yaw rate (ω):

$$\omega(t) = -\left(K_p \cdot e(t) + K_d \cdot \frac{de(t)}{dt} \right)$$

Where K_p determines responsiveness and K_d dampens oscillations (prevents overshooting).

3. Adaptive Linear Velocity (v): To prevent skidding on sharp curves, linear speed is throttled based on the turning magnitude:

$$v(t) = \begin{cases} v_{max} - \lambda|\omega(t)| & \text{if } |\omega| > \omega_{thresh} \\ v_{max} & \text{otherwise} \end{cases}$$

Differential Drive Kinematics

Final commands sent to the motors (Baseline L):

$$v_L = v(t) - \frac{\omega(t)L}{2}, \quad v_R = v(t) + \frac{\omega(t)L}{2}$$

Part 1: How to Run

To execute the lane following pipeline, distinct terminals are used for each node:

```
# [TurtleBot SBC] Start Camera
ros2 run camera_ros camera_node --ros-args -p format:='RGB888'

# [Remote PC] Terminal 1: Intrinsic Calibration
ros2 launch turtlebot3_autorace_camera intrinsic_camera_calibration.launch.py

# [Remote PC] Terminal 2: Extrinsic Calibration
ros2 launch turtlebot3_autorace_camera extrinsic_camera_calibration.launch.py

# [Remote PC] Terminal 3: Lane Detection
ros2 launch turtlebot3_autorace_detect detect_lane.launch.py

# [Remote PC] Terminal 4: Control (Lane Following)
ros2 launch turtlebot3_autorace_mission control_lane.launch.py

# [TurtleBot SBC] Bringup Robot
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_bringup robot.launch.py
```

Part 1: Lane Following Demo

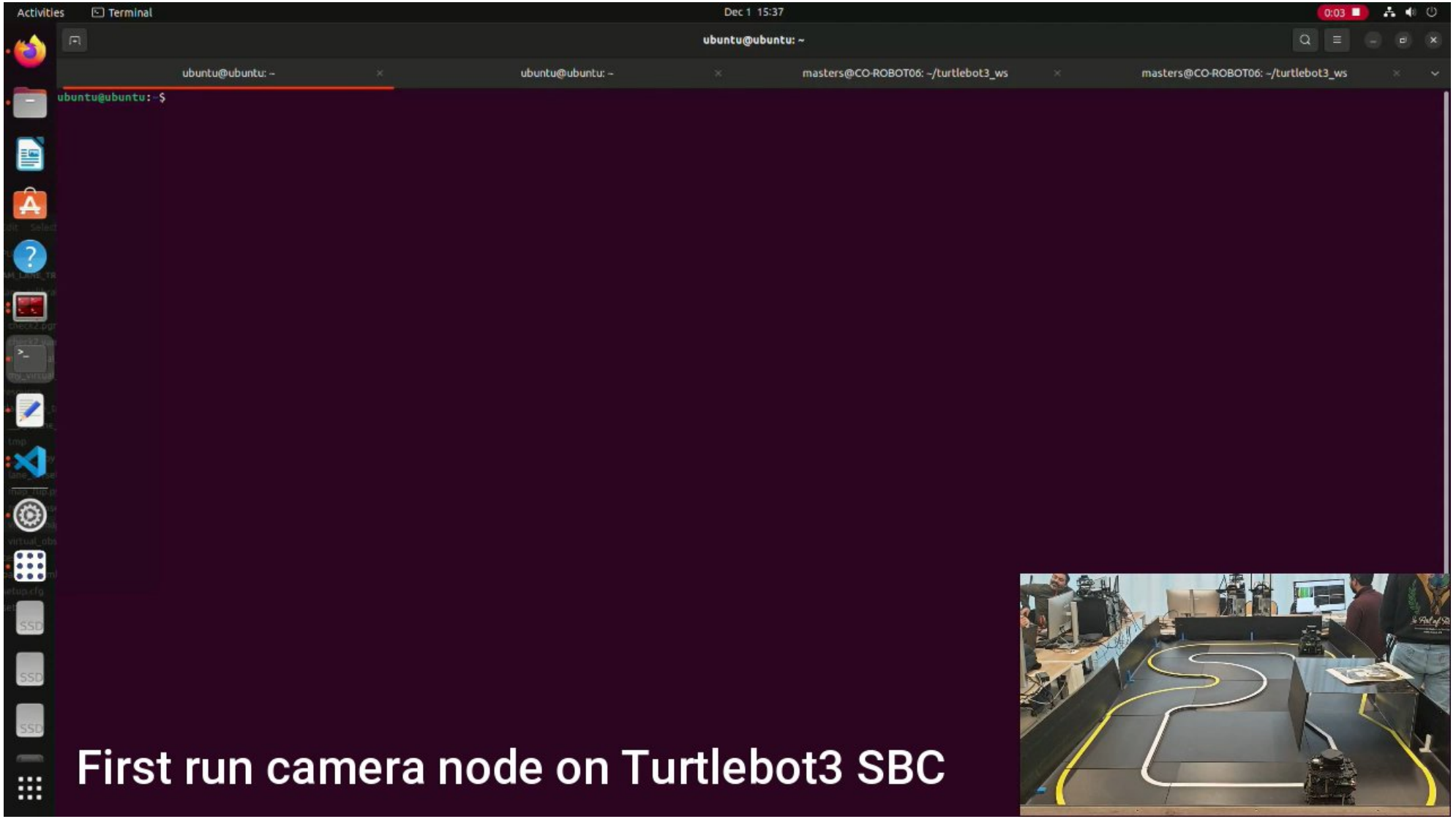


Figure 7: Real-time autonomous lane following.

Part 2: Simulation Environment

To ensure the transition to real hardware is smooth, we first reproduced the real-world track inside Gazebo.

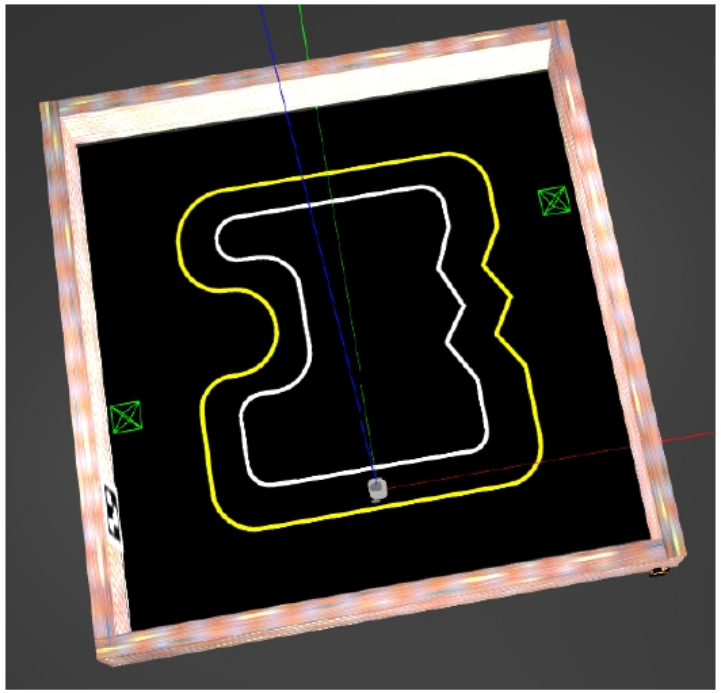


Figure 8: Simulated world mimicking the real track.

Part 2: Calculating Lane Offsets

Objective: Mapping pixel distances (C_x) to real-world meters (d_{right} & d_{left}) is crucial for the virtual map.

1. **Pixel Offsets**:

$$\Delta x_{left} = C_x - x_{yellow}$$

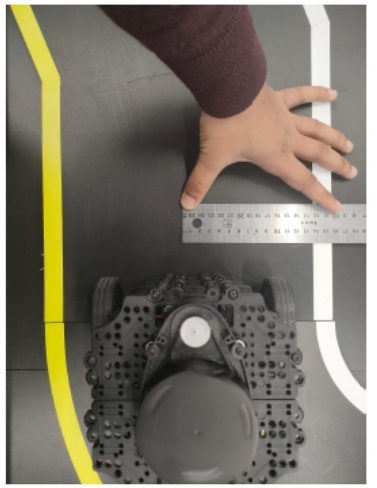
$$\Delta x_{right} = x_{white} - C_x$$

2. **Scale Factor (s)**: Measured at a known physical distance D:

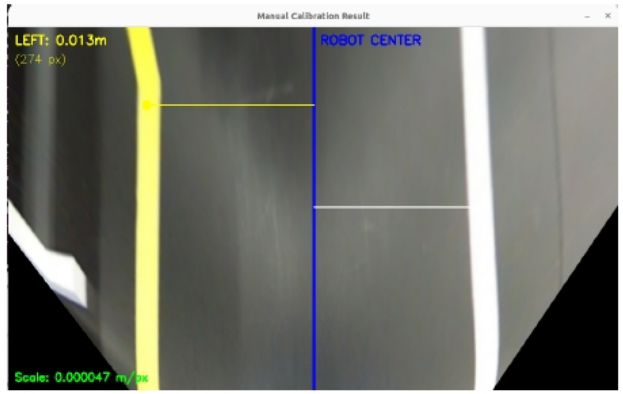
$$s = \text{meters/pixel} = D/\Delta x$$

3. **Real World Distance**:

$$d_{left} = s \cdot \Delta x_{left}, \quad d_{right} = s \cdot \Delta x_{right}$$



Calibration View



Calculated Offsets

```
masters@CO-ROBOT06:~/turtlebot3_ws$ ros2 run slam_lane_tracking_ros2 lane_offsets_logger --calibration-method manual --known-distance 0.013
[INFO] [1764581942.396569755] [lane_offset_calibrator]: Looking for topic: /camera/image_projected
[INFO] [1764581942.426475538] [lane_offset_calibrator]: ✓ Subscribed to raw image: /camera/image_projected

============================================================
LANE OFFSET CALIBRATOR - With Visual Verification
============================================================
Verification images will be saved in: lane_calibration_verification

============================================================
MANUAL CALIBRATION MODE
============================================================
1. Place your robot EXACTLY 0.013m from a lane line
2. Make sure the line is visible in the projected view
3. Press Enter when ready...

[INFO] [1764581947.210230089] [lane_offset_calibrator]: Waiting for image...
✓ Verification image saved: lane_calibration_verification/calibration_verify_20251201_103907_manual_setup.jpg

✓ Detected YELLOW (left) line at 274.0 pixels from center
✓ Calculated: 0.000047 meters/pixel
✓ Image width: 1000 pixels = 0.047 meters
✓ Verification image saved: lane_calibration_verification/calibration_verify_20251201_103907_manual_calibration.jpg
✓ Check the saved image to verify: 0.013m distance looks correct

Taking final measurement...
[INFO] [1764581949.297481333] [lane_offset_calibrator]: Waiting for image...
✓ Verification image saved: lane_calibration_verification/calibration_verify_20251201_103909_final.jpg

============================================================
CALIBRATION VERIFICATION
============================================================
✓ Verification image saved: lane_calibration_verification/calibration_verify_20251201_103909_final_verification.jpg
Please verify the measurements:
  - Check saved image: lane_calibration_verification/calibration_verify_20251201_103909_final_verification.jpg
  - Left distance:  0.013m (should match your physical measurement)
  - Right distance: 0.013m (should match your physical measurement)
```

Figure 9: Procedure of offset distance measurement

Part 2: SLAM & Map Resolution

We run SLAM (Cartographer) briefly to generate an initial map file. This provides the **Resolution** and **Origin**.

Importance: The virtual map builder needs the `resolution` (e.g., 0.05 m/pixel) to correctly translate the robot's meter-based coordinates into the pixel-based occupancy grid.

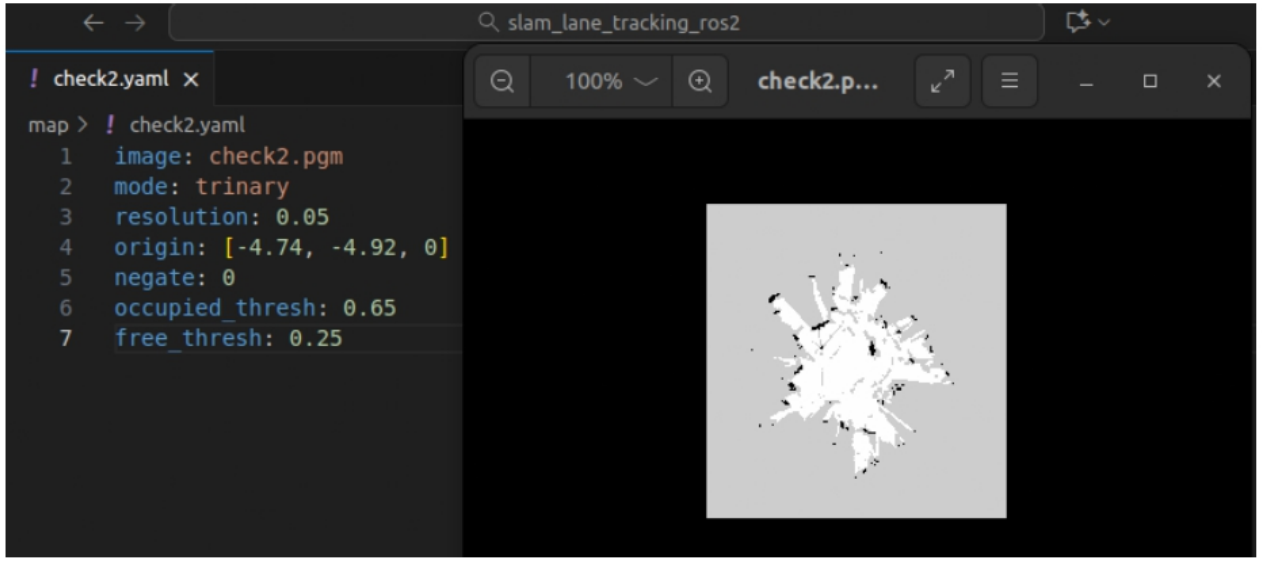


Figure 10: Initial YAML file showing Map Resolution.

Part 2: Virtual Map Builder - Mathematics

The builder constructs a polygon around the robot based on the detected lane width.

1. Robot Heading (θ): Derived from Quaternion (x, y, z, w):

$$\theta = \arctan 2(2(wz + xy), 1 - 2(y^2 + z^2))$$

2. Boundary Calculation: Calculates perpendicular points (L, R) from robot center:

$$x_L = x + L\cos(\theta + \tfrac{\pi}{2}) \quad x_R = x + R\cos(\theta - \tfrac{\pi}{2})$$
$$y_L = y + L\sin(\theta + \tfrac{\pi}{2}) \quad y_R = y + R\sin(\theta - \tfrac{\pi}{2})$$

3. World to Map:

$$p_x = \frac{x - x_0}{r}, \quad p_y = H - 1 - \frac{y - y_0}{r}$$

4. Polygon Filling:

- Collect points $P = [L_1 \dots L_n, R_n \dots R_1]$.
- OpenCV `fillPoly`: Inside $= 255$ (Free), Outside $= 0$ (Occupied).

Part 2: Virtual Map Builder - Mathematics (cont'd)

How we convert the visual polygon into a ROS-compatible Map.

1. **Polygon Filling (Canvas)**:
   - OpenCV `fillPoly` paints the drivable area White (255) and obstacles Black (0).
2. **Conversion to PGM**:

$$\text{pgm}(x, y) = \begin{cases} 254 \text{ (Free)}, & \text{if canvas} = 255 \\ 0 \text{ (Occupied)}, & \text{if canvas} = 0 \end{cases}$$

3. **YAML Generation**: The script automatically exports a `.yaml` file matching the PGM, containing:
   - `image`: path to pgm
   - `resolution`: from SLAM
   - `origin`: $[x_0, y_0, 0]$
   - `negate`: 0
   - `occupied_thresh`: 0.65
   - `free_thresh`: 0.196

Part 2: Virtual Map Result

The algorithm runs in simulation as the robot traverses the lane, "painting" the drivable area.

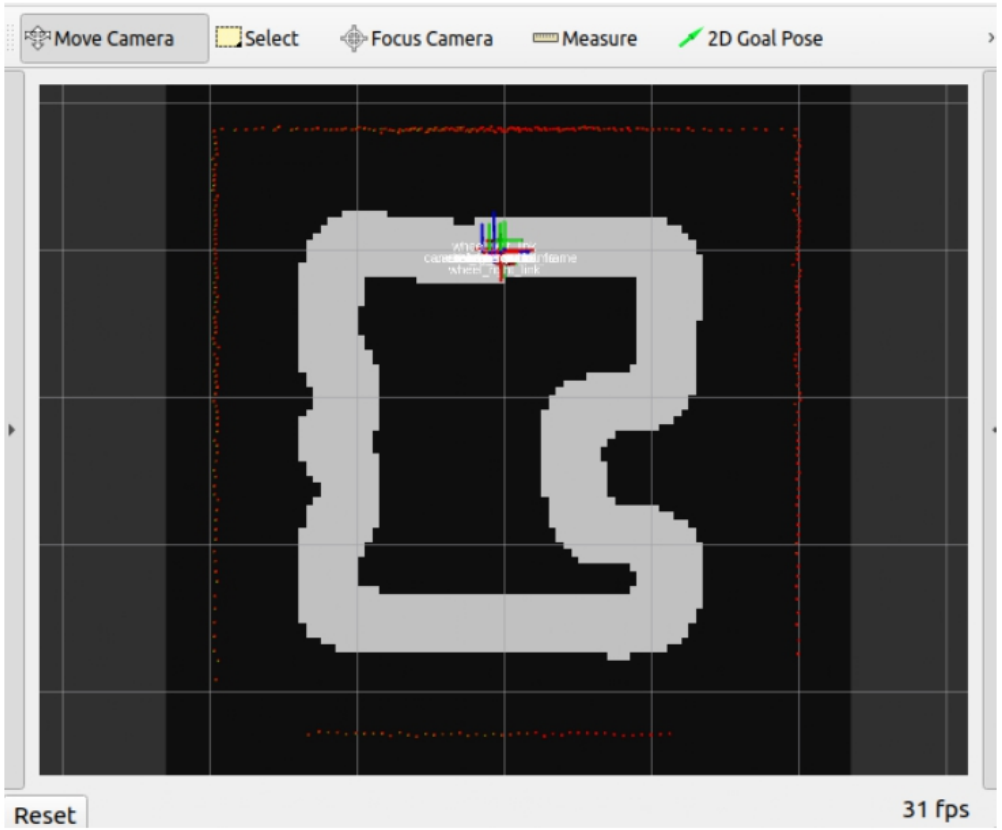


Figure 11: The generated Virtual Obstacle Map (pgm/yaml) ready for Nav2.

Part 2: ArUco Tag for Origin Fixation

Problem: The robot's starting position in the real world is arbitrary, but the map requires a fixed reference.

Methodology

- An **ArUco tag** is placed at the origin.
- We run the detector to find the exact pose.
- **Command**: `ros2 run slam_lane_tracking_ros2 aruco_detector --ros-args -p calibration:=True`

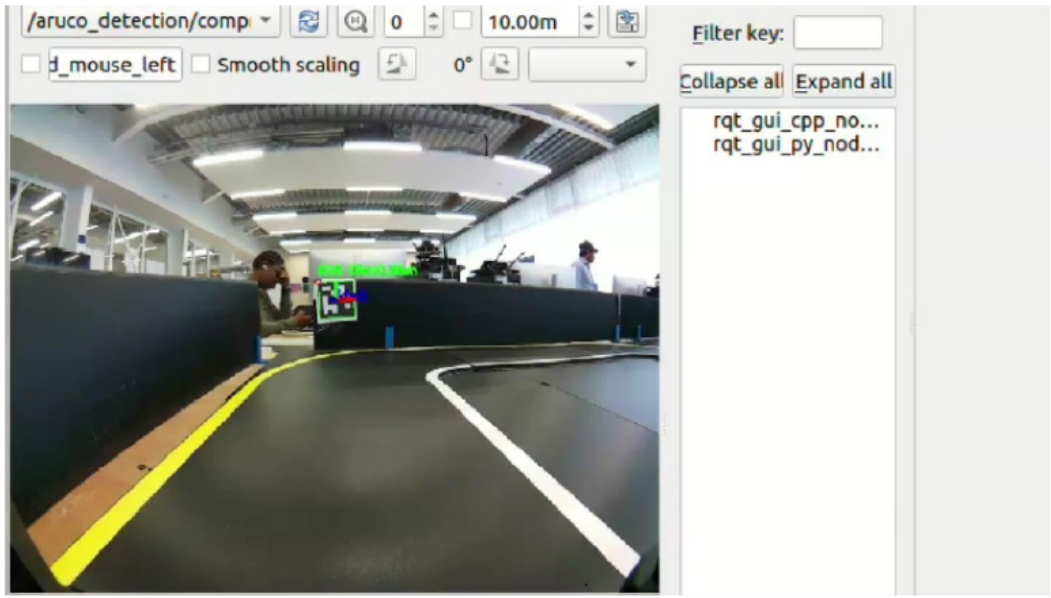


Figure 12: ArUco setup for Origin detection.

Part 2: How to Run Virtual Map

```
# [Terminal 1] Bringup (SBC)
ros2 launch turtlebot3_bringup robot.launch.py

# [Terminal 2] Camera (SBC)
ros2 run camera_ros camera_node --ros-args -p format:='RGB888'

# [Terminal 3] Calibration/Detect (Remote PC)
ros2 launch turtlebot3_autorace_camera intrinsic_camera_calibration.launch.py
ros2 launch turtlebot3_autorace_camera extrinsic_camera_calibration.launch.py
ros2 launch turtlebot3_autorace_detect detect_lane.launch.py

# [Terminal 4] Cartographer (For Localization)
ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=False

# [Terminal 5] Virtual Map Builder (Calculates Polygon)
ros2 run slam_lane_tracking_ros2 virtual_map_builder

# [Terminal 6] Save Map (When Complete)
ros2 run nav2_map_server map_saver_cli -f ~/my_virtual_map -t /virtual_map
```

Part 2: Virtual Map Results (Sim vs Real)

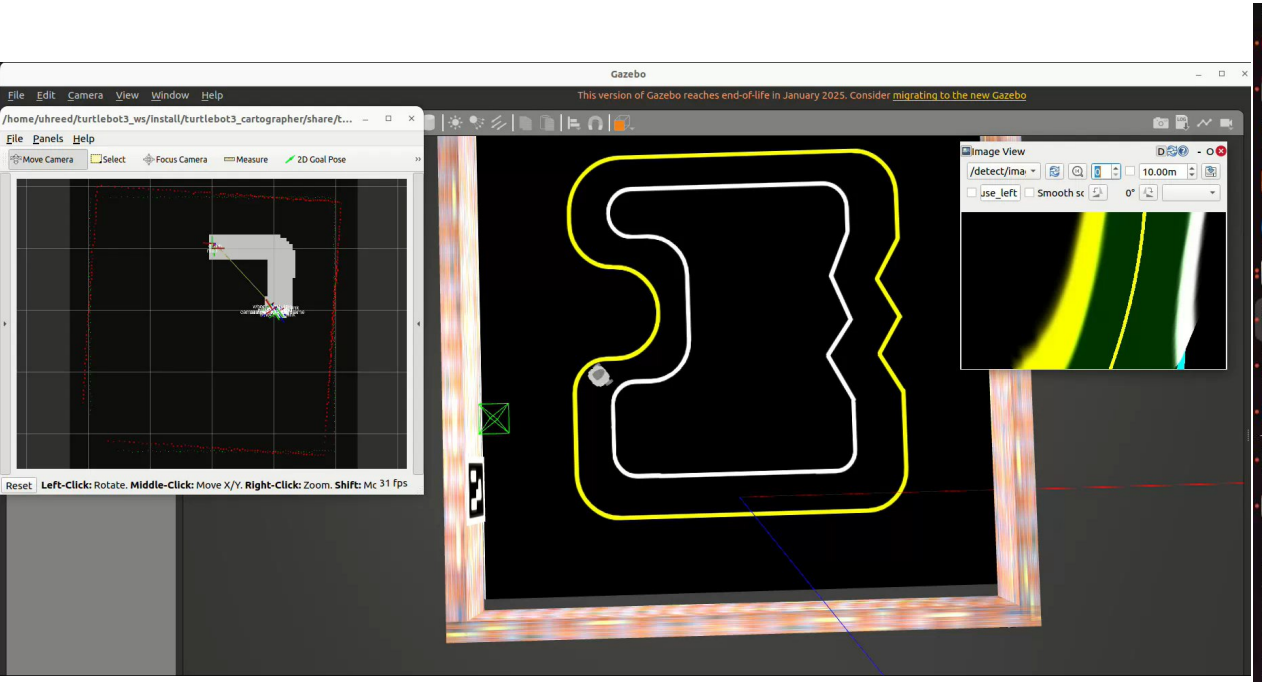


Figure 13: Simulation Result.

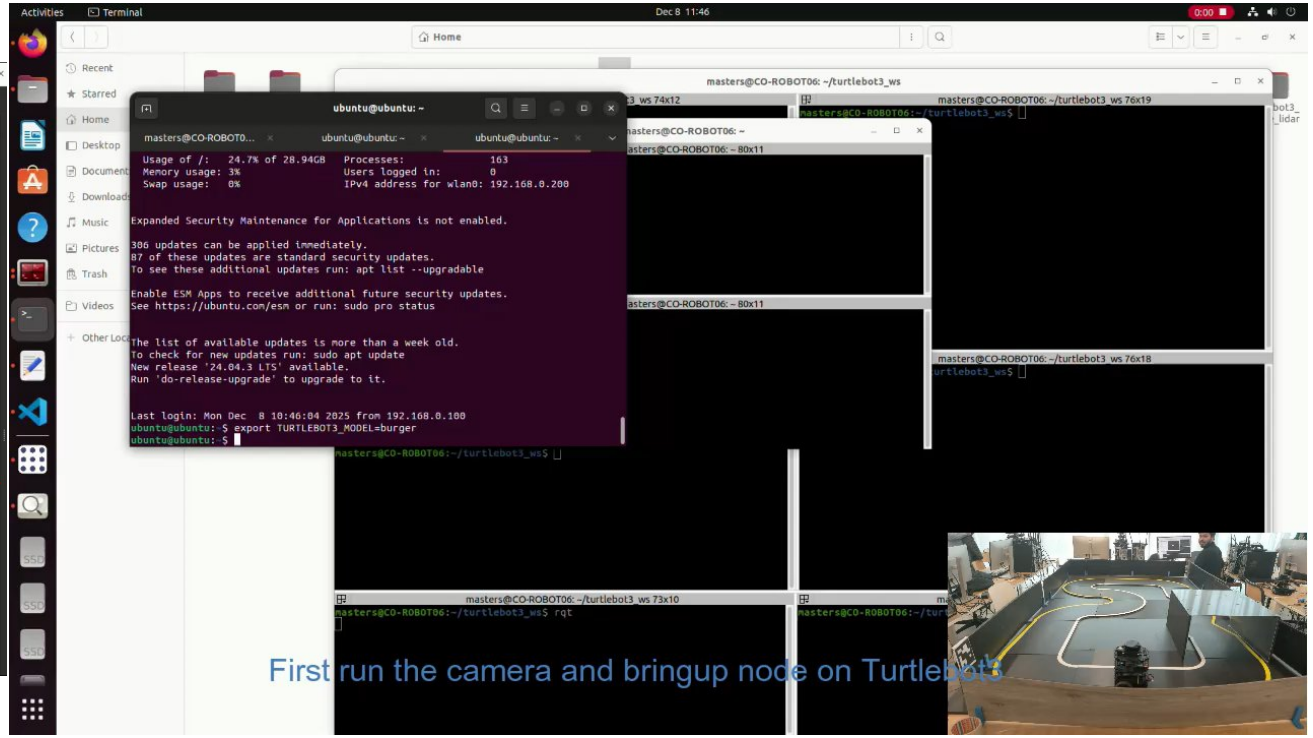


Figure 14: Real-World Result.

Navigation: Setup & Tuning

Once the map is saved, we transition to the Navigation Stack (Nav2).

- **Loading Origin**: We utilize the origin calculated via the ArUco step to initialize the localization.
- **Parameter Tuning**: The standard `burger.yaml` parameters are too loose for the narrow track.
  - `robot_radius`: Reduced to 0.05.
  - `inflation_radius`: Reduced to 0.1 to allow movement in tight lanes.

Part 2: ArUco Calibration Demo

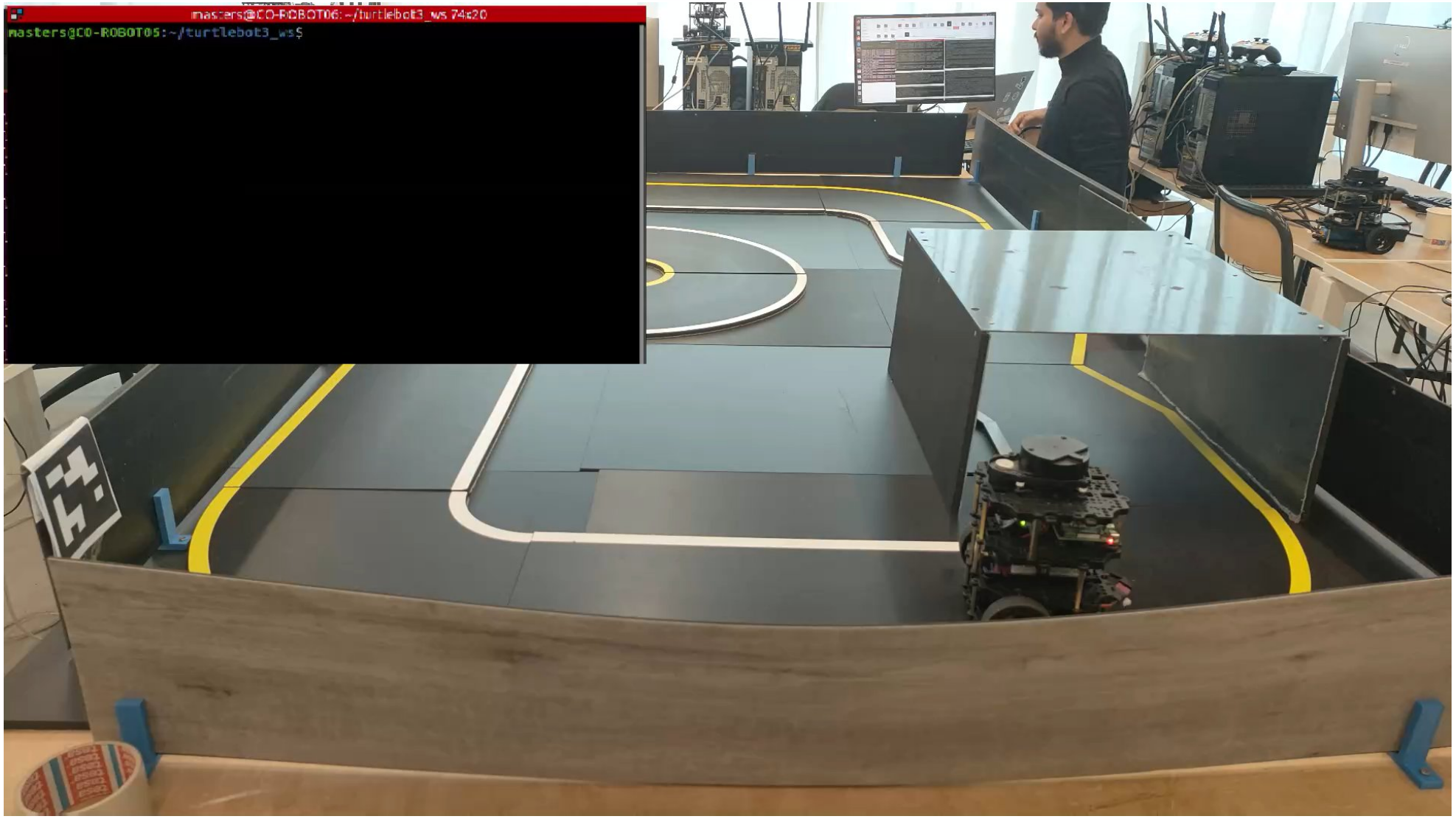


Figure 15: Finding the origin point before navigation.

Navigation: Execution

Process:

1. Launch `navigation2.launch.py` with the virtual map.
2. Set initial pose estimation (using ArUco coordinates).
3. Send goal via Rviz/Nav2 Goal.

Navigation: Execution (Sim vs Real)

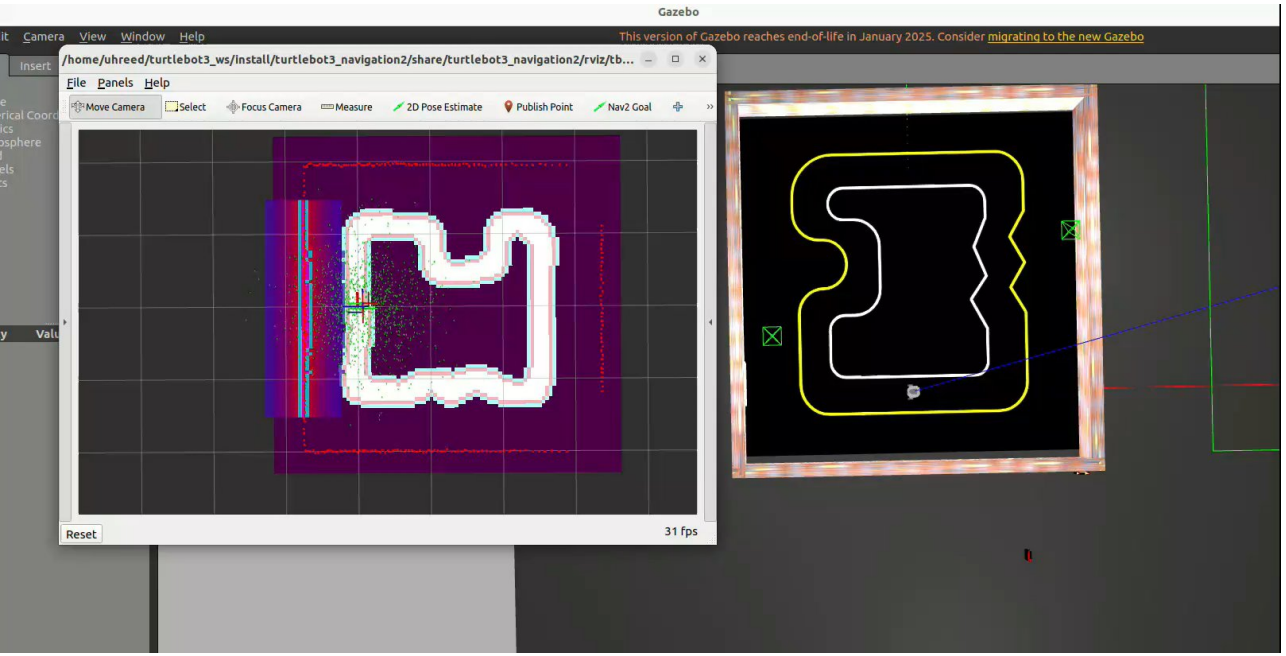


Figure 16: Simulation Result.

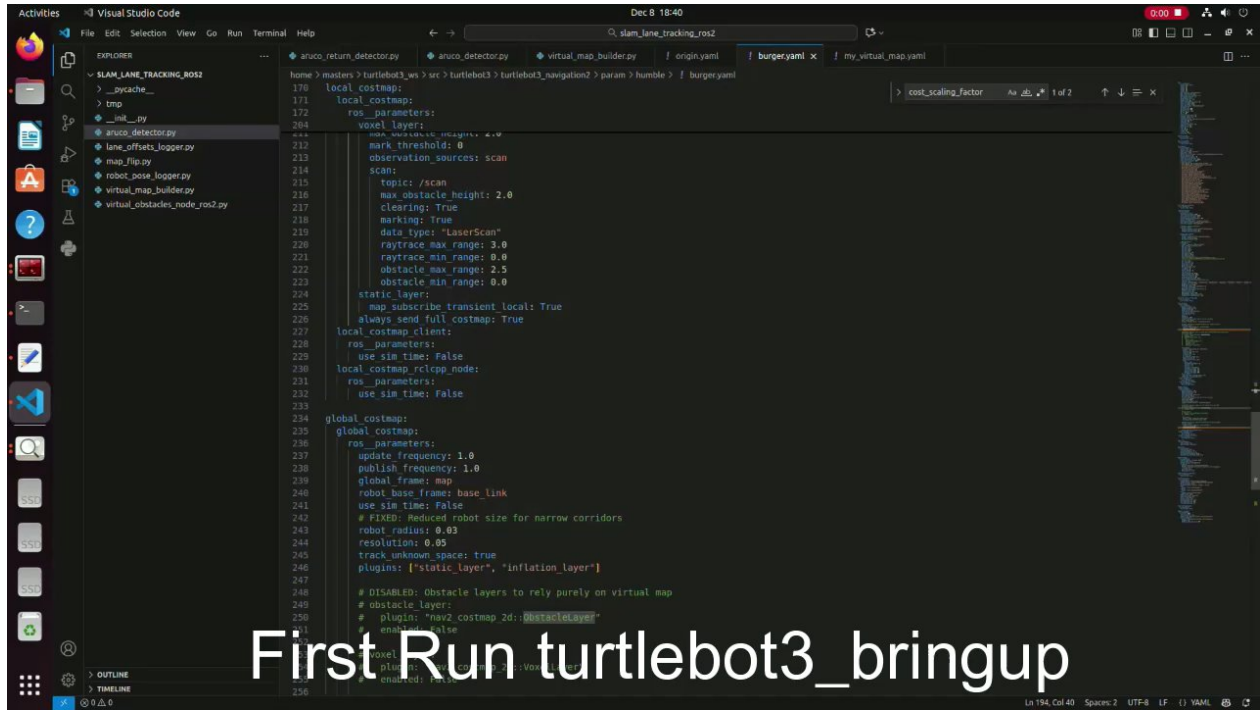


Figure 17: Real-World Result.

Future Work

Lane-Aware Costmap

Currently, the robot relies on a static virtual map created beforehand.

Proposed Improvement:

- Generate a **Real-Time Costmap Mask** directly from live lane detection.
- Convert detected lane boundaries immediately into an occupancy mask (100 = Outside, 0 = Inside).
- This transforms the robot into a truly lane-aware autonomous system without pre-mapping.

Thank You