I2ROS SS25 Project Report for Group 14

Jiaxiang Yang
jiaxiang.yang@tum.de

Qianke Ye
qianke.ye@tum.de

Enze Wang
enze.wang@tum.de

Yicun Song
yicun.song@tum.de

Abstract

This report presents the work of Group 14 for the course Introduction to ROS at TUM in the Summer Semester 2025. The project focuses on building an autonomous driving system using ROS Noetic, integrating Unity-based simulation with modules for perception, planning, and control. Within the simulated urban environment, the vehicle is required to follow a predefined route efficiently while obeying traffic rules and ensuring safety. The system is designed to detect traffic lights, recognize nearby vehicles, maintain lane boundaries, and avoid collisions. This report outlines the division of tasks among team members, describes the structure and function of each module, and highlights the key implementation details. The document concludes with a brief discussion and reflection on the outcomes and potential improvements of the project.

1 Introduction

The objective of this project is to enable an autonomous vehicle to complete a predefined path in a Unity-based simulation environment. To achieve this, the vehicle must perceive its surroundings, identify the correct route, detect traffic lights, and avoid collisions. A state machine is used to ensure compliance with traffic rules, while a vehicle controller is responsible for handling acceleration, braking, and steering. Based on these functional requirements, the system is structured into four main modules: perception, planning, state machine, and control. This report first outlines the team's task distribution (see Tab. I) and provides a detailed explanation of each module. A ROS graph illustrating the overall system architecture (see Fig. 0) is also included. Any remaining unmet requirements and use of external code are clearly documented to ensure transparency.tructions.

2 PERCEPTION

The perception module is designed to extract meaningful information from raw sensory data, enabling the autonomous vehicle to understand its environment and make informed decisions. This includes detecting relevant objects in the scene, estimating their spatial positions, and interpreting traffic-related cues that influence driving behavior. In this project, the perception pipeline primarily addresses two essential tasks: vehicle detection and distance estimation, and traffic light recognition and status publishing.

For vehicle detection, the system leverages semantic segmentation data to identify other cars in the scene and combines this with depth information to compute their relative positions in the camera frame. This spatial awareness is critical for collision avoidance and local planning. For traffic light recognition, the module detects light poles using semantic input and classifies the light color based on RGB intensity values, allowing the system to determine whether to stop or proceed. Furthermore, it estimates the 3D distance to each traffic light, providing necessary input for high-level decision-making modules.

Both tasks are implemented using synchronized RGB, semantic, and depth image streams from Unity-based simulated sensors. The data are processed in real-time using ROS, OpenCV, and standard perception algorithms, ensuring reliable and interpretable outputs that support downstream modules such as planning and control.

2.1 Coordinate Frame Build-up and Occupancy Map

To enable spatial consistency across various sensors in the simulation environment, a series of static coordinate transformations is established using multiple instances of the *static_transform_publisher* node from the *tf2_ros* package. These transformations define the relative positions between the INS, vehicle center, sensor base, and individual cameras (RGB, semantic, and depth). The complete TF

tree ensures that all sensory data—especially depth and image streams—are aligned within a unified spatial reference frame, enabling accurate 3D projection and data fusion.

The core transformation chain is as follows:
OurCar/INS → OurCar/Center → OurCar/Sensors/SensorBase → [Each Camera Frame]

This setup ensures that the spatial relationships between sensors are consistent and allows real-time modules to transform data across frames as needed.

To construct a 3D representation of the environment, the system integrates a **depth-to-point cloud conversion and filtering pipeline** with the OctoMap framework. The *depth_image_proc/point_cloud_xyz* nodelet converts raw depth images from the front camera into unstructured 3D point clouds. These points are then passed through a Z-axis height filter (*PassThrough* node from the *pcl package*, which retains only the points within a height range of 0 to 0.3 meters, reducing ground noise and eliminating irrelevant obstacles.

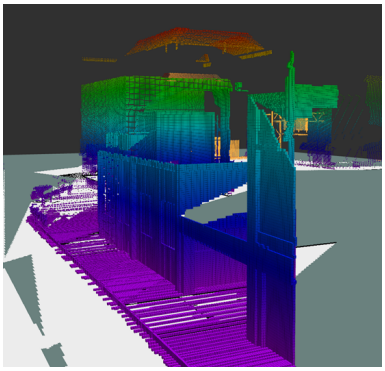

Figure 1. 3D Octomap

The filtered point cloud is subsequently fed into the octomap_server_node, which builds a probabilistic occupancy map using the octree-based OctoMap framework. The resolution is set to 0.1 meters, and a 2D projection of the map is also enabled. The resulting occupancy map provides a compact and efficient spatial representation of the environment, suitable for real-time planning and navigation. The entire mapping process is visualized in RViz using a preconfigured display configuration file.

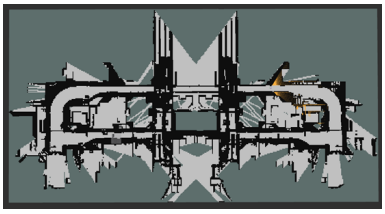

Figure 2. 2D projection of the map

2.2 Vehicle Detection and Distance Estimation

The *car_detector* node is responsible for detecting vehicles in the driving scene and estimating their distances using fused semantic, RGB, and depth camera data. Semantic segmentation images are used to identify red-colored regions corresponding to cars. A binary mask is applied to extract contours of potential vehicle instances. Contours with very small areas are filtered out to reduce false positives, and bounding boxes are computed for valid targets.

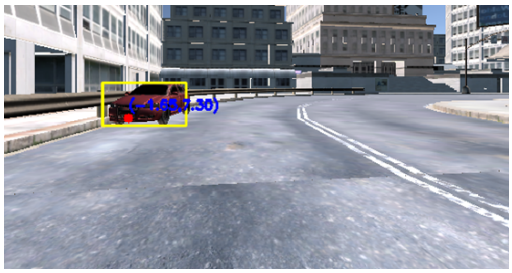

Figure 3. Vehicle Detection and Distance Estimation Using Semantic and Depth Data

To associate depth data with detected vehicles, bounding boxes are upscaled to match the resolution of the depth and RGB images. Within each ROI, only valid depth values (typically between 0 and 100 meters) are considered. To ensure robustness against noise and outliers, the **median depth value** is selected from the valid region.

In order to convert the 2D detection into a 3D camera coordinate, camera intrinsics—including focal lengths and the principal point—are used via the pinhole camera model:

$$X = \frac{(u - c_x) \cdot d}{f_x}, \quad Z = d$$

where u is the pixel coordinate corresponding to the depth sample, d is the selected depth value, and Z represents the forward axis in the camera coordinate frame. The pixel coordinate (u,v) is chosen based on the pixel with the **smallest depth** inside the ROI, to better approximate the nearest surface of the vehicle.

A depth unit scaling mechanism is also implemented to handle varying encodings (e.g., **uint16** or **float32**) based on observed data ranges. Detected positions are visualized on the RGB image with red dots and text labels showing their 2D projected positions in meters. The final 3D coordinates (X, Z) for all detected vehicles are published through the *car_detection_distances* topic for downstream motion planning modules. The annotated RGB frame is published via *car_detection_roi_vis* for runtime monitoring and debugging.

The node also supports dynamic logging and semantic color introspection upon the first run, assisting developers in adjusting color filters for different semantic camera settings.

2.3 Traffic Light Recognition

The *traffic_light_color_detector* node determines the current state of nearby traffic lights by synchronizing and analyzing RGB, semantic, and depth image streams. Semantic segmentation is used to locate yellow poles, which typically indicate traffic light structures. The corresponding pixel regions are upscaled to match the RGB resolution and used to define Regions of Interest (ROIs) for further analysis.

Within each ROI, a **channel-dominance thresholding method** is applied to classify the light color. Specifically, the algorithm compares the intensity of each RGB channel and identifies a pixel as red or green based on whether the red or green channel is dominant and exceeds a predefined brightness threshold. The total number of red and green pixels is computed for each ROI, and the result is used to determine the overall status:

- 0 indicates red light is present,
- 1 indicates green light,
- -1 indicates no clear signal detected.

Each detected traffic light is visualized with a colored bounding box—green, red, or yellow—based on the classification result. Text annotations are also added beside the boxes to display the recognized state.

To improve robustness, the algorithm discards small ROIs (less than 5 pixels in width or height) and clips all bounding box coordinates to avoid exceeding image boundaries. A slight horizontal offset is applied to the extracted region to better center the traffic light in the ROI.

The final traffic light state is published via the */traffic_light_status* topic and can be consumed by behavior planning modules. The annotated RGB image is simultaneously published on */traffic_light_roi_vis* for visualization and debugging.

2.4 Data Synchronization and Camera Alignment

Both the vehicle detection and traffic light recognition nodes rely on synchronized input from three simulated camera sensors: RGB, semantic, and depth images. To achieve this, the system uses *ApproximateTimeSynchronizer* from the *message_filters package*, ensuring that incoming messages with slightly different timestamps are matched and processed together. This is essential in simulation environments where perfect synchronization is not always guaranteed.

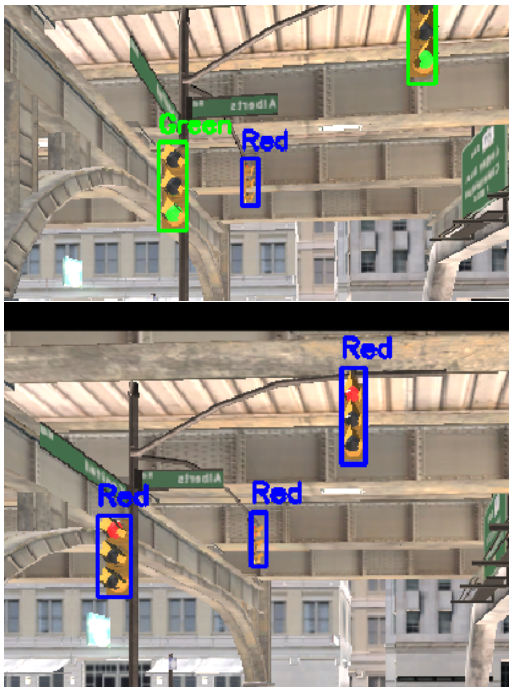

Figure 4. Traffic Light Recognition Using Semantic Segmentation and RGB Analysis

Since the image resolutions differ across modalities (e.g., semantic images may be lower resolution than RGB or depth), the system calculates spatial scaling factors for both axes and resizes bounding box coordinates accordingly. These adjustments ensure that each region of interest (ROI) identified in one modality correctly maps to the same location in the others. This alignment is crucial for depth-aware detection, as incorrect scaling would result in inaccurate pixel-to-distance associations and failed projections.

In addition, camera intrinsic parameters such as focal lengths and principal points are manually specified to support accurate back-projection from image space to 3D coordinates. For all tasks requiring 3D localization—such as vehicle and traffic light distance estimation—these calibrations are used consistently to ensure spatial accuracy.

2.5 Implementation Overview

The entire perception pipeline is implemented in Python using the Robot Operating System (ROS) and OpenCV. It consists of modular ROS nodes that independently handle tasks such as color-based object segmentation, depth extraction, geometric projection, TF tree construction, point cloud filtering, and occupancy mapping.

All real-time processes are supported by visualization

outputs in RViz and image publishers. For instance, detected vehicles and traffic lights are annotated with bounding boxes and coordinate labels and displayed through dedicated image topics. Logging statements are embedded throughout the pipeline, allowing developers to inspect runtime information such as depth values, bounding box dimensions, and semantic segmentation feedback.

While no deep learning models were used in this pipeline, the system demonstrates that carefully tuned, rule-based perception techniques—leveraging domain-specific knowledge of color masks, TF hierarchies, and camera models—can achieve highly interpretable and reliable results in structured simulation environments. The design prioritizes modularity, clarity, and ease of debugging, making it ideal for educational and prototyping purposes.

3 Decision Making Module

The **decision-making** module is responsible for issuing high-level driving commands by evaluating perception and localization results. It ensures the vehicle behaves safely and logically in response to its surroundings.

3.1 Module Responsibility

This module determines whether the vehicle should proceed or stop by processing real-time traffic light status, obstacle distances, and vehicle localization data. It outputs a Boolean signal to the control system for execution.

3.2 Inputs and Outputs

Subscribed Topics

- */traffic_light_status (Int32)*
  Indicates the current traffic light status:
  - 1: Green
  - 0: Red
  - -1: Undetected (treated as green by default)
- */car_detection_distances (Float32MultiArray)*
  Array of distances to detected vehicles ahead.
- */odom (nav_msgs/Odometry)*
  Provides the vehicle's current position for decision context.

Published Topic

- */ssc (Bool)*
  High-level decision command:
  - True: Proceed
  - False: Stop

3.3 Decision Logic

The node evaluates all inputs periodically and applies a **priority-based rule system**:

1. **Emergency Braking** If a vehicle is detected within an emergency threshold, the system issues an immediate stop command.
2. **Traffic Light Check**
   - Red light → Stop
   - Undetected light (-1) → Assume green by default for continuity
3. **Proceeding** If no danger is detected and the light is not red, the vehicle is allowed to continue.

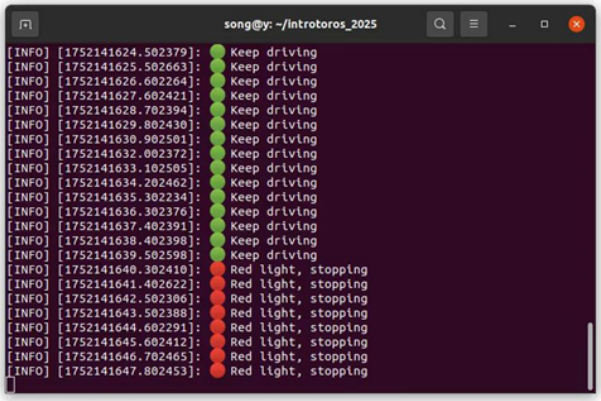

Figure 5. Terminal Output of Decision-Making Module Based on Traffic Light Status

3.4 Robustness and Safety Features

Fail-Safe Defaults In case of missing inputs, fallback values (e.g., assume green light) are used to prevent system freezes.

Decoupled Timing The module runs independently at a fixed rate (e.g., every 0.1s) to ensure consistent decision-making.

Emergency Handling Priority Emergency braking always overrides other conditions for safety.

3.5 Integration with Other Modules

Perception: Supplies obstacle distances and traffic light states. **Localization:** Provides odometry for vehicle state awareness. **Control:** Receives */ssc* to execute proceed/stop commands.

3.6 Conclusion and Future Work

This rule-based module ensures basic safe navigation in autonomous driving scenarios.
It is **modular**, **robust**, and easy to integrate with upstream and downstream modules.

Potential Enhancements

- Incorporate vehicle dynamics (speed, acceleration) into decision logic
- Use behavior trees or FSMs for complex traffic handling
- Add lane-level logic: merging, overtaking, intersections
- Transition toward learning-based or hybrid decision-making approaches

4 Vehicle Control Module

4.1 Overview

In the autonomous driving simulation system, the vehicle control module serves as a key component that bridges the gap between high-level planning decisions and low-level actuation commands. It is responsible for interpreting the desired motion trajectories and producing appropriate throttle, brake, and steering signals that drive the vehicle within the simulated environment. This module was implemented using the Robot Operating System (ROS) and was designed to operate in real-time with robust performance and clear modularity.

4.2 Module Functionality

The control module is encapsulated in a dedicated ROS node that operates continuously at a fixed frequency. It interacts with the rest of the system through several ROS topics, receiving the vehicle's current state and the planner's desired trajectory, and publishing control commands accordingly.

4.2.1 Inputs

- **Desired Velocity:** Received as a *geometry_msgs/Twist* message, this contains the linear and angular velocities that the vehicle is expected to follow. It represents the trajectory output from the motion planner.
- **Current State Feedback:** Provided via a *geometry_msgs/TwistStamped* message, this includes the current velocity of the vehicle, obtained from simulation sensors or virtual odometry.
- **Decision Signal:** A boolean flag indicating whether the vehicle should proceed or stop, typically reflecting high-level decisions like traffic signal compliance or emergency stops.

4.2.2 Outputs

- **Vehicle Control Command:** The module outputs control commands in the form of a custom message type. These commands include throttle, brake, and steering values that are consumed by the simulator's vehicle model to move the car accordingly.

4.3 Control Strategy

The control algorithm follows a structured decision-making pipeline that prioritizes safety and responsiveness.

4.3.1 Emergency and Decision Handling

A high-priority check is always performed to determine if the vehicle should stop. If a `STOP` signal is received from the decision-making module, the controller immediately overrides any motion commands and issues full braking to halt the vehicle safely and promptly.

4.3.2 Longitudinal Control

The longitudinal controller regulates the forward motion of the vehicle. It computes the difference between the desired and current velocities and applies a simple proportional control strategy to determine the level of throttle or braking required. Acceleration is capped by a predefined maximum to ensure physical feasibility and smooth driving behavior. Braking is similarly governed by a proportional response to negative velocity errors.

4.3.3 Lateral Control

The lateral controller is responsible for steering. It uses an Ackermann kinematic model to convert the desired angular velocity and linear velocity into a steering angle. The resulting steering command is normalized and constrained within a physical maximum steering angle. This ensures that the vehicle can follow curved paths accurately while maintaining stability and realism.

$$\delta = \arctan\left(\frac{L \cdot \omega}{v}\right) \tag{1}$$

4.4 Parameter Configuration

The control module is highly configurable through external YAML files. Key parameters include the vehicle's wheelbase, maximum steering angle, acceleration and

speed limits, and proportional control gains for throttle and braking. These parameters are loaded at runtime and allow for fine-tuning of the vehicle's dynamic response without modifying the source code.

4.5 System Integration

The controller was fully integrated into the larger ROS-based simulation architecture. It worked in conjunction with the planning module, decision-making components, and simulated sensors. This allowed for dynamic driving behavior that responds both to environmental conditions (such as traffic signals) and to path planning objectives, ensuring the vehicle could navigate effectively in a variety of scenarios.

5 Planning

5.1 Introduction

The ROS Navigation Stack is a collection of packages designed to enable autonomous navigation for mobile robots. It allows a robot to move from one point to another safely and efficiently in a known environment using sensor data and a map. The stack integrates path planning, obstacle avoidance, and localization functionalities. Key components include:

- **move_base:** The central node that ties together global and local planners to generate and execute safe paths.
- **Global Planner:** Plans a path from the current position to the goal using a map (e.g., Dijkstra or A*).
- **Local Planner:** Creates short-term, dynamic paths that avoid obstacles while following the global path.
- **Costmaps:** Represent the environment, including static and dynamic obstacles, for the planners.

This stack is widely used in autonomous robot systems and provides a flexible and modular framework for path planning and navigation tasks.

5.2 Parameter files

5.2.1 base_local_planner_params.yaml

This file configures the **local planner**, which is responsible for generating velocity commands to safely navigate the robot along the global path while avoiding obstacles. It typically includes parameters like:

- **max_vel_x, min_vel_x:** Linear velocity limits.

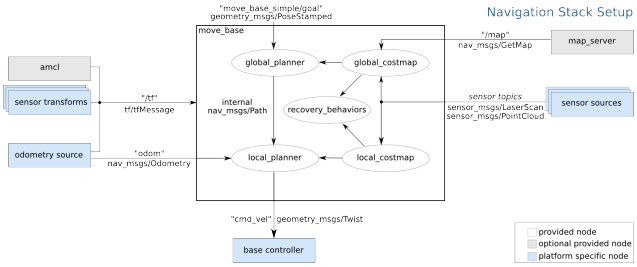

Figure 6. Overview of Navigation

- **max_rotational_vel:** Angular velocity limit.
- **acc_lim_x, acc_lim_theta:** Acceleration limits.

This configuration directly influences how smoothly and reactively your robot moves in real-time around obstacles.

5.2.2 costmap_common_params.yaml

This file includes settings shared between the **global and local costmaps**, which are 2D occupancy grids used for obstacle avoidance and path planning. It typically contains:

- **obstacle_range, raytrace_range:** Sensor range settings for detecting obstacles.
- **footprint:** Size of the robot used to inflate obstacles.
- **inflation_radius:** How far obstacles are expanded in the costmap to ensure safe navigation.
- **observation_sources:** Sensor topics used to populate the costmaps.

These parameters help define the robot's interaction with its environment and ensure safe path planning.

5.2.3 global_costmap_params.yaml

This file sets up the **global costmap**, used by the global planner to calculate a path from the current location to the target goal using a static map. Key settings include:

- **global_frame:** Reference frame (world).

These settings ensure the planner knows where the robot is in the world and can build a feasible long-term path.

5.2.4 local_costmap_params.yaml

This file configures the **local costmap**, which represents a short-term, sensor-based map around the robot. It's used for dynamic obstacle avoidance. Typical settings include:

- **rolling_window:** True, the costmap moves with the robot.

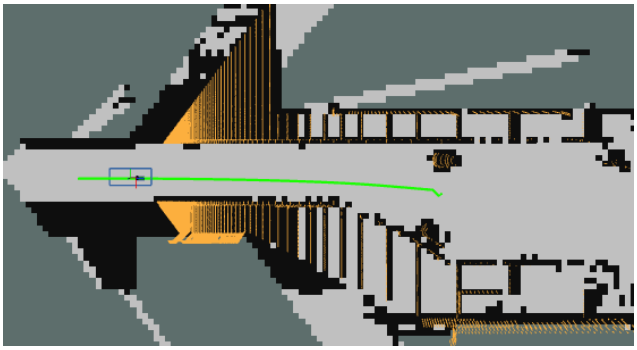

Figure 7. Path Planning

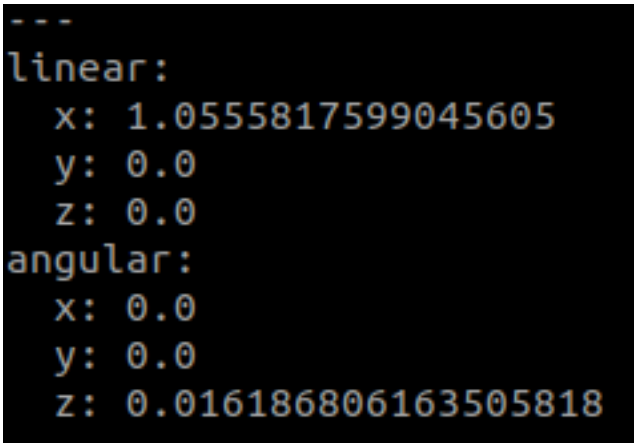

Figure 8. Command Velocity Sending

- **update_frequency, publish_frequency:** True, the costmap moves with the robot.

This costmap allows the robot to adapt on the run to unexpected obstacles or people crossing its path.

5.3 Odometry Publisher Node

In order for the **move_base** node to function properly, it requires accurate and timely vehicle pose and velocity data, typically provided on the **/odom** topic. The **odometry_publisher** node fulfills this role by aggregating pose and twist data from separate sources and publishing a standardized **nav_msgs/Odometry** message.

5.3.1 Functionality Overview

- **Input Topics:**

  /Unity_ROS_message_Rx/OurCar/CoM/pose : Provides the vehicle's current position and orientation. **/Unity_ROS_message_Rx/OurCar/CoM/twist (geometry_msgs/TwistStamped)**: Provides the vehicle's linear and angular velocity.

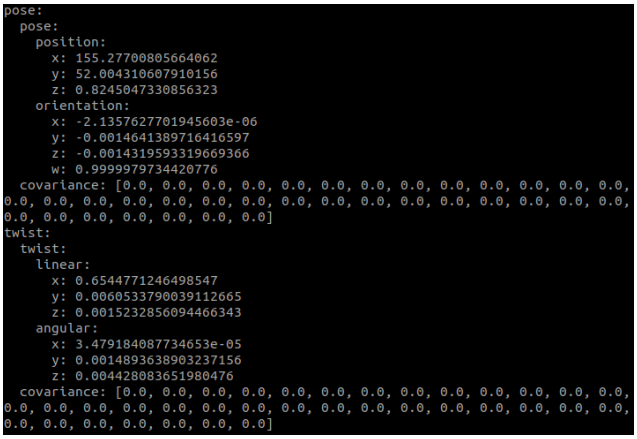

Figure 9. Odometry Messages

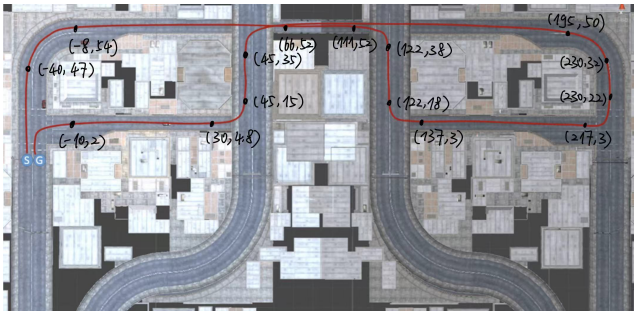

Figure 10. Short-term Goal Selection

- **Output Topic: /odom (nav_msgs/Odometry)**: Combines position and velocity into a single message for the Navigation Stack.

Core Components:

TF2 Integration: The node uses a TF2 buffer and listener to transform velocity data from its original frame into the vehicle's body frame (e.g., "OurCar/INS"), which is necessary for consistent odometry.

Covariance Matrices: Both pose and twist covariance matrices are set to zero by default, but can be populated with sensor uncertainty values if available.

Publishing Frequency: A timer callback publishes updated odometry data every 0.02 seconds (50 Hz), which ensures smooth performance for planners and controllers.

5.3.2 Purpose in Navigation

This node is essential in providing a reliable odometry stream to move_base, enabling accurate localization, path planning, and control. It acts as a bridge between raw sensor outputs and the standardized messages expected by the ROS Navigation Stack.

5.4 Simple Navigation Goals Node

The **simple_navigation_goals** node is responsible for sending a series of navigation goals to the **move_base** node and monitoring the vehicle's progress toward each goal. This node acts as the mission executor, providing persistent and autonomous goal execution.

Goal Data Input: The structure of the node begins with the definition of a simple **GoalPoint** structure containing x and y coordinates. These target coordinates are read from a **CSV file**, enabling flexible modification of mission waypoints without recompiling the code. The file is loaded using a custom CSV parser, which skips invalid entries and returns a vector of valid points.

Integration with move_base: The node uses an actionlib::SimpleActionClient to interact with the **move_base** action server. It waits for the server to become available and then begins sending goals one by one. Before dispatching the goals, the node subscribes to the **/odom** topic to obtain the robot's current position in real time, which is necessary for monitoring goal proximity.

Goal Dispatch and Monitoring Logic: Each goal is sent using a move_base_msgs::MoveBaseGoal message. The node continuously checks whether the robot is within a specified proximity (5 meters in this implementation) of the target goal, based on the Euclidean distance between the robot's current position and the target. Once the goal is considered reached, the node moves on to the next point in the list.

This method ensures that goals are not only dispatched, but also verified spatially, regardless of the internal **move_base** action result. The goal detection logic is decoupled from the action result and instead relies on continuous odometry feedback, making it robust to unexpected robot behavior.

At the end of the mission, when all goals are reached, a completion message is logged.