COSC-5207EL02: Assignment 4 Report

Group Number: 8

Group Member:

NAME	STUDENT#	EMAIL
Haoliang Sheng	0441916	hsheng@laurentian.ca
Songpu Cai	0441024	scai1@laurentian.ca

1. Introduction

This report outlines our efforts to develop an autonomous navigation and Visual SLAM system for the TurtleBot3 in a Gazebo-simulated environment. Starting with a **Basic Implementation** that utilized A* path planning and simple Visual SLAM for obstacle avoidance and environment mapping, we demonstrated the feasibility of sophisticated navigation without Lidar data. Building on this foundation, we attempted an **Advanced Version** aiming for greater modularity and precision through specialized ROS nodes. Despite facing technical challenges, such as synchronization issues and incomplete features, our work lays the groundwork for future advancements. This document details our approach, the obstacles we encountered, and outlines a path forward for enhancing autonomous navigation technologies.

For a comprehensive understanding of the implementation and its results, a YouTube video demonstration will be provided https://youtu.be/sg6w09B0gPU.

2. Basic Implementation

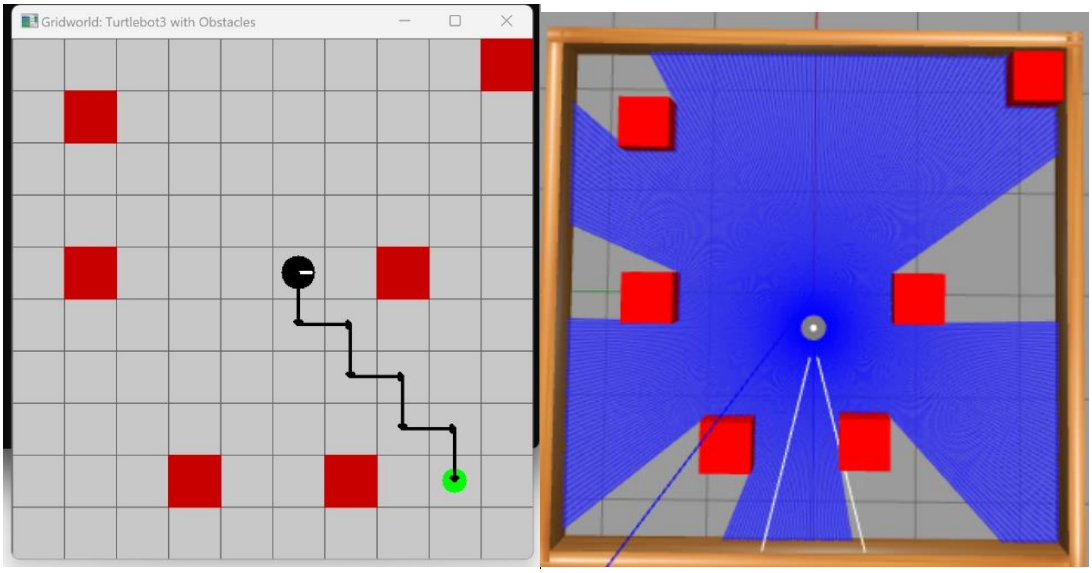
2.1 Overview

Our basic implementation enabled the TurtleBot3's autonomous navigation in a Gazebo-simulated environment using A* path planning. Central to this setup is a ROS node, **navigation_move_cam.py**, which integrates with the adapted **path_planning.py** script for dynamic obstacle avoidance. Despite challenges with Gazebo's performance and time synchronization, necessitating device-specific

adjustments, this foundation demonstrates the viability of advanced autonomous navigation and SLAM without Lidar data.

2.2 A* Path Planning Implementation

In our implementation, the A* algorithm's role is pivotal for guiding the TurtleBot3 through a Gazebo-simulated environment. Utilizing the **path_planning.py** script, the **find_shortest_path_Astar** function calculates the optimal path by considering obstacles and the goal. The subsequent **generate_path** function translates this path into a series of robot movements.

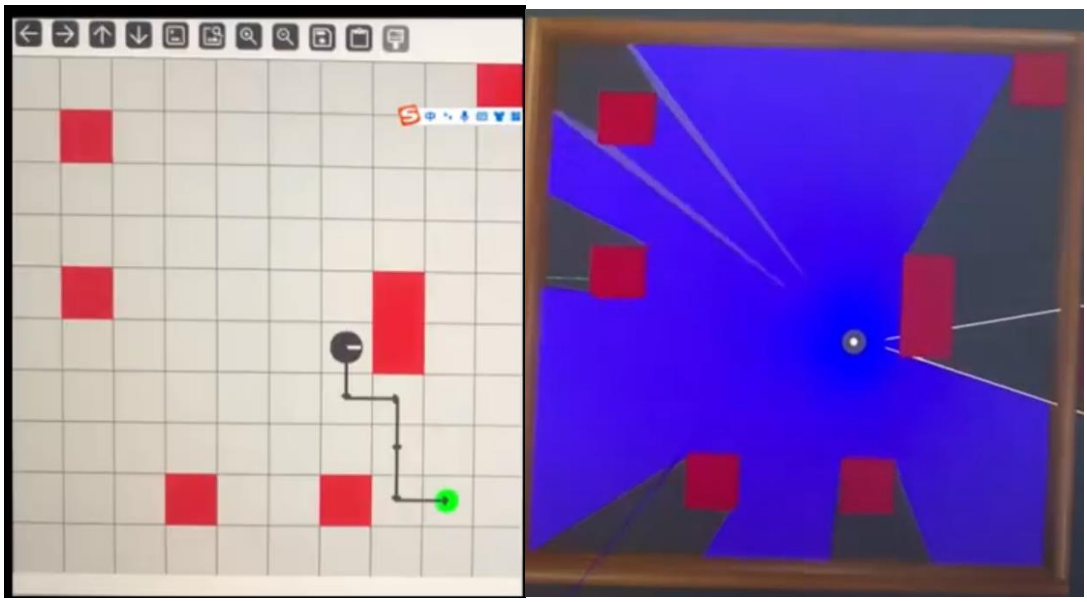


Robot motion is controlled by issuing commands to the **/cmd_vel** topic based on the action sequence generated. This direct control mechanism, however, lacks feedback from the **/odom** topic, leading to potential inaccuracies in the robot's movements due to the simulation's inherent limitations. Despite these challenges, this approach effectively demonstrates basic autonomous navigation within a simulated context.

2.3 Visual SLAM Implementation

Our Visual SLAM approach enables the TurtleBot3 to navigate and map its environment in real-time, using its camera to detect obstacles. Images from the **/camera/image_raw** topic are processed with OpenCV to identify red objects, marking potential obstacles. While this method determines the obstacle's presence and

direction, it falls short of measuring distances.



Upon obstacle detection, the map is updated via **get_new_obstacle_position**, altering the robot's understanding of its surroundings. The robot then recalculates its path using **calculate_new_path**, adapting its route to avoid these newly detected obstacles.

This implementation provides a basic yet effective framework for dynamic obstacle avoidance and path planning, despite its limitation in distance measurement, underscoring the potential for sophisticated navigation strategies in complex environments.

3. Advanced and Incomplete Version

3.1 Overview

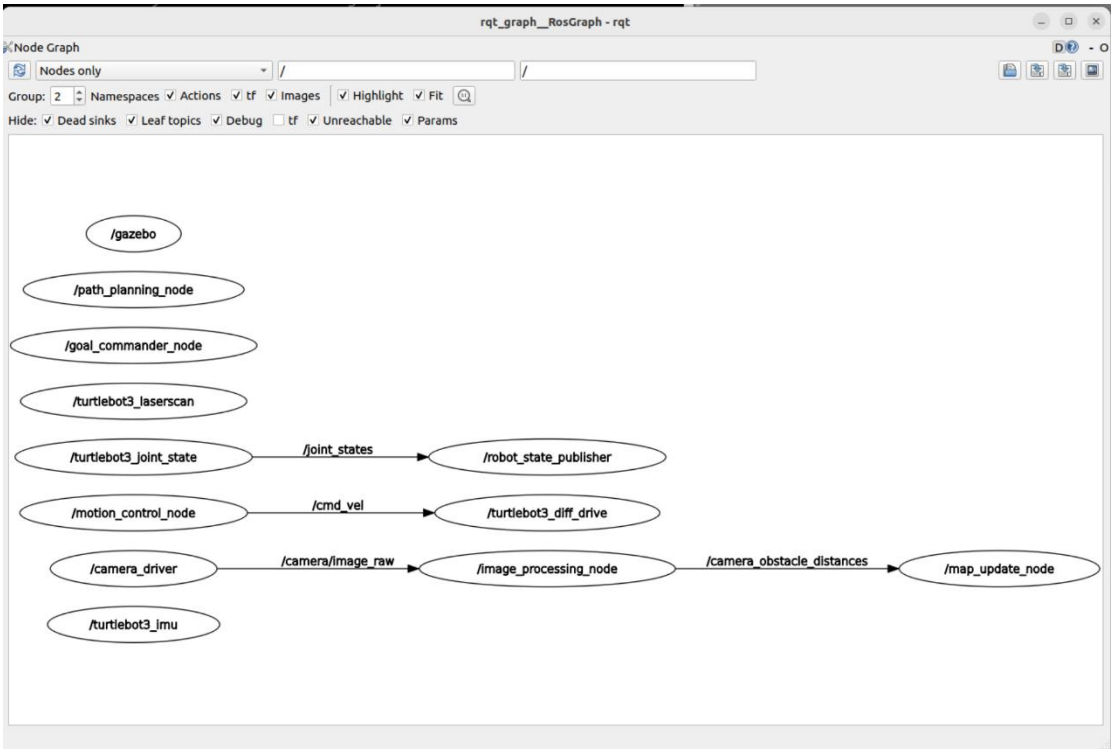
Our advanced system aimed to refine navigation and SLAM through a modular setup involving five ROS nodes: **image_processing_node**, **map_update_node**, **path_planning_node**, **goal_commander_node**, and **motion_control_node**. This structure was intended to enhance precision and functionality by dedicating specific tasks to each node.

Despite the innovative approach and the potential for improved system performance, the project remained incomplete due to time constraints. While not fully functional,

the effort towards this advanced version underscores our commitment to exploring sophisticated autonomous navigation and SLAM solutions, laying groundwork for future development.

3.2 Communication Between Nodes

In the advanced and incomplete version of our system, the communication between nodes was intricately designed to facilitate a complex yet efficient process of navigation and obstacle avoidance. Here's a simplified breakdown of each node and their interactions within the system:



Gazebo Simulation Node

- **Functionality:** Initiates and operates the custom Gazebo world, incorporating the TurtleBot3 and preset obstacles.
- **Connection:** Supplies other nodes with the simulated environment and sensor data, such as camera images **/camera/image_raw** and **/odom** pose information.

Camera Image Processing Node

- **Functionality:** Subscribes to the TurtleBot3's camera topic, employing OpenCV for image processing to detect obstacles (red cubes) and walls.

- **Connection:**
    - Receives image data from the Gazebo Simulation node via the **/camera/image_raw** topic (**Image.msg**).
    - Publishes detected obstacle directions and distances on **/camera_obstacle_distances** topic (**PoseArray.msg**), used by the Map Update Node.

Map Update Node

- **Functionality:** Updates the grid map based on information from detected obstacles. Involves TF mapping algorithms (camera_link -> map), which were not fully implemented.
- **Connection:**
- Receives obstacle direction and distance from the Camera Image Processing node via **/camera_obstacle_distances** (**PoseArray.msg**).
    - Provides obstacle locations through **/get_map_obstacles** Service (**GetMapObstacles.srv**) for use by the Goal Commander Node.

Path Planning Node

- **Functionality:** Implements the A* algorithm to generate the shortest collision-free path from the robot's current location to the goal.
- **Connection:**
    - Requests obstacle positions maintained by the Map Update Node through **/get_map_obstacles** Service.
    - Offers path planning results via **/plan_path** Service (**PlanPath.srv**) for the Goal Commander Node to process robot actions.

Goal Commander Node

- **Functionality:** Sends iterative Action drive commands to the robot based on the planned path.
- **Connection:**
    - Queries the Map Update Node for obstacle positions via **/get_map_obstacles** Service to confirm if obstacle locations have changed.

    - Requests path planning through **/plan_path** Service and initiates **/move_robot** Action requests (**MoveRobot.action**).

Motion Control Node

- **Functionality:** Executes motion commands, controlling the TurtleBot3's movement according to the instructions.
- **Connection:**
    - Receives Action commands from the Goal Commander Node through **/move_robot** (**MoveRobot.action**).
    - Controls the robot's linear and angular velocity via **/cmd_vel** topic (**Twist.msg**) for grid movement.
    - Utilizes **/odom** information (**Odometry.msg**) to adjust the robot's turning angles and movement distances, facing potential synchronization issues.

This system is vast and employs three types of communication mechanisms: **Topics**, **Services**, and **Actions**. Initially, we underestimated the complexity of the system, resulting in some functionalities not being successfully implemented, such as TF coordinate transformations and unresolved issues with **/odom** blocking.

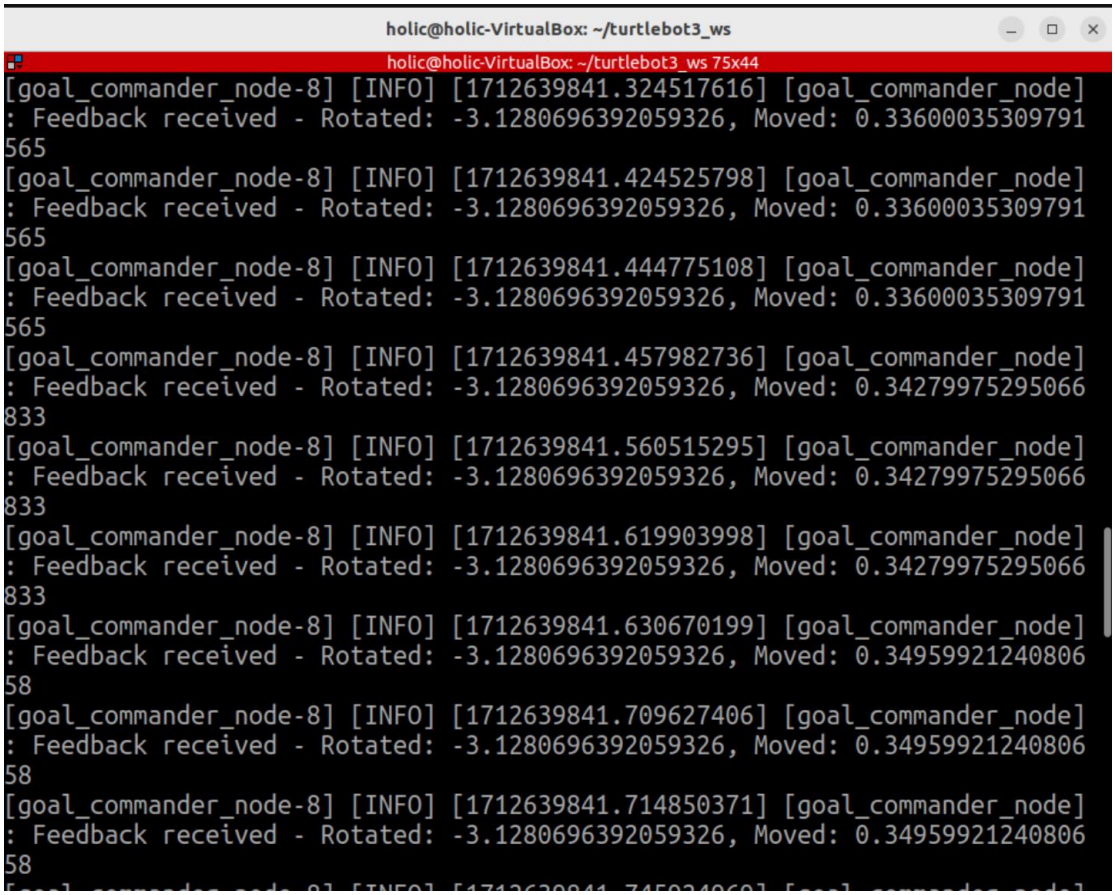
3.3 A* Path Planning Implementation

In the advanced version, we aimed to enhance the A* path planning's flexibility by introducing modular processing through the Goal Commander Node (Action Client) and the Motion Control Node (Action Server). This structure enabled real-time action tracking with feedback for prolonged actions, a leap towards a more adaptable navigation system.

Modular Design

- **Goal Commander Node:** Acts as an Action Client, issuing navigational goals based on A* planned paths, and dynamically adjusts these based on feedback.
- **Motion Control Node:** Serves as an Action Server, executing movement

commands from the Goal Commander Node, with /odom feedback aiming to refine the robot's trajectory.



Blocking Challenge

A critical hurdle was the blocking issue with **/odom** updates within a loop function, preventing real-time adjustments in the robot's navigation. This challenge limited the effectiveness of our advanced modular system, underscoring the complexities of achieving fully responsive autonomous navigation.

3.4 Advanced Visual SLAM Implementation

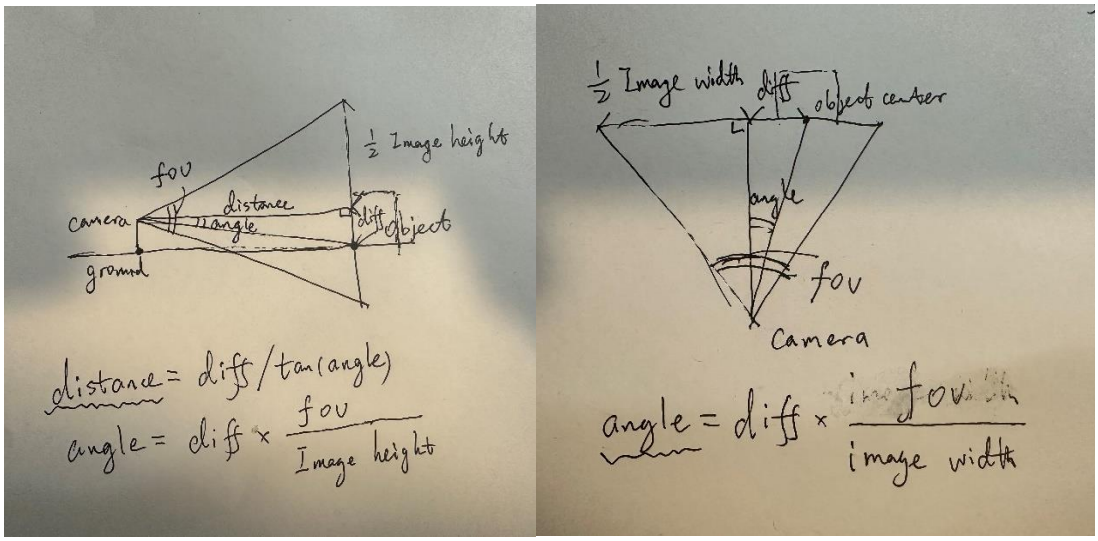
In the advanced Visual SLAM implementation, we aimed to refine obstacle detection and distance estimation using OpenCV. This method involved identifying red obstacles, performing binary thresholding, and employing geometric calculations to estimate distances and angles from the camera to obstacles.

Obstacle Detection and Distance Estimation

- **Red Obstacle Detection:** We utilized OpenCV to detect red obstacles within

the camera's field of view, applying binary thresholding to isolate these objects.

- **Bounding Rectangle and Midpoint Calculation:** For each detected obstacle, we calculated a bounding rectangle, focusing on the bottom midpoint as a key reference for distance estimation.
- **Distance and Angle Calculation:** By leveraging the camera's field of view (FOV) and applying trigonometric functions, we estimated the distance and angle from the camera to the obstacle's base. This estimation process is visually represented in the diagram below:



Challenges:

The intended final step was to transform these estimates from the **camera_link** frame to the **map** frame for accurate mapping, using TF coordinate transformations. Unfortunately, due to time constraints, this crucial transformation process was not implemented, impacting the system's ability to update the global map with precise obstacle locations.

3.5 Future Work

To enhance our autonomous navigation and Visual SLAM system, we've identified key areas for future development:

- **Resolve /odom Blocking:** Improving real-time position and orientation updates by fixing the **/odom** topic's blocking issue will enable smoother

navigation and dynamic path adjustments.

- **Implement TF Coordinate Transformation:** Completing TF coordinate transformations will accurately localize obstacles in the global map, improving navigation and obstacle avoidance.
- **Apply Advanced Filtering Techniques:** Using Gaussian or particle filters to update the map will increase accuracy by reducing sensor noise and uncertainty.

Addressing these challenges will significantly improve the system's performance, paving the way for more reliable and adaptable autonomous navigation solutions.