# PAR
# Search Navigation
# Challenge Report

Rafat Mahiuddin (s3897093)

Adhiraj Jain

Lab Demonstration

# 1. Package dependencies & configuration

## 1.1. ROS Packages

The following packages are marked as a dependency in our package:

- **rospy**
  Rospy is a Python library for ROS that allows developers to use Python code to create ROS nodes and execute ROS operations, offering a Python interface to ROS functionality for building Python packages that interface with ROS.
- **husarion_ros**
  Includes drivers necessary for integrating Husarion robots with the ROS ecosystem and provides a set of ROS nodes for interacting with different Husarion platform components, including sensors, motors, and peripherals, via ROS communication protocols.
- **rosbot_description**
  Provides all of the details about the RosBot robot model, such as physical dimensions, joint settings, and sensor data.
- **std_msgs**
  This package is used for interprocess communication between nodes in ROS. It offers a collection of common message types used to convey data between ROS nodes, such as integers, floating point numbers, booleans, and strings.
- **move_base_msgs**
  Offers messages to direct robot movements by setting target coordinates and includes navigation-related data, such as the rosbot's location, orientation, and target destination.
- **actionlib_msgs**
  Offers messages for controlling and creating actions, for e.g., explorer and controller nodes can communicate with each other using goal, feedback, and result elements.

## 1.2. Libraries

Our package utilised a number of libraries for movement and vision recognition. For example, we used the GMapping in order to initiate the SLAM process for our RosBot. This GMapping used custom tuned parameters for the RosBot, which was configured in the route_admin_panel. We also used move_base as part of our navigation stack, alongside our custom parameters for the trajectory planning and the local and global costmaps. find_object_2D was used for recognition of hazard signs.

### 1.3. Launch files

The controller.launch file is the primary launch file used to start our challenge. This file includes references to other launch files that are required in completing the challenge. The first of these launch files is rosbot_pro.launch which launches the RosBot drivers. Then there is navigation.launch, which initiates our navigation stack using gmapping and a custom tuned move_base. The path tracking and exploration nodes are launched afterwards. Finally, the vision node launches our hazard detection algorithm alongside find_object_2d.

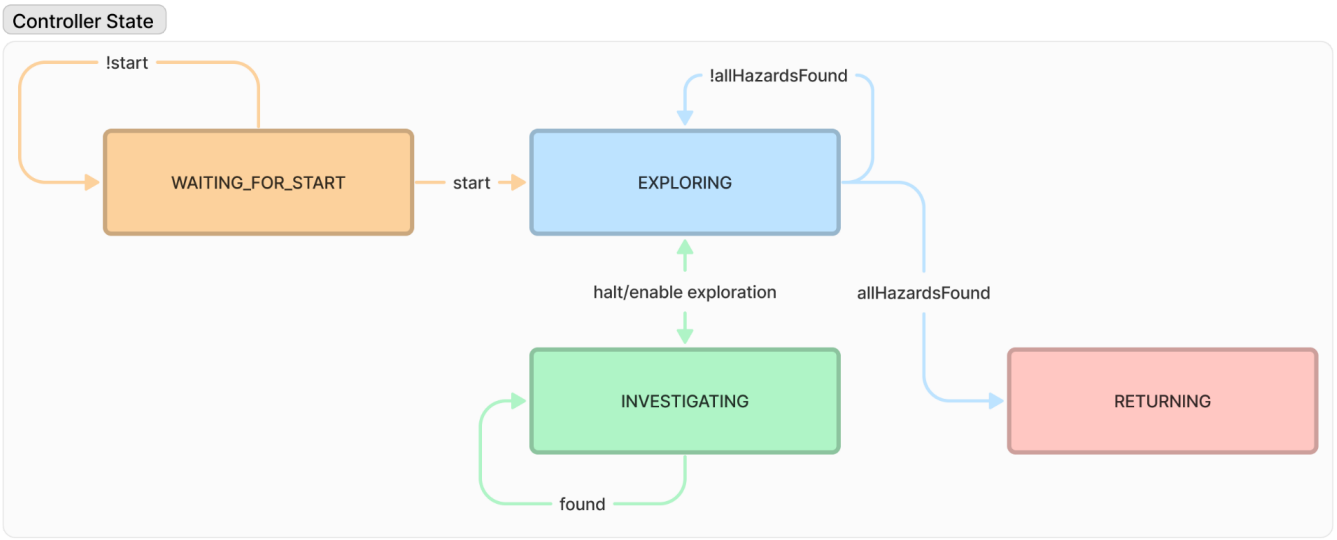
### 1.4. Custom configuration and training models

Extensive customization was made to the costmap_params and trajectory_planner yaml files located under ./config. An in-depth analysis will be performed in the third section of this report. Our training models are located under ./hazards and included a combination of straight and angled images of the hazard signs.

# 2. ROS nodes

### 2.1. Controller

The control node was tasked with controlling the state of the machine. More specifically, the states that the program may be in are the waiting_for_start, investigating, exploring and returning states. The diagram below illustrates the relationships between the states within our program.



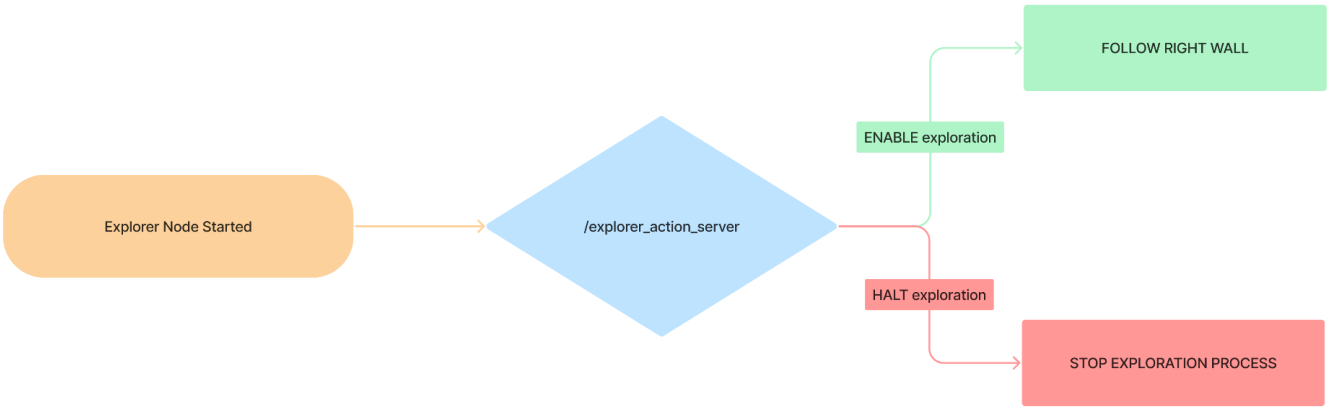

When the controller switches state, the controller will always execute an exit strategy for the state before moving to the next state. For example, if the state switch is exploring → returning, then the controller will run the exit strategy for the explorer node (which is to halt the explorer) before executing the state strategy for the returning state [Adapted from 11].

The controller also implements a return_to_starting_position method which gets the initial position of the robot's path and moves the RosBot to that coordinate in the correct orientation. Before this method is called, we utilise GMapping to incrementally build up the map environment during the exploration stages. Afterwards, the robot will evaluate and repair the map if necessary using the AMCL localisation strategy [10]. A global path is then created using Navfn to the start node, which computes a global path using the Djikstra's algorithm [5, 7, 9]. We then use TrajectoryPlanner as the local path planner, which takes care of navigation in the robot's immediate environment [6].

As seen in class, our localisation algorithm becomes more confident when the robot begins to move, as it can take advantage of historical environmental data to deduce the robot's location on the map [8, 10].

## 2.2. Explorer

The explorer node was assigned the task of navigating a maze using the bug algorithm, which involves following the right wall. The idea of using this algorithm came during one of the weekly course contents which is navigation, it was discussed as one of the simplest approaches to avoid obstacles and explore an unknown environment. The first step involved extracting data from the LIDAR scan, which provided distance information for all 360 degrees around the RosBot. The data was then divided into segments, including the front, front-right, front-left, right, and left, based on specific angles. A maximum wall distance threshold was also set to ensure that the RosBot stayed close to the wall once it was detected. The node also contains predefined velocity commands for different tasks such as, finding the wall, turning left, moving diagonally, or following the wall [1]. The algorithm covers eight possible occurring scenarios for RosBot in an unknown maze. The explorer node listens to the controller node using Actions server-client method, using this the controller node provides a goal to the explorer node which contains the state for exploration as either ENABLE or HALT, and based on the state the explorer node either halts the exploration or continue exploring by analysing the current scenario and sending velocity commands to the RosBot [2].




## 2.3. Tracker

The tracker node is responsible for tracking the robot's path during the navigation challenge and publishing it to the /path topic. To avoid unnecessary computation, the tracker only updates the path when the robot location has changed.

To derive the path, a kinematic transformation is performed from the robot's base frame to the map frame. We use the frame identifier to ensure synchronisation between the frames [4]. After the pose transformation, we derive the x, y and z coordinates alongside the yaw orientation using the euler_from_quarternion function [12].

If the pose has changed, the new pose is added to the path model and the updated path is published to /path.

### 2.4. Vision

The vision node is responsible for detecting hazards, detecting the location of the hazard and publishing a visualisation marker to the /hazards topic.

The vision node subscribes to the /objects topic which publishes any objects detected by find_object_2d [3]. On callback, vision extracts and interprets the object message with the appropriate hazard label.

The callback also initiates a get_image_location method to get an estimated location of the hazard. To get the location, we find the minimum laser range in the front 60 degrees of the robot. We then derive the angle, get the x and y coordinates of the shortest laser in the laser's coordinate frame and perform a kinematic transformation from the laser to the robot's base position. We then perform another kinematic transformation from the robot's base location to the map [4, 12].

# 3. Analysis and Evaluation

### 3.1. Controller (move_base)

Upon initial experimentation, the move_base package was very poorly tuned. The algorithm will halt within a metre from the goal, spin around needlessly, move and turn very slowly, and will crash into walls on tight turns. The global path it produced was less than optimal as they were far too tight, causing the local planner to become confused and nervously spin around for alternative directions. In addition, the local path planner seemed to be far too conservative, to a point where it will get stuck in a higher costmap it got into after naively following the global planner.

Our first tune was to ensure the robot was going to the correct goal position. I've done this by lowering the goal tolerance to 0.1 for both yaw and xy distances. I have also dramatically loosened the inflation scale factor of obstacles in the global costmap to 15 while tightening the local costmap to 27.

This forced the global costmap to have much larger inflated obstacles over the local map. As a result, the global path was smooth and consistent while being safe enough for the robot to navigate quickly without over/under steering. As the local costmap was far tighter then the global, the robot had enough freedom to navigate around tight turns without getting stuck.
However, this introduced the possibility of the robot straying too far away from the global path, causing the robot to make sudden adjustments to re-adjust to the path. This was resolved by favouring the global path over the local goal through the path/goal_distance_bias parameters.

Given the very lenient local costmap, there were cases where the robot would almost carelessly scrape the wall. To limit this, the maximum velocity was decreased, the sim_time adjusted (so the robot has enough time to evaluate changes and make decisions) and the occdist_scale was increased to 0.09. This made the robot become more fearful of obstacles, and the RosBot will immediately readjust if it detects the costmap value is too high.

Fine-tuning greatly increased the performance of move-base, both in the simulation and the real world. Move base became accurate, quick and precise. However, move_base required quality maps to operate, which was challenging to provide given our explorer algorithm corrupting the

map on harder turns and sudden stops. However, this is a limitation of the explorer and not move_base.

### 3.2. Vision (Hazard marking)

For our vision, we faced an issue where hazards were not being detected at an angle. To resolve this, we trained our image model on both straight and angled images, which greatly improved recognition accuracy.

In terms of marking hazards, our initial algorithm involved taking the pose of the closest laser as the hazard's location. This worked surprisingly well as our robot will only recognise objects when it is very close to the hazards. However, as our vision training improved, our robot began to detect objects from a further distance, and began marking hazards on a wall perpendicular to the real location. While we considered using 3D recognition packages (such as find_object_3d), we figured that the added computation/streaming of the depth camera will significantly limit the rate at which we can capture and analyse frames. In combination with how well our initial algorithm worked, we decided on improving our existing solution. To do this, we cropped laser data to the front 60 degrees of the robot, which was the field of view of the camera.

Although far more accurate algorithms exist for depth detection, for our use case, we have deemed this to be accurate enough. Our algorithm will always accurately place a marker on the wall, and by only looking at the front laser, the marker will almost always be very close to the hazard sign.

As mentioned previously, one may improve this by using find_object_3D which creates a tf point of where it thinks the object is on the map. We can also calculate the homography matrix of the returned object, find the relative degree of the detected object, and then create a corresponding laser mapping to perform the same kinematic transformations to find the hazard coordinates. If needed, both of these methods will map the centre point of the detected object more accurately then our current implementation.

### 3.3. Explorer (wall-follower algorithm)

During various tests, few challenges were faced with the RosBot's movements. In real life, the robot will sometimes jerk suddenly when halting or taking a turn. Although this was not present in the simulator, this sudden jerk lurched the robot forward and deviated the map in the direction of the lurch. This is due to how the RosBot determines its position and orientation. Accordingly, the RosBot relied on cameras, lidar, and odometry, to create a map of the environment. However, the sudden jerking motions caused the sensors to register irregular movements, resulting in inaccuracies in the RosBot's estimated location and orientation. As the RosBot continued to move, these inaccuracies accumulated, leading to warped or corrupted maps. Moreover, inaccurate localization also made it difficult for the RosBot to return to its initial position. Several solutions were discussed and tried, but a simple one was selected which was to reduce the linear speed while following the wall and giving a little linear velocity while taking a turn, which helped to reduce the jerking motion significantly.

During various tests and the demonstration, another problem was discovered where the RosBot was getting too close to the wall, which resulted in collisions when taking turns or moving diagonally. To address this issue, the distance between the RosBot and the wall has been increased. By doing so, the RosBot was able to move with a safe distance from the wall, thus avoiding collisions.

During the initial experimentation phase, it was noticed that when the RosBot was following the wall in a straight line, its camera was unable to detect a hazard sign on the wall. To address this issue, an angular velocity was added to the movement so that the robot would always face and move towards the right wall (i.e., the chosen wall). By doing so, the camera could easily detect any hazard signs that were hung on the wall.

The demonstration revealed a drawback where the rosbot was unable to detect a hazard attached to an obstacle in the middle of the maze, which was not connected to the wall. To address this issue, one possible solution is to use Active SLAM, a technique in robotics that allows for simultaneous mapping of an unknown environment and determination of the robot's location within it. The robot collects sensor data as it moves through the area and employs it to construct a map of the surroundings. In Active SLAM, the robot actively selects which actions to take to enhance the quality of the map and localization, such as visiting a specific location or gathering new sensor data. This method of active exploration enables the robot to create an accurate map of the environment and locate itself effectively.

# Appendix

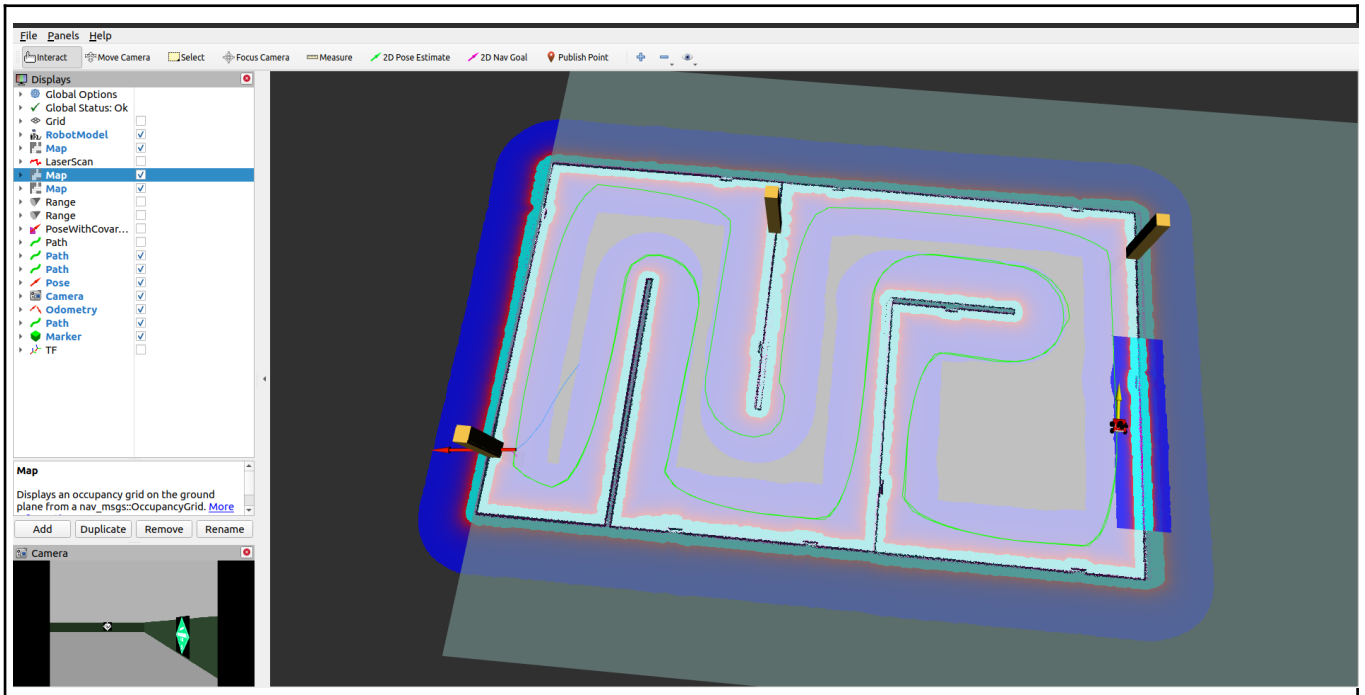



**Figure 1**. Rviz screenshot taken during a test run before demonstration day.

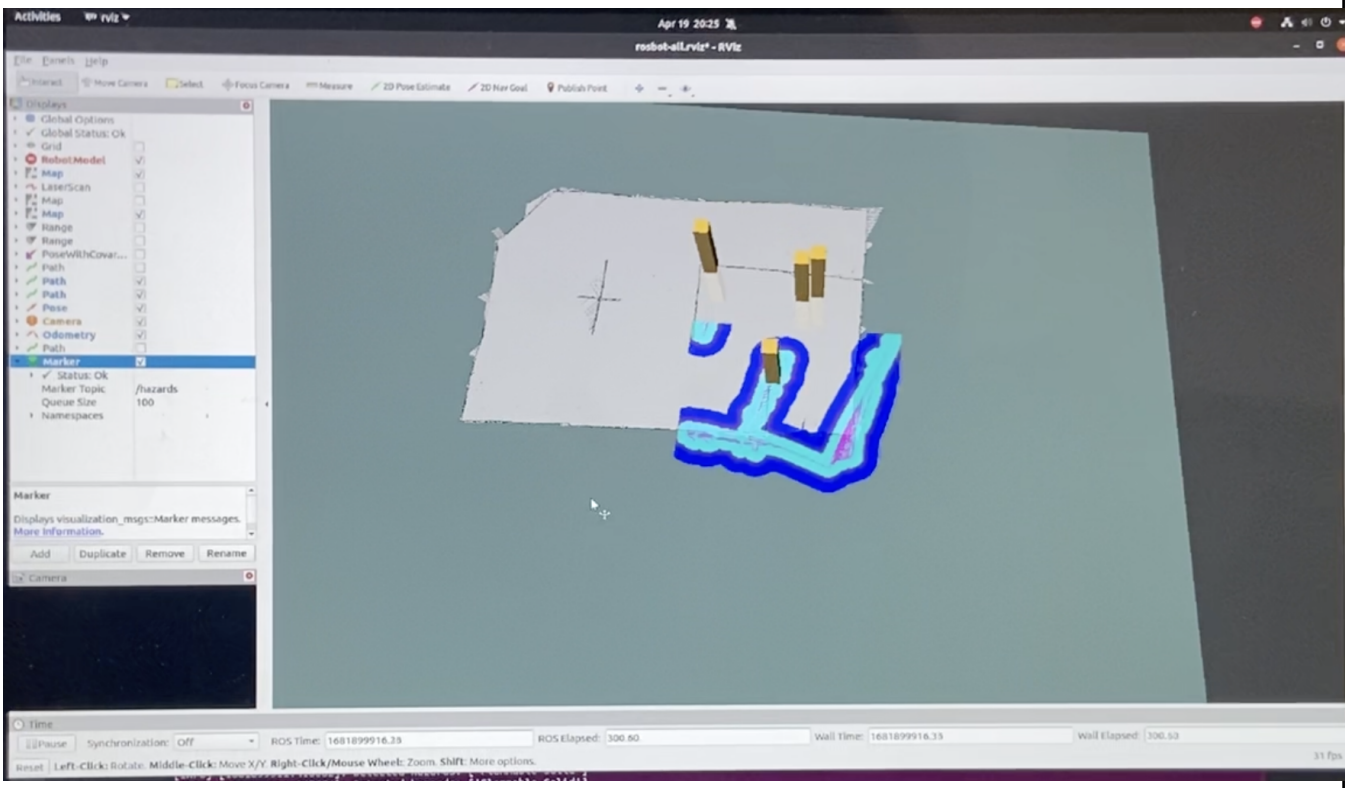



**Figure 2**. Rviz screenshot taken during a test run in the lab

# References


[1] Arruda, M. (2019, 05 23). *Exploring ROS with a 2 wheeled robot #7 – Wall Follower Algorithm*. Retrieved from The Construct: https://www.theconstructsim.com/wall-follower-algorithm/

[2] Nimbekar, N. (2020). *Wall-follower-in-ROS-using-Python*. Retrieved from github: https://github.com/nimbekarnd/Wall-follower-in-ROS-using-Python/blob/main/src/motion_plan/nodes/follow_wall.py

[3] MathieuLabbe. (2016, 09 21). *find_object_2d*. Retrieved from ROS.org: http://wiki.ros.org/find_object_2d

[4] Howie Choset, H. L. (2019). An Introduction to Robot Kinematics. *The ROBOTICS INSTITUTE.*

[5] NickLamprianidis. (2018). *costmap_2d*. Retrieved from ROS.org: http://wiki.ros.org/costmap_2d

[6] NicolasVaras. (2019). *base_local_planner*. Retrieved from ROS.org: http://wiki.ros.org/base_local_planner?distro=noetic

[7] TristanSchwörer. (2021). *global_planner*. Retrieved from ROS.org: http://wiki.ros.org/global_planner?distro=noetic

[8] nickfragale. (2020). *move_base*. Retrieved from ROS.org: http://wiki.ros.org/move_base?distro=noetic

[9] jihoonl. (2014). *navfn*. Retrieved from ROS.org: http://wiki.ros.org/navfn?distro=noetic

[10] AV. (2020). *amcl*. Retrieved from ROS.org: http://wiki.ros.org/amcl?distro=noetic

[11] refactoring.guru. (n.d.). *strategy*. Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/strategy

[12] jarvisschultz. (2017). *tf*. Retrieved from ROS.org: http://wiki.ros.org/tf