Exploration of Unknown Map with ROS2 and Nav2

METR4202 Interim Assessment

Aiden Ortiz, 46936367
Genevieve Nissen, 48035198
Blaise Delforce, 48021191
Harveer Birk, 48105130

1.0 Introduction

Autonomous robotics pose many dynamic and complex challenges. Localisation, navigation and mapping become all the more challenging in a completely unknown environment. This project's goal is to have a TurtleBot autonomously explore and map unknown territory and visually identify designated targets by leveraging existing robotics libraries and simulation tools.

In its current state, the TurtleBot is able to navigate an unknown area and build a map of its surroundings in simulation. It's able to interpret sensor data and map its environment, then use said data to plan movements. It makes use of Robot OS 2 (ROS2), Simultaneous Localisation and Mapping (SLAM) Toolbox, and Navigation 2 (Nav2), which are well known libraries for robotics automation. It also uses Gazebo and RViz for simulation and visualisation environments.

2.0 Design Goals

In short, the robot must be able to explore an unknown environment using sensor data to navigate, process the collected sensor data and store it as a map. During this process, it must identify designated AruCo markers, scanning them and marking their location on the map. The run will be considered successful when an AruCo marker is found and marked accurately.

A functionality-specific goal breakdown is focused on implementing exploration with mapping. Identified sub-goals are listed below.

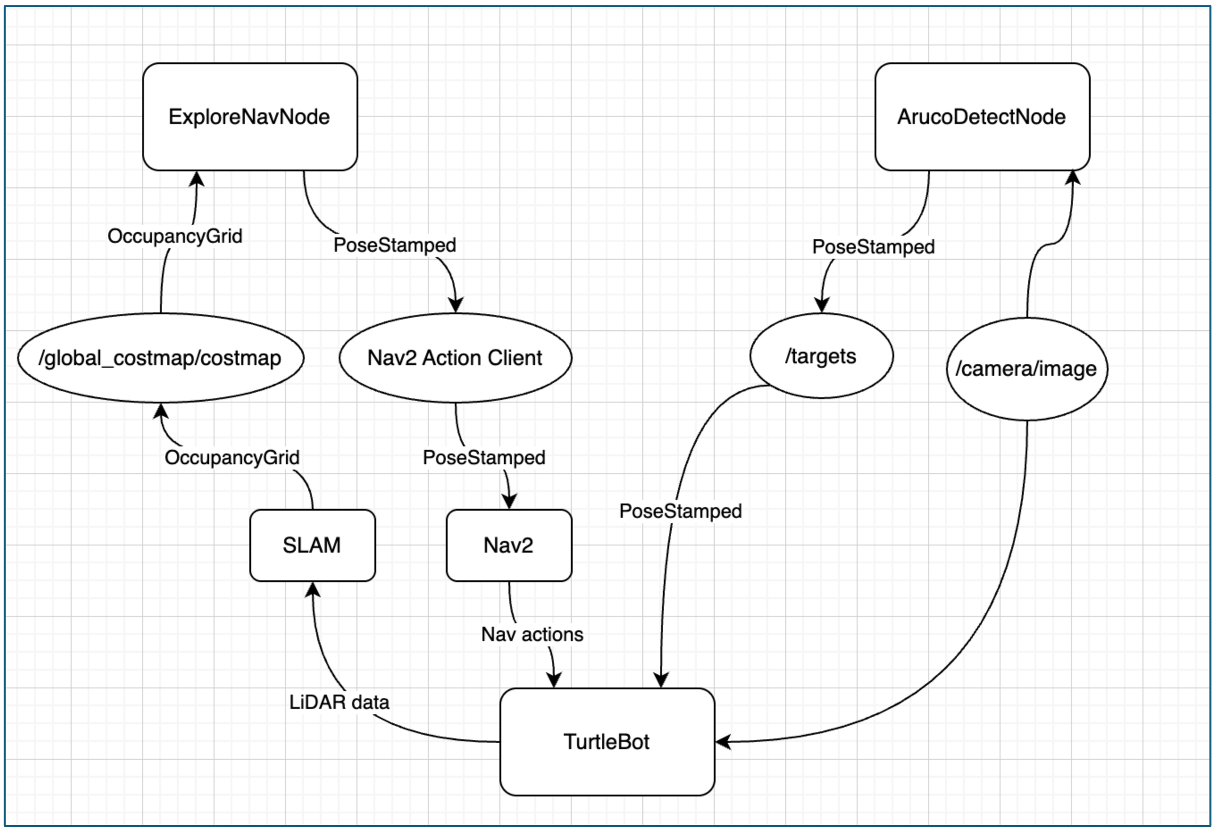
1. Before Navigation, use an algorithm to identify unexplored frontiers.
2. Before Navigation, have a developed frontier selecting algorithm.
3. Navigate to a specified waypoint from an arbitrary starting position.
4. During Navigation, use sensor data to avoid obstacles.
5. During Navigation, use LiDAR data to build a map of environment.
6. During Navigation, use Camera to identify Aruco Markers in the testing environment
7. After Navigation, publish a completed map with Aruco Marker position identified.

3.0 System Architecture

3.1 Node Structure

The system will use a node structure, with each node controlling an aspect of robot behaviour and communicating to other nodes. These nodes are built over the ROS2 communication protocol, which uses a publisher-subscriber system. Nodes either publish or subscribe to topics, and can communicate instructions, status updates, or sensor data via different message types.

An exploration and a perception node will both be implemented. The exploration node is responsible for all exploration and decision-making functionality. It will interpret map data, identify frontiers and select the most desirable frontier to explore. The perception node will use existing computer vision libraries to identify and scan Aruco markers.



3.2 TurtleBot 3

The existing TurtleBot system is equipped with a LiDAR sensor and camera. Its movement capabilities include forward and backward driving as well as on the spot rotation. Operation of these motors and sensors is handled by existing libraries.

3.3 Exploration & Navigation Module

The mapping functionality and navigation with waypoints is implemented using the SLAM Toolbox, and existing localisation and mapping library. Frontier identification and selection is a custom written python module. This approach allows optimisation by tweaking the frontier selection algorithm and weighting properties such as proximity, area and nearby obstacles.

The **`ExploreNav`** node has a single-subscriber-single-publisher pipeline, with message data objects being used for communication. It uses the **`OccupancyGrid`** object from the **`/global_costmap/costmap`** topic which is processed to identify frontiers algorithmically and rank them by exploration benefit. The most up to date map is regularly published to this topic by the existing **`SLAM`** module. Once the goal frontier has been decided, a Nav2 action client is used to communicate with the TurtleBot.

4.0 Implementation

4.1 Exploration

The exploration algorithm functions by identifying frontiers, clustering them to a single point, and ranking the value of exploring each point. Setting a waypoint goal is done asynchronously from the map updates, to ensure the robot can reevaluate its current goal as more information becomes available.

4.1.1 Frontier Detection

In a costmap **`OccupancyGrid`**, cells can be either unknown (-1) or known, with a cost from free space (0) to lethal (100). The **`find_frontiers()`** function scans the occupancy grid to locate the boundaries between unknown cells and cells with a reasonable cost. A threshold of 60 was used to allow the robot to traverse narrow corridors with no freespace, whilst maintaining frontier distance from lethal cells. Using this threshold ensures goals are not placed in unreachable areas.

Step	Description	Justification
1. Prepare Data	Check map availability, convert to 2D numpy array from 1d costmap data	Allows easy parsing to determine the x and y coordinate swiftly
2. Create Masks	Create 2 binary masks (1. Unknown cell mask [U] with unknown cells being –1. And 2. Free cell mask [F] with values between 0 and 60).	Allows separation of the data for calculation, as the F mask will be used for the convolution.
3. Define Kernel	Create a kernel (K), which is a matrix of 3x3 representing the 8 connected cells around a cell (init with all 1s). This will be used later	Used for calculations to find frontier cells with convolution.
4. Count free neighbours	Convolve the free mask (F) with the kernel (K). The result of convolving (F, K)	Allows us to see the neighbourhood of each cell.

	is a map with each cell essentially containing the count of adjacent free cells (blurs together),	
5. Identity Frontiers	The frontier mask (M) is therefore the logical AND of the unknown mask with the convolved mask (with condition that the cell has >1 free neighbour)	Combining the new convolution result with the prior unknown mask gives us a way of seeing the frontiers (and filtering everything else out, e.g. unknowns not near free space)
6. Create Frontier Map	A frontier map is created where cells in M are set to a high value and all others 0.	This map is used by our subsequent algorithms for goal calculation.
7. Return map	Return valid clusters map	

These steps are found in `_find_frontiers.`

4.1.2 Frontier Clustering

Frontiers are clustered using a grouping algorithm called Breadth-First Search (BFS) to gather these unknown cells into clusters, making them easier to manage as exploration targets. In short, BFS ensures that only unknown regions that are fully spatially connected are grouped, so that the system essentially has an accurate definition/measure of the frontiers available to it.

Part of the clustering algorithm includes modifying the points representative of frontiers depending on robot position. When frontier waypoints are selected close to and in front of the robot, they are extended into unknown space to avoid becoming stuck in long corridors, where LiDAR data is difficult to obtain due to the lack of obstacles. It also filters out smaller clusters (below minimum threshold; currently sitting at 3 cells) to ensure noise/uncertainty is reduced.

Step	Description	Justification
1. Cluster Identification	Use Breadth First Search Algorithm to find connected clusters	Quick algorithm to find connected cells
2. Size filtering	Use size filter to disregard small noisy/insignificant clusters	Ensures noise is filtered
3. Selection/sorting	Sort the remaining clusters, keeping only top 5-10 clusters.	Ensures prioritisation of best exploration goals
4. Centroid calc	Calculate centre of the cluster	Ensures we are trying to go to location that has most frontiers close to it (centre)

5. Map-World conversion	Convert cell indices to world coordinates (using map resolution/origin)	Allows us to see where the actual centre is on the map
6. Point Extension	Extend the point away from the robot into space along a vector including robot pose and point	Allows more efficient exploration since reaching an extended goal means more exploration in less goals.
7. Point Storage	Store world coordinate	Publishes point

4.1.3 Goal Selection and Ranking Strategy (Heuristic)

Goal Retraction

Once clusters are formed, the **`set_goal()`** function evaluates them and chooses the most suitable one for the robot to explore. This decision is based on proximity, direction, and size, with preferrable goals being closer, in front of the robot, and being surrounded by unknown cells. Goals are set asynchronously from map updates, so that the robot is always exploring the most optimal goal. To prevent the robot from deliberating between far away points, a high weighting on was granted for goals in the direction the robot was facing, promoting fast and thorough exploration. The waypoint dispatch process is managed through Nav2 Action Client which serves as the interface between the interface between exploration node and the ROS 2 Navigation stack. Once a valid exploration target is determined, the node verifies that the Nav2 action server is ready before dispatching the navigation request.

Frontier Ranking Score

Score for a candidate goal:

= *Heuristic Score*

$$S = W_s x + W_w y + W_\theta \theta$$

- x = number of unknowns (count) within vicinity
- Y = distance from cluster to robot
- θ= angle bias (cosine of angle between robot and cluster vector)

Weight	Value	Justification
W_S	3	High weighting for clusters with high unknown cell count.
W_w	-2	Negative value favours nearby waypoints. Low est value to ensure depth of exploration.
W_Theta	5	Highest weighting promotes efficient exploration with little doubling back.

Select & Rank Goal Algorithm set_goal`()`

Step	**Description**	**Justification**
1- Init	Init empty scored list	
2- Iterate Points	For each frontier point, perform steps 3-7. Then skip to step 8.	
3- Value Score calc	Define local neighbourhood around point using a VALUE_RADIUS and count num. Of unknown cells within neighbourhood vicinity. This is potential for new exploration gain value.	Ensure maximal exploration value
4- Distance Score	Calculate 3D distance from robot current pose to frontier pose (travel cost)	Ensure minimal travel cost
5- Angular Bias Score	Calculate angular difference between robot's current yaw and vector to frontier pose (starting at current pose). Use cosine function to give maximal score when 0 degrees difference.	Ensure minimal heading change
6- Score Aggregation	Score goals using Equation (combining all scores with weight coefficients)	Ensure prioritisation performed
7- Selection	List all goals in descending order of value.	Allow a lineup to have backup goals upon failure
8- Return Best	Return the frontier point with highest score.	

4.2 Aruco Perception Node

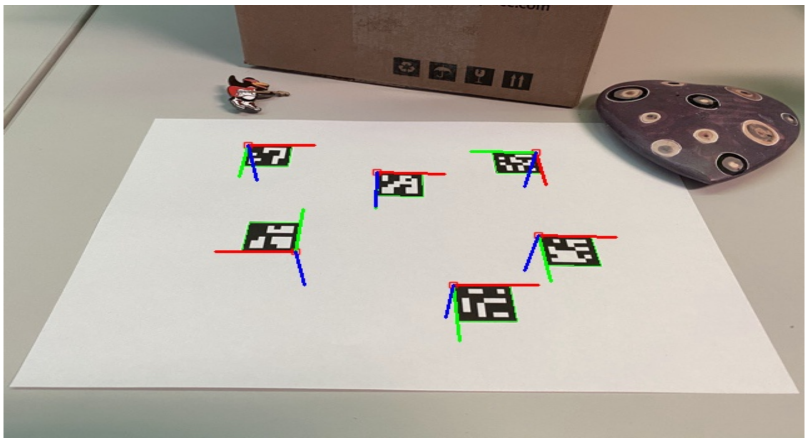
A custom Aruco Node has been developed for the purpose of identifying, locasting and displaying the pose of randomly located Aruco Markers from the testing environment, onto the Rviz display.

4.2.1 Initialisation and Calibration

The Aruco Node initially subscribes to the /camera/image_raw image and /camera/camera_info calibration data, and publishes the detected marker poses via a pose array to the /targets topic. The 6x6 dictionary and 100mm side length. is hardcoded in as per requirements. The intrinsic matrix and distortion coefficients are stored from the camera data for accurate pose estimation via proper camera calibration.

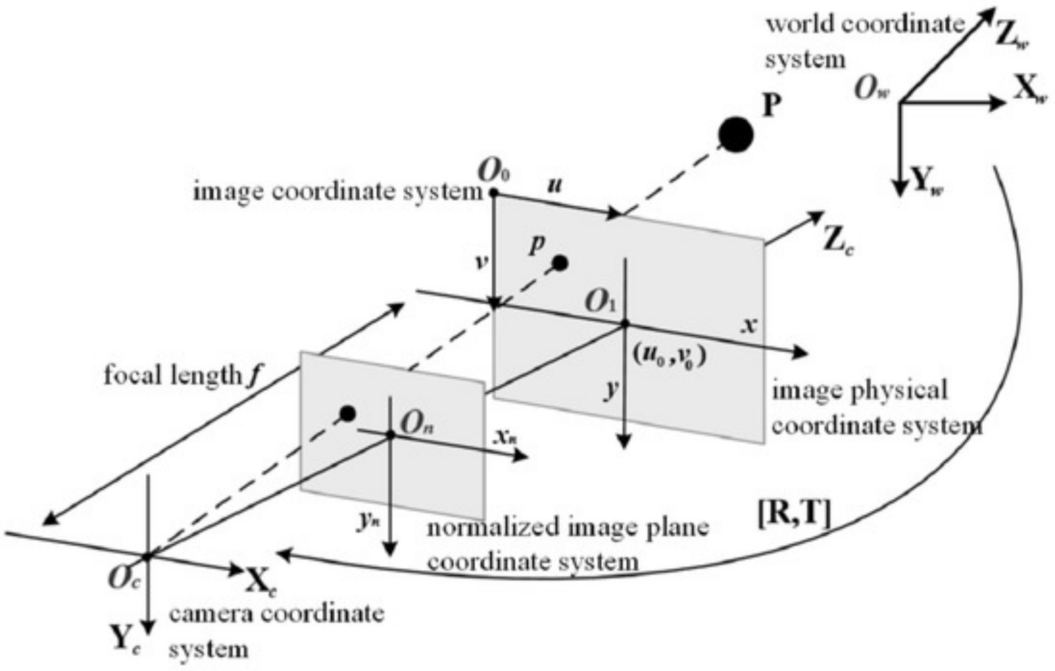
4.2.2 Image Processing

For each image, it is converted to an OpenCV type image (bgr8) and it is grey scaled, before the cv2.aruco.detectMarkers() with a predefined dictionary to find the corners/ID's. This gives us an initial coordinate to work with, as well as the number of the Aruco Marker to check accurate parsing of the pattern.



4.2.3 Pose Estimation & Transforms

The SolvePnP() function from the cv2 library is used to turn the corner coordinate of the marker into a 3D position (with a translational and rotational vector) relative to the RGB camera's frame. The Perspective-n-Point Algorithm (PnP) essentially allows the estimation of the camera's pose relative to a known 3D object (The Aruco marker).



4.2.4 Tracking/Smoothing and Publishing

The last N (SMOOTHING_WINDOW_SIZE) poses is tracked for each unique Aruco Marker, and detections older than the MAX_TRACK_AGE_SECONDS are automatically pruned from the queue.

Markers that are tracked with enough observations (MIN_HITS_FOR_PUBLISH) are published in the pose array – This pose is the moving average of the XYZ position and circular average of the orientation (yaw) over the tracking history.

This process aims to smooth the publishing to prevent large error/noise and prioritise stability of confirmed target location reporting. This is published to Rviz2 as a visualisation MarkerArray, alongside the PoseArray.

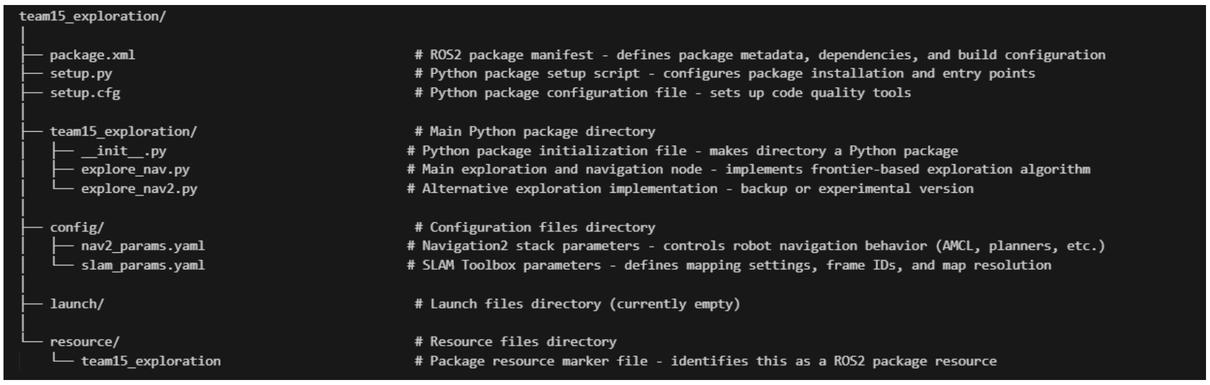
4.3 Integrated Libraries

To demonstrate what prebuilt libraries were sourced and used, the table below describes their usage in the system

Library	Description	Implementation
ROS2	Core operating system	Defines the node graph and message passing between the nodes
SLAM Toolbox	Set of ROS2 Packages providing simultaneous localisation (where is it) and mapping (what's around it)	Explore uses the \map topic (occupancy grid from slam)
NAV2	Provides global and local path planning, obstacle avoidance and control execution	The MapsToPose Action client is used to send the frontier goals/track results
TF2	Handling, Buffering and looking up coordinate frame transformations	Look up robots current pose (relative to the base footprint frame to the map frame)
RCLPY	Python client for ros2 (timers, nodes etc)	Meta managing of node class and operation
Numpy	Numerical computing for large arrays	Used to convert data 1->2D

Math	Math functions	Distance and trig calculations e.g. quaternion orientation
Deque	Double ended queue (efficient appending and pop)	Implement blacklist efficiently
Geometry Msg	Geometric primitives (poses, poses and quaternions) - A ros2 message type	Define poseStamped msg for waypoints and Point msg for blacklist coordinates
STD_SRVS	Simple msg request/response pattern with no fields and a success Boolean in response	Manual trigger for exploration cycle
SciPy	Convolution Algorithm	For convolving frontier algorithm

4.4 File Structure



```
team15_exploration/
│
├── package.xml                        # ROS2 package manifest - defines package metadata, dependencies, and build configuration
├── setup.py                           # Python package setup script - configures package installation and entry points
├── setup.cfg                          # Python package configuration file - sets up code quality tools
│
├── team15_exploration/                # Main Python package directory
│   ├── __init__.py                   # Python package initialization file - makes directory a Python package
│   ├── explore_nav.py                # Main exploration and navigation node - implements frontier-based exploration algorithm
│   └── explore_nav2.py               # Alternative exploration implementation - backup or experimental version
│
├── config/                            # Configuration files directory
│   ├── nav2_params.yaml              # Navigation2 stack parameters - controls robot navigation behavior (AMCL, planners, etc.)
│   └── slam_params.yaml              # SLAM Toolbox parameters - defines mapping settings, frame IDs, and map resolution
│
├── launch/                            # Launch files directory (currently empty)
│
└── resource/                          # Resource files directory
    └── team15_exploration             # Package resource marker file - identifies this as a ROS2 package resource
```

***Note**: The Yaml files are currently not used*

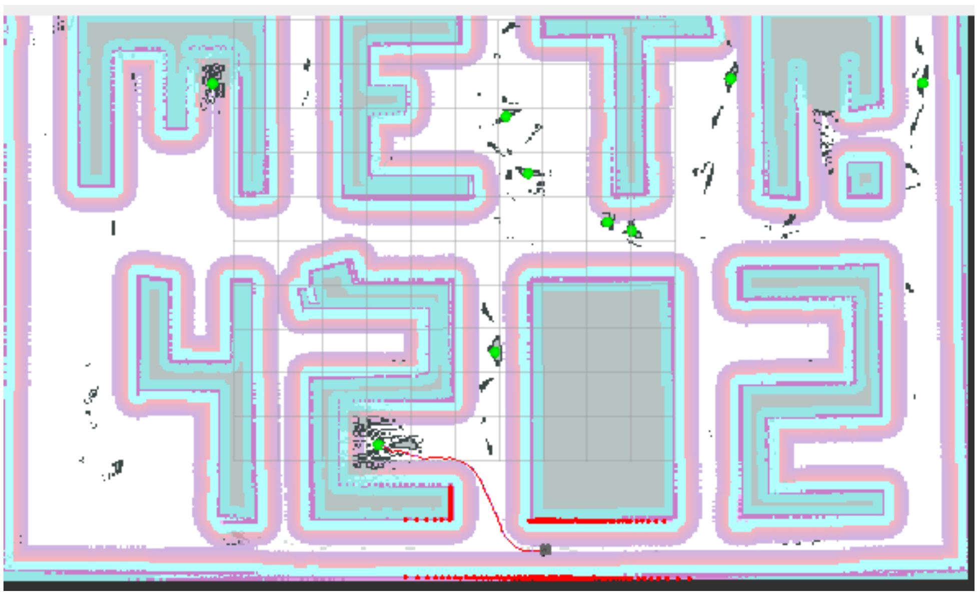
5.2 GitHub Repo

https://github.com/BlazeCentral/metr4202_2025_team15/blob/main/README.md

6.0 Testing

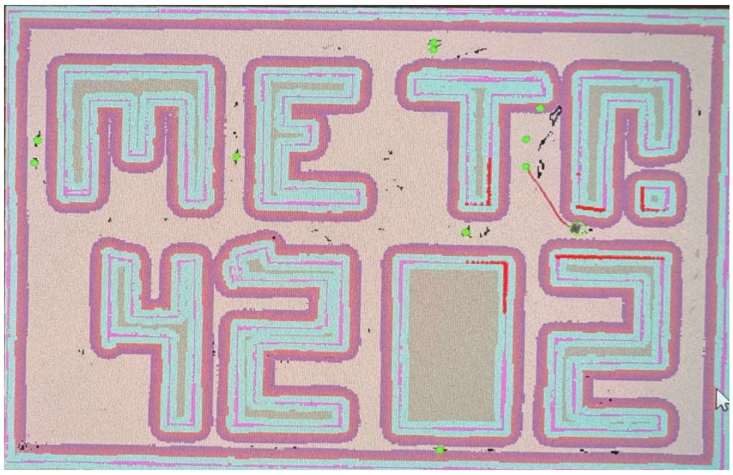
6.1 Visualisation

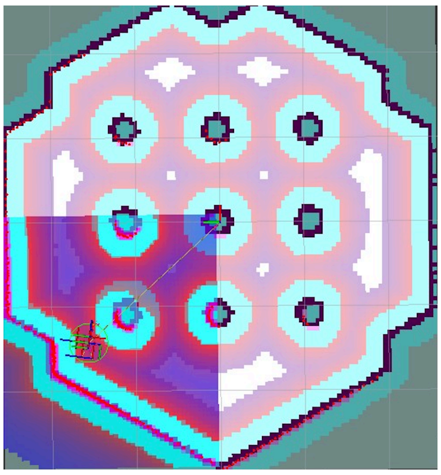
To assist with testing and validation, frontiers and frontier points are displayed on Rviz using OccupancyGrid and Marker objects. This helped display whether the algorithm was working as intended and allowed implemented changes to be evaluated effectively. As seen below, black borders represent frontiers, green markers indicate available waypoints, and the red line indicates the planned path.



6.2 Exploration Performance

Test results indicate a successful implementation of the exploration node. Several maps of increasing complexity were used for testing, with refinements to the algorithm based on performance. The first image below is the automatic base map from gazebo, once this run was getting 100% explored efficiently a more complex map was developed. This map can be seen in the second image below, a replica of the marking assessment map. This map highlighted the issues in the exploration techniques and changes were constantly made to develop the most efficient and successful code. Once developed, multiple tests were completed with run times averaging 25-28 minutes per test and 100% exploration.





7.0 Conclusion

Overall, the navigation and exploration logic of the project is working successfully, with the TurtleBot able to navigate a completely unknown environment and generate a virtually complete map. Strengths lie in the scoring algorithm which selects frontiers based on how much new map area they will reveal. Current issues lie in the map traversal speed, as the robot sometimes debilitates decisions for longer than necessary. Progress is looking promising, with AruCo functionality soon to be implemented.

7.0 References

Reference	Use
Gemini	Used to develop pseudocode for explore_nav/aruco file, and from a flow chart made from this, Gemini was used to develop the full py code files. Also the file structure and definitions were made using Gemini.
Git	The Aruco came from a specific library, and gemini was used to adapt it for Rviz2