Exploration of Unknown Map with ROS2 and Nav2

METR4202 SEM2 - FINAL REPORT 2025 – TEAM 15

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

1.0 Introduction 4
2.0 Design Goals 4
3.0 System Architecture 5
3.1 Node Structure 5
3.2 TurtleBot 3 5
3.3 Exploration & Navigation Module 6
4.0 Implementation 6
4.1 Exploration 6
4.1.1 Frontier Detection 6
4.1.2 Frontier Clustering 9
4.1.3 Goal Selection and Ranking Strategy (Heuristic) 10
4.2 AruCo Perception Node 12
4.2.1 Initialisation and Calibration 12
4.2.2 Image Processing 12
4.2.3 Pose Estimation & Transforms 12
4.2.4 Tracking/Smoothing and Publishing 13
4.3 Integrated Libraries 14
5.0 Recovery & Finishing 15
5.1 Stuck State Definition and Detection 15
5.2 Recovery Mechanism 15
5.3 Finish Condition 15
6.0 Testing 16
6.1 Visualisation 16
6.2 SIMULATION: Exploration Performance 17
6.3 SIMULATION: Aruco Detection Performance 17

6.4 HARDWARE: Exploration 18
6.5. HARDWARE: Detection Only 19
7.0 Conclusion 19
8.0 References 20

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. The goal of the project is to code a TurtleBot to autonomously explore and map unknown territory while also visually identifying designated targets by leveraging existing robotics libraries and simulation tools.

In its current state, the TurtleBot is able to successfully and reliably navigate an unknown area and build a map of its surroundings in simulation. In addition, the robot is able to interpret live camera footage to identify AruCo markers during exploration. The TurtleBot is 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. In addition, it is able to perform to a high standard on hardware, clearly mapping and identifying targets given that LiDAR and Cost maps are running.

2.0 Design Goals

On the low level, the robot must be able to explore an unknown environment using sensor data to navigate and 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 pose on the map. The run will be considered successful when the map is 100% completed (ignoring noisy/minimal frontier clusters under 4 cells) and at least one 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.

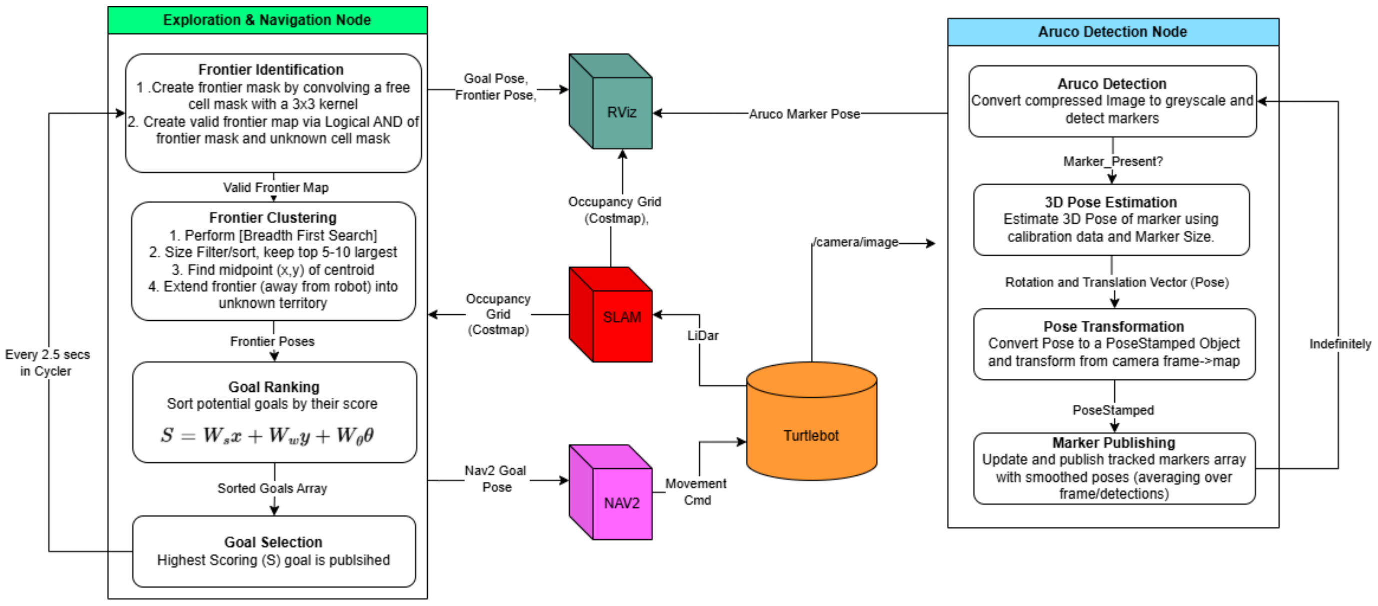
1. Before Navigation, use an algorithm to identify unexplored frontiers.
2. Before Navigation, have a developed frontier/goal selecting algorithm.
3. Navigate to a specified waypoint from an arbitrary starting position.
4. During Navigation, use sensor data to avoid obstacles.
5. During Navigation, use SLAM to build a map of environment.
6. During Navigation, use the Camera and OpenCV to identify Aruco Markers in the testing environment
7. After Navigation, update/publish a completed map with Aruco Marker position(s) 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 (OpenCV) libraries to identify and scan AruCo markers.



Exploration & Navigation Node
Frontier Identification
1 .Create frontier mask by convolving a free cell mask with a 3x3 kernel
2. Create valid frontier map via Logical AND of frontier mask and unknown cell mask
Valid Frontier Map
Frontier Clustering
1. Perform [Breadth First Search]
2. Size Filter/sort, keep top 5-10 largest
3. Find midpoint (x,y) of centroid
4. Extend frontier (away from robot) into unknown territory
Frontier Poses
Goal Ranking
Sort potential goals by their score
$S = W_s x + W_w y + W_\theta \theta$
Sorted Goals Array
Goal Selection
Highest Scoring (S) goal is publsihed
Every 2.5 secs in Cycler
Goal Pose, Frontier Pose,
RViz
Aruco Marker Pose
Occupancy Grid (Costmap),
Occupancy Grid (Costmap)
SLAM
LiDar
/camera/image
Turtlebot
Nav2 Goal Pose
NAV2
Movement Cmd
Aruco Detection Node
Aruco Detection
Convert compressed Image to greyscale and detect markers
Marker_Present?
3D Pose Estimation
Estimate 3D Pose of marker using calibration data and Marker Size.
Rotation and Translation Vector (Pose)
Pose Transformation
Convert Pose to a PoseStamped Object and transform from camera frame->map
PoseStamped
Marker Publishing
Update and publish tracked markers array with smoothed poses (averaging over frame/detections)
Indefinitely

Figure 1: System Diagram

3.2 TurtleBot 3

The existing TurtleBot system is equipped with a LiDAR sensor and camera. The 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, cluster area and angle of waypoint from the Turtlebot.

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 for the simulation and 75 for the hardware was used to allow the robot to traverse narrow corridors with no free space, whilst maintaining frontier distance from lethal cells. Using this threshold ensures goals are not placed in unreachable areas. Table 1 shows this algorithm on the next page.

Step	Description	Justification
1. Prepare Data	Check map availability, convert to 2D numpy array from 1d Cost map 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) is a map with each cell essentially containing the count of adjacent free cells (blurs together),	Allows us to see the neighbourhood of each cell.
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.	

TABLE1: Frontier Algorithm

(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 and 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. Centre calculation	Calculate midpoint of the cluster by choosing midpoint of (x,y) arrays.	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

Table 2: Cluster Algorithm

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. Table 4 on the next page shows the algorithm in full.

Frontier Ranking Score

Score for a candidate goal (Weightings shown in Table 3):

S = 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	Simulation Value	Hardware Value	Justification
W_S	2.3	3	High weighting for clusters with high unknown cell count.
W_w	-2	-5	Negative value favours nearby waypoints. Low est value to ensure depth of exploration.
W_Theta	5	6	Highest weighting promotes efficient exploration with little doubling back.

Table 3: Weighting Choices

Select & Rank Goal Algorithm set_goal()

Step	Description	Justification
1- Init	Init empty scored list	N.A.
2- Iterate Points	For each frontier point, perform steps 3-7. Then skip to step 8.	Ensure each point is handled
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.	N.A.

Table 4: Select and Rank Algorithm

4.2 AruCo Perception Node

A custom AruCo Node has been developed for the purpose of identifying, locating 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.

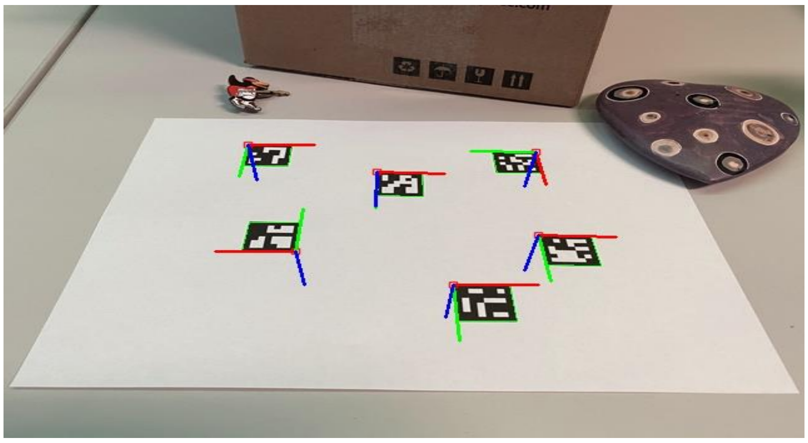


Figure 1: Aruco Markers

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 marker pose relative to a known 3D object (The camera frame) by taking these vectors and converting them to a yaw quaternion, and a PoseStamped is built in the camera's optical transform frame. Finally, that pose is transformed into the map frame using the TF2 transform tree, from the camera to the map in order for Rviz to have access to the specific pose for display purposes.

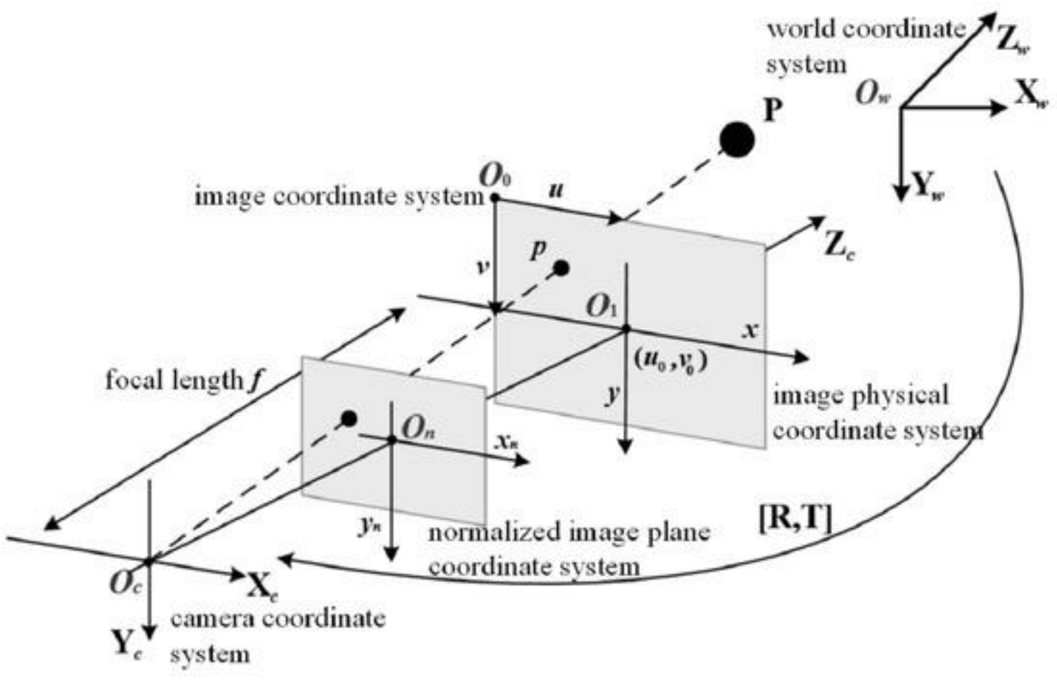


Figure 2: Solve PNP Algorithm

4.2.4 Tracking/Smoothing and Publishing

Each marker is tracked via a memory array with smoothing system by averaging pose across multiple valid detections using a weighted exponential moving average, in order to stabilise the reported pose. After 2 successful detections, the marker is classified as confirmed. These markers are continuously published to the Rviz map for visualisation, and the ID is printed to the terminal.

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 robot's 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 trigonometric 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 message request/response pattern with no fields and a success Boolean in response.	Manual trigger for exploration cycle.
SciPy	Convolution Algorithm.	For convolving frontier algorithm.

Table 5: Library Use

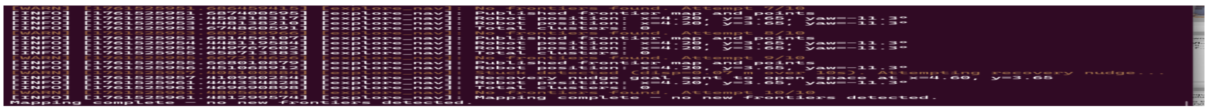
5.0 Recovery & Finishing

5.1 Stuck State Definition and Detection

We have a continuously monitoring watchdog function to check if there's been a displacement of a certain minimal distance (0.3m) from its prior spot (10 seconds ago), every 10seconds. If it detects a displacement <0.3m, and it has been >15secs since last recovery, then it initiates recovery mode.

5.2 Recovery Mechanism

The mechanism (try_recovery_nudge()) will try to 'nudge it' first by sampling candidate destinations around the robot in 3 rings of differing radii at different angles around it. It then checks to see the candidate is in open, safe space using _world_point_is_safe(), and reject any non-valid cells or high cost (>60) cells. Then, the first safe candidate found is sent as a Nav2 goal for the robot to try reach, with a blocking sleep() to push it towards recovery state.

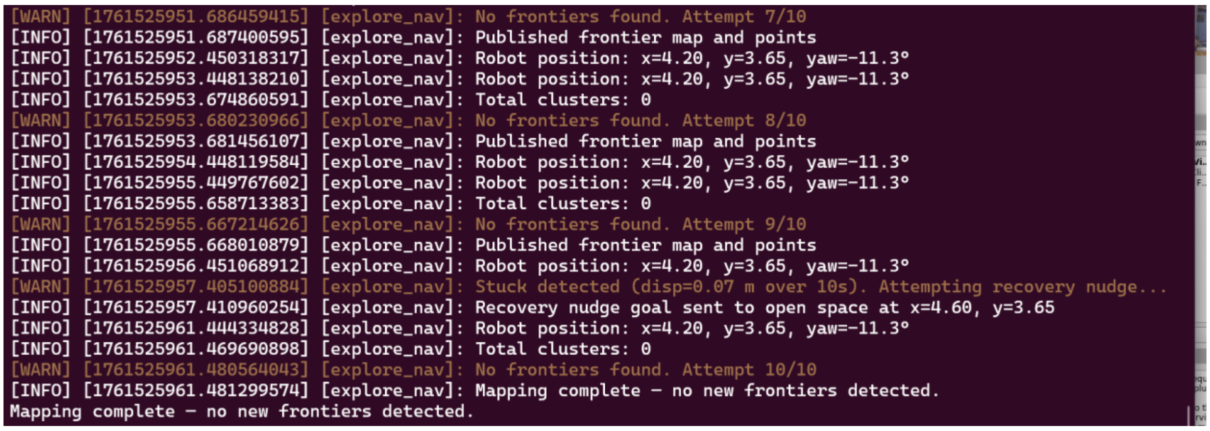


```
[WARN] [1761525951.6864594415] [explore_nav]: No frontiers found. Attempt 7/10
[INFO] [1761525951.6874000595] [explore_nav]: Published frontier map and points
[INFO] [1761525952.4503183317] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525953.4481382210] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525953.6748600591] [explore_nav]: Total clusters: 0
[WARN] [1761525953.6802309666] [explore_nav]: No frontiers found. Attempt 8/10
[INFO] [1761525953.6814556107] [explore_nav]: Published frontier map and points
[INFO] [1761525954.4481195584] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525955.4497667602] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525955.6587133383] [explore_nav]: Total clusters: 0
[WARN] [1761525955.6672146626] [explore_nav]: No frontiers found. Attempt 9/10
[INFO] [1761525955.6680108879] [explore_nav]: Published frontier map and points
[INFO] [1761525956.4510689912] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[WARN] [1761525957.4051008884] [explore_nav]: Stuck detected (disp=0.07 m over 10s). Attempting recovery nudge...
[INFO] [1761525957.4109602254] [explore_nav]: Recovery nudge goal sent to open space at x=4.60, y=3.65
[INFO] [1761525961.4443348828] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525961.4696908898] [explore_nav]: Total clusters: 0
[WARN] [1761525961.4805640043] [explore_nav]: No frontiers found. Attempt 10/10
[INFO] [1761525961.4812995574] [explore_nav]: Mapping complete – no new frontiers detected.
Mapping complete – no new frontiers detected.
```

Figure 3: Recovery Debug

5.3 Finish Condition

The exploration process terminates when no new frontiers can be identified for ten consecutive update cycles. Each cycle attempts to generate new frontiers from the most recent occupancy grid; if no valid frontier clusters are found, a counter is incremented and printed to the terminal, showing the number of unsuccessful attempts. Once this counter reaches ten, the system concludes that the environment has been fully explored, and a "Mapping complete" message is displayed on the terminal, signalling the end of the exploration process.



```
[WARN] [1761525951.686459415] [explore_nav]: No frontiers found. Attempt 7/10
[INFO] [1761525951.687400595] [explore_nav]: Published frontier map and points
[INFO] [1761525952.450318317] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525953.448138210] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525953.674860591] [explore_nav]: Total clusters: 0
[WARN] [1761525953.680230966] [explore_nav]: No frontiers found. Attempt 8/10
[INFO] [1761525953.681456107] [explore_nav]: Published frontier map and points
[INFO] [1761525954.448119584] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525955.449767602] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525955.658713383] [explore_nav]: Total clusters: 0
[WARN] [1761525955.667214626] [explore_nav]: No frontiers found. Attempt 9/10
[INFO] [1761525955.668010879] [explore_nav]: Published frontier map and points
[INFO] [1761525956.451068912] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[WARN] [1761525957.405100884] [explore_nav]: Stuck detected (disp=0.07 m over 10s). Attempting recovery nudge...
[INFO] [1761525957.410960254] [explore_nav]: Recovery nudge goal sent to open space at x=4.60, y=3.65
[INFO] [1761525961.444334828] [explore_nav]: Robot position: x=4.20, y=3.65, yaw=-11.3°
[INFO] [1761525961.469690898] [explore_nav]: Total clusters: 0
[WARN] [1761525961.480564043] [explore_nav]: No frontiers found. Attempt 10/10
[INFO] [1761525961.481299574] [explore_nav]: Mapping complete - no new frontiers detected.
Mapping complete - no new frontiers detected.
```

Figure 4: Finish Condition

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.

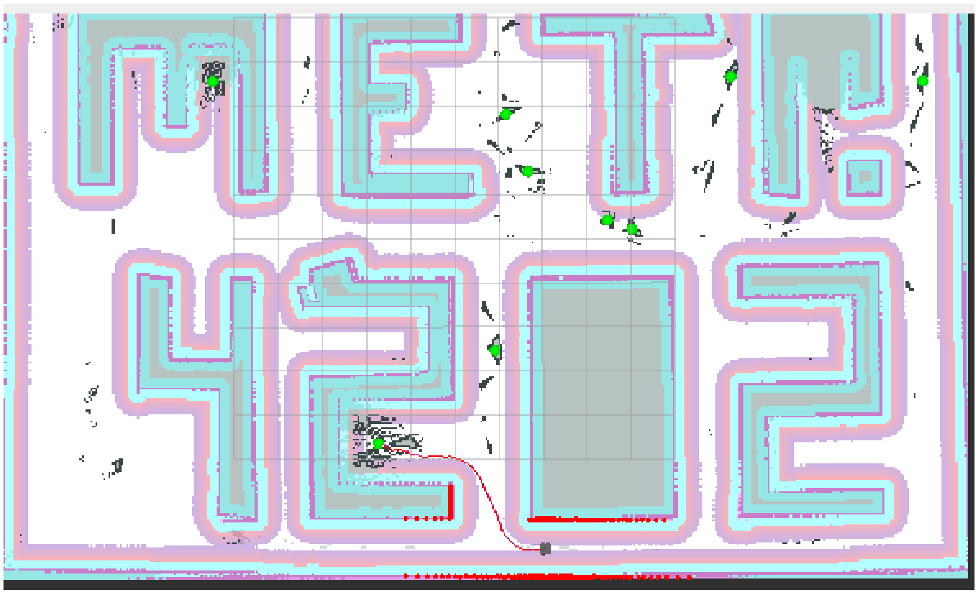
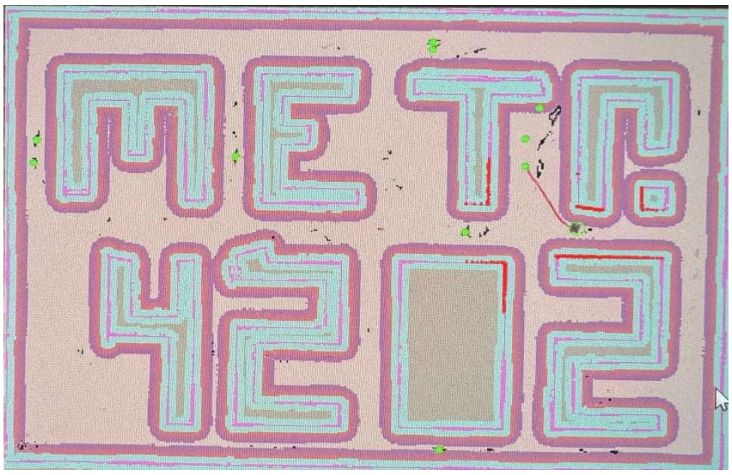
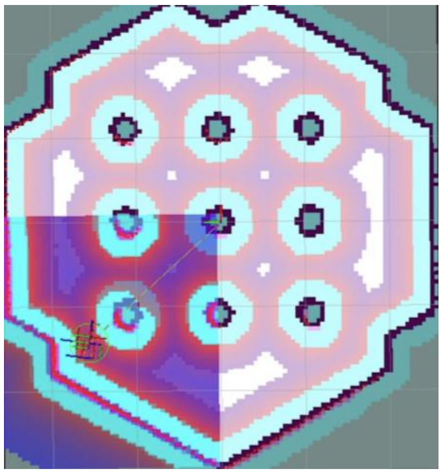


Figure 5: Frontier Map Rviz

6.2 SIMULATION: 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 replication 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.





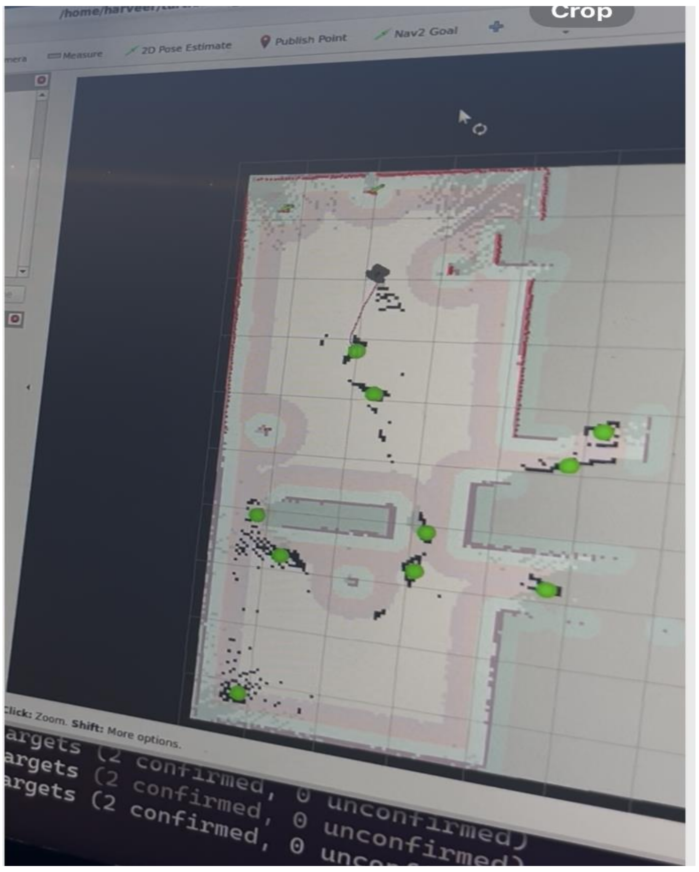
Figures 6-7: Simulation Exploration Performance

6.3 SIMULATION: Aruco Detection Performance

The Aruco Marker Detection performance in the simulations had significantly high-quality accuracy. It managed to find multiple (unique ID) AruCo markers and return their accurate marker ID as well as an accurate pose within ±0.05m published to the map (displayed on Rviz). It showed up quickly on the visualisation (within <1sec of the marker being in the frame of the RGB camera even at a distance of 5-6metres).

When this node was launched with the exploration node it still retained high accuracy even when the robot's pose was changing rapidly (e.g. rotating or traversing quickly).

Therefore, we considered the robot's combined (AruCo and exploration) system to be very successful in simulations.



```
argets (2 confirmed, 0 unconfirmed)
argets (2 confirmed, 0 unconfirmed)
argets (2 confirmed, 0 uncon
```

Figures 8: Aruco Marker Detection on hardware

6.4 HARDWARE: Exploration

Once success was reached on simulation for all components, it was necessary to begin testing on hardware. When testing on hardware, the team came across a few problems some that could be fixed and others were a bit out of the team's control. The first problem faced was that the robot would travel at a speed faster than the lidar and cost map would update. This meant that on multiple initial tests the robot would run through the walls. To fix this the max speed of the robot was updated. Another issue that was fixed was the waypoint distance and waypoint extension distance. In simulation the maps used for testing were much larger than in simulation. This meant that in hardware testing the waypoint was getting put on the opposite side of the map which was sometimes behind or within walls. However, the more prominent issue with the far waypoint distance was that it would give the LiDAR limited time to update.

From the beginning, a prominent issue that was noticed was the uncertainty of the LiDAR software during launch and during the testing process. Multiple times, the LiDAR would need to be relaunched due to the cost map or the initial LiDAR data not being loaded into Rviz. In addition, even if all necessary layers were loaded into Rviz in the beginning once the robot would begin to move to its first waypoint, the LiDAR would never seem to load, and the robot would fully just rely on the cost map to avoid obstacles and move around. This meant that the robot would get to random points around the map and stop moving or appear stuck. This uncertainty in testing meant that the hardware explore nav file could never be fully debugged or tested instead it was only able to be slightly optimised to the no LiDAR conditions. During the last block of testing, the LiDAR only appeared to update properly once in a 6-hour block.

The combination of the uncertainty of the LiDAR with the smaller testing map meant that the weightings of the frontier selection had to be modified. It was noticed that without the LiDAR the robot would go to frontier goals it had already been to, therefore the weighting for goals in front of the robot was increased. Additionally, the distance weighting was further decreased to ensure that closer waypoints were favoured due to the smaller map size. The cluster count weighting remained the same.

6.5. HARDWARE: Detection Only

However, despite all this once camera calibration was set up and the software was understood, the AruCo Detect function was very successful at finding a AruCo marker nearly every time. The only issue faced with the QR detection was that when the robot was still and facing the AruCo marker it would place the waypoint on the Rviz map perfectly however if the robot was moving the waypoint placement on the map was inaccurate. However, this problem was hard to debug because it was unclear if this was due to problems with the transforms or the LiDAR not updating.

7.0 Conclusion

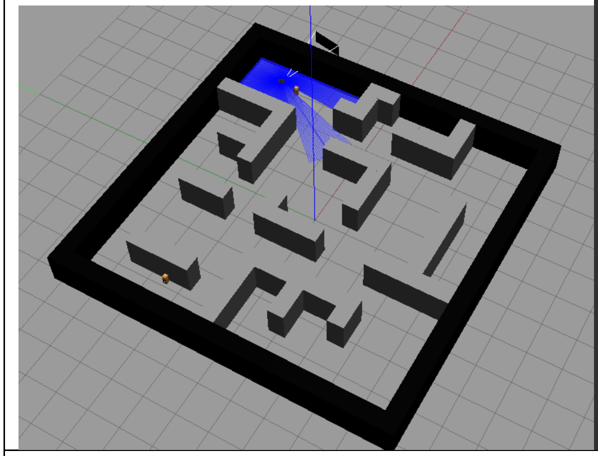
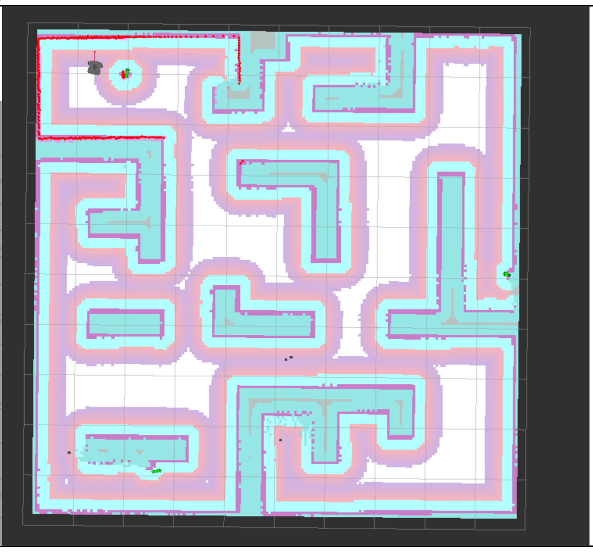
Overall, the navigation and exploration logic of the project is working successfully, with the TurtleBot able to navigate a completely unknown environment and generate a virtually complete map. Strengths lie in the scoring algorithm which selects frontiers based on how much new map area they will reveal.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

8.0 References

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

Reference	Use
AI	A combination of Cursor AI, ChatGPT and Gemini was used to develop much of the foundations of the code. Pseudocode and flow charts where inserted into the chats in order to intelligently develop out ideas into working prototype code which were adjusted and customised for the robot based on our needs and performance. The code was finalised with human hands especially the weightings and specific parameters, and the heuristic scoring code.
Git	The following Github repository was used to help with the development of the Aruco Detection Node: https://github.com/Rishit-katiyar/ArUcoMarkerDetector/tree/main The repository of provided aruco course resources was also used to help with development: https://github.com/METR4202/metr4202_aruco_explore

Table 6: Referencing

Testing Results
 
+ Map used was from **Metr 4202 GitHub** + Best completion time **10mins 31sec** + Average Estimated Completion Time **13mins** + Able to detect all **3 Aruco makers** + **Did not hit a wall**