Autonomous Simultaneous Exploration and Mapping of a 2D Indoor Environment with Known Pose Using TurtleBot in Gazebo Simulation

Course: MECH 524
Year: 2025

Prepared by:
Xuezheng Chen 32470387
Liang Yan 33140351
Connor McAllister 65908436

1. Project Overview

This project proposes the development of an autonomous exploration system for a mobile ground robot (TurtleBot2 platform) operating in a simulated indoor environment. The goal is to enable the robot to autonomously navigate a previously unknown space, avoid obstacles, and incrementally build an accurate 2D occupancy map of the entire room with the frontier exploration methodology. The project is implemented entirely in simulation using Gazebo and developed in C++ with ROS (Robot Operating System). The final deliverables of this project are a stand-alone ROS package and a final report. The detailed capabilities of the package are in the section Summary of Project Capabilities.

The entire task project is split into three subtasks:

1. Occupancy map generation
2. Global path-planning, and
3. Local path-planning

Each of the three group members is responsible for one subtask, while contributing to the shared tasks, such as simulation set-up, coding shared utility functions, and integrated testing and debugging.

Since the map used is static, the experiment process and results can be reproduced with proper installation of the compatible operating system (Ubuntu 16.04), installation of all necessary dependencies, and correct compilation of ROS package. The detailed installation and compilation processes will be included in the README.md file in the GitHub repository.

2. Summary of Project Capabilities

The project is a ROS package compatible with ROS Kinetic that supports only Ubuntu 16.04 (Xenial) Operating system. The ROS package contains two nodes in parallel: "occupancy grid node" and "path planning node":

1. The occupancy grid node is responsible for subscribing to continuous data streams from sensors mounted on the TurtleBot, including the current occupancy grid and visit count grid, odometry data feed, and bumper, continuously updating its knowledge of the environment. It then publishes these data streams to the path planning through individual ROS topics.

2. The path planning node is responsible for robot locomotion. The global planning module generates a path from the robot's current position to the nearest of all nearest accessible points in proximity of all current frontiers, where accessibility indicates that if the robot's center is on a given cell, its rim does not hit any obstacle or unknown cell. The local planning module follows the accessibility rule as the global one. However, during testing, the robot's radius was inflated by 1.7, as using exactly the radius would cause the robot to get stuck when turning around tight corners.

The two nodes communicate with each other through ROS topics and broadcasts, both public data buses that allow the occupancy node to publish the current occupancy and visit grids, as well as the current robot states to the path planning node. The path planning node then processes this information and makes the next decision. These run concurrently along with other nodes from the third-party TurtleBot hardware-related packages.

Upon running the ROS package, a TurtleBot is spawned in the center of the Gazebo simulation environment. Two empty maps, one occupancy map that marks each cell as one of occupied, open, or unknown, and a robot visit count map that counts how many times the robot has visited a specific location, are generated. From these two maps, the most critical decision the robot must make is choosing a next destination that allows it to explore the largest amount of unknown space while avoiding obstacles and minimizing repeated traversal of highly visited locations. These locations are called frontiers -- locations on the map that lie between unknown and free cells. The sensory information needed for the robot to identify such frontiers and to constantly detect and pre-emptively avoid obstacles en route is primarily obtained by using a LiDAR scanner. It marks the space scanned as either free or occupied, while leaving locations never scanned as unknown. Once a frontier is identified, the global planning module provides a path from the robot's current position to the frontier that exclusively traverses known open cells while also minimizing the total distance. The local planning then commands the robot to keep on the path while also reactively avoiding obstacles previously unseen at global path planning time. The robot will keep exploring the environment for a fixed time period, or until there is no unexplored space on the map.

The detailed guidelines for running the program will be included in the README.md file in the GitHub repository. The following three sections introduce the detailed functionalities of our three modules and provide illustrations.

3. Functionality of Occupancy grid generation

3.1 Maps

The occupancy grid module manages the TurtleBot's sensors and its perception of its environment. The robot perceives the world by continuously updating two maps. The occupancy map is a discrete map with "cells" as the minimal unit. The map has a predefined number of rows and columns, both 300, and the resolution is set to 0.05m, which means the map covers an area of 15m by 15m. The occupancy state of each cell is represented by an integer: -1 for an unknown (unexplored) cell, 0 for a free (empty) cell, and 100 for an occupied cell, where intermediate values are later used for smoothing. This setting is in accordance with the existing occupancy grid message type in ROS, which uses integers from -1 to 100 to represent cell states. The second map is called the robot visit map and counts how many times the robot has visited each cell. This map has identical dimensions and resolution to the occupancy map. To filter out redundant visit counts when the robot is stationary in a cell or performing a rotation, the robot needs to reach a threshold speed when passing a cell to add the visit count.

3.2 Frames and robots

There are two frames of our concern in the simulated world - the occupancy grid frame and the odometry frame. The origin of the occupancy grid frame is at the bottom left corner of the world, so that all coordinates used in our algorithms are non-negative. This is the frame where all computation is done. The odometry frame is attached to the robot and measures the robot's current pose relative to its initial pose, where a pose is the robot's (x,y) position and orientation at any given moment. The robot's pose is continuously tracked by ROS's `/odom` topic. Since this information as provided by ROS is in the odometry frame, the occupancy node stores a transformation matrix for each robot that converts coordinates from the odometry frame to the occupancy grid frame. This transform is also broadcast to the path planning node to transform every odometry update it receives to the occupancy map frame. The transformed robot coordinate is used for map updates using laser and bumper data.

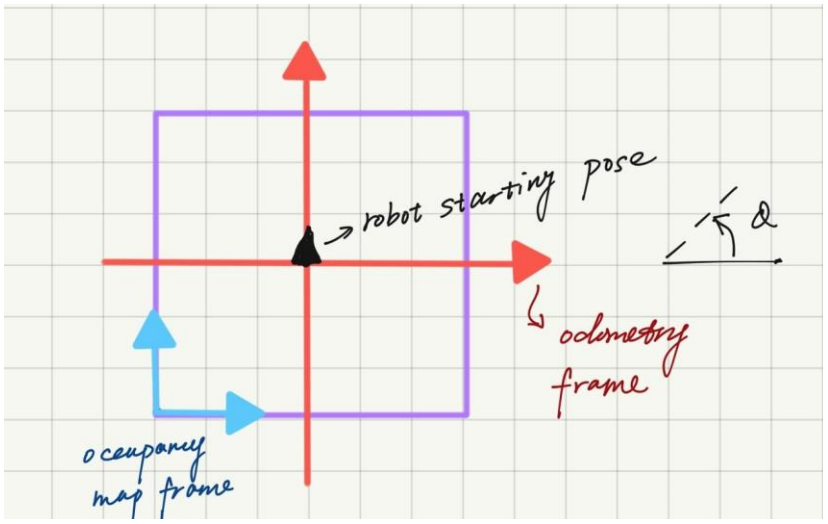


Figure 1. A visual representation showing the relationship between different frames and the robot's initial pose

3.3 Map update with laser data

The LiDAR sensor is the most effective way for the robot to generate the occupancy map. A schematic of a laser sensor is shown in Figure 3. The sensor data encodes the angle increment and how far an emitted ray has travelled before it is reflected to the receiver mounted on the Kinect sensor suite.

Upon receiving the laser data input, our algorithm first filters out invalid readings that are either below the minimum detection range or beyond the maximum detection range, which indicate either that the obstacle is immediately in front of the robot or the laser ray never returned, and the obstacle is too far for the LiDAR to detect. Since the more extended the detection range, the more dispersed the laser beams will be, to guarantee a smooth and continuous map, all laser data is also capped at 2 m, such that the algorithm will mark the point as occupied if a valid reading is less than 2 m, and skip the particular beam if more than.

After pre-processing, the algorithm uses Bresenham's line-drawing algorithm to mark cells in the discrete occupancy map based on the pre-processing result. Subsequently, all cells between the robot’s current position and the cells marked in the previous step are marked as free. The process is iterated with a frequency of 5Hz.

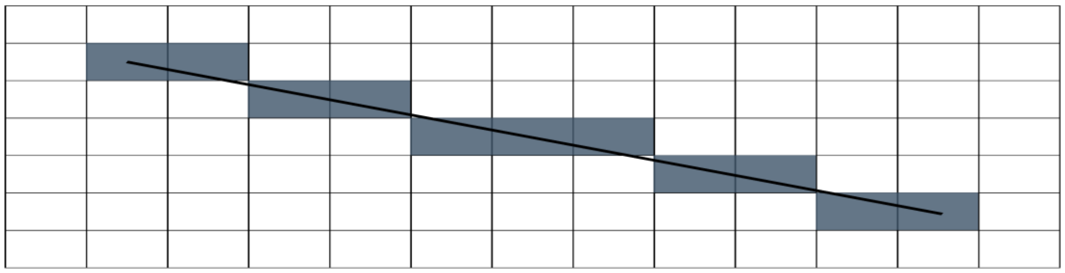


Figure 2. A schematic of Bresenham’s line-drawing algorithm to mark discrete cells with two points. Credit: [1]

3.4 Map update with bumper data

The robot has three bumpers, one front bumper and two side bumpers. Each side bumper forms a 60-degree angle with the robot's front. Due to hardware limitations, only one bumper can be triggered at a time. Our algorithm accepts bumper messages every 0.5 seconds. If a bumper is triggered, five cells parallel to its surface will be marked as occupied. Even though the bumper is not the primary source of map update information, it still serves a critical role in obstacle avoidance when the laser scanner result is not able to capture all obstacles due to reasons such as rapid robot movement, small obstacles, or obstacles with irregular shapes.

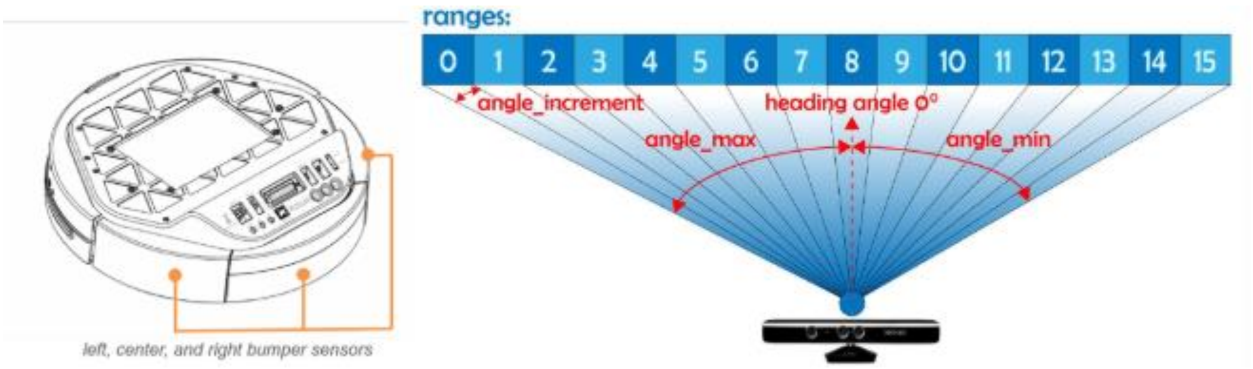


Figure 3. The main sensors of the TurtleBot. The left image shows the location of the front and side bumpers. The right image illustrates the laser scanner. The laser scanner emits laser rays separated by an angle increment and bounded by the maximum angle on both sides. Credit: [2]

3.5 Frontiers

Inspired by [3], we employed a frontier-based exploration strategy. The frontier-based exploration strategy is centered around “to gain the newest information about the world, move to the boundary between open space and uncharted territory”, and is famous for guaranteeing that all unknown spaces on the map can be explored if accessible to the robot. According to [3], a free cell with at least one unknown cell among its eight neighbors is defined as a frontier edge. Frontier cells serve as targets for the robot to explore. Whenever a request for a new frontier is received, the occupancy grid module will scan through the entire discrete occupancy map and obtain all frontiers. Then, the Euclidean distance between the robot's current position and all frontiers will be computed, and the nearest frontier will be published as the robot's next exploration destination.

4. Functionality of Global path planning

4.1 Local and Global Mapping

Path planning occurs on its own ROS node to allow separation from map generation and path planning logic. As such, incoming occupancy grid messages must first be converted to a format that is easily usable by various path planning algorithms.

Global mapping was done by storing two separate map types. A `GlobalMapGrid` object is first used to store occupancy and visit count data for each node within the occupancy grid, as well as relevant data such as dimensions and map resolution. This object is updated regularly from ROS messages sent by the mapping node. It is meant to act as the link between all discretized path planning algorithms and the real world, with several methods meant to convert between (row, column) indices and nodal distances to real (x, y) coordinates and distances.

While the `GlobalMapGrid` is meant to represent real space and is shared by all robots within a given environment, `RobotLocalMap` instances are unique to each robot, and store real distances to the nearest obstacle, as well as nodal accessibility as it pertains to each individual robot. This class was defined in response to the finite size of a robot, where the odometer located at the centre of the robot must clear obstacles by a minimum distance to avoid collisions. Within this grid, each node stores distance to the nearest obstacle, computed via a brush fire algorithm [4]. This was then compared to the minimum clearance of the robot to determine accessibility of each node.

4.2 Discrete Path Generation

`DiscretePath` objects were used to store a generated path of a robot within the discrete nodal coordinates of the occupancy grid, containing a vector of `MapIndex` structs and the timestamp of the most recent update. Each robot has its own `DiscretePath`.

A `DiscretePath` is instantiated as empty, and updated with a `PathPlanner` object, which contains non-owning pointers to the environment's `GlobalMapGrid` and the robot's `RobotLocalMap`, as well as an abstract `generatePath` method. This allows `PathPlanner` objects to be used interchangeably within the overall structure of the application, where derived members can implement different algorithms as needed. The `PathPlanner` class contains a `SearchGrid` instance, used to store cost data of each node, and meant to act as a temporary workspace for the `PathPlanner` to conduct its search. For this application, an `AStarPlanner` class was created to implement a modified A* search algorithm [5]. In this application, the F cost of the generated path was updated to account for the number of times the nodes within the path have been visited, dictated by a sensitivity parameter decided by the user. This was chosen to be 0.5, such that traversing a visited node would be less favorable than travelling a further distance.

Once the `DiscretePath` was updated, a `refinePath` method was implemented to remove unnecessary points from the generated path of the robot. Beginning at the first index of the path, a discretized straight line was drawn to the end of the path, using Bresenham's line search algorithm discussed in 3.3. If any nodes within this path were deemed inaccessible, a straight-line path was evaluated to the second-to-last node. This process was repeated until a valid shortcut was found. The end node of this valid shortcut was added to the refined path, and this process was repeated with the end node as the new start node until the entire path was refined. This process resulted in a new path with only essential turning points for the robot being kept.

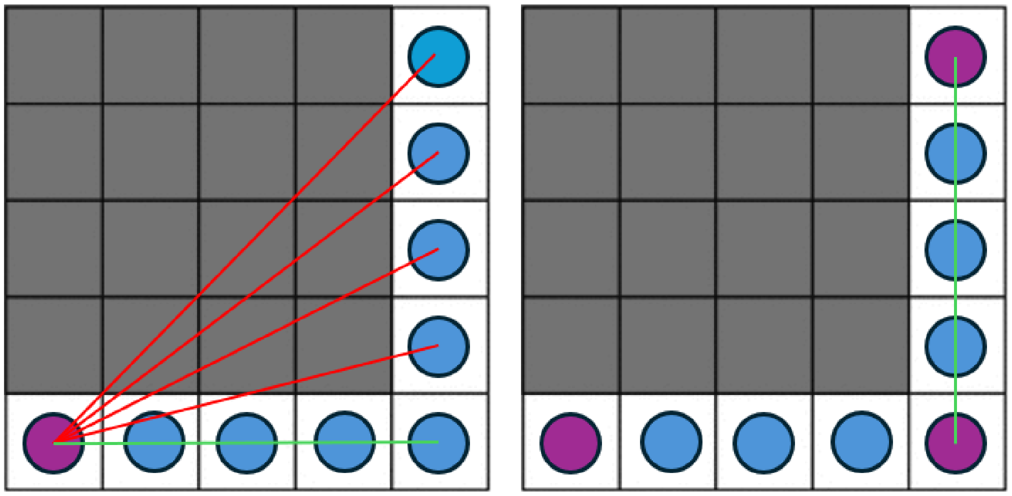


Figure 4. Graphic depicting shortcut-based path refinement for a simply 5x5 grid, with a corner turning path. Blue circles represent original nodes while purple circles represent refined nodes.

4.3 Real Path Interpolation

Once a discrete nodal path was obtained, it was converted to real coordinates and interpolated using a `PathInterpolator` object. Like `PathPlanner`, this class is abstract, with a purely virtual interpolate method, such that different types of path interpolators could be used interchangeably in the final product. For this application, a simple linear interpolator was selected in response to the TurtleBot's differential drive, where there is no minimum turn radius, and any introduction of path smoothing, as seen in cubic spline or quadratic Bezier interpolation, would result in deviation from the planned path. `RealPath` instances contain a non-owning pointer to a `DiscretePath` instance and are updated based on the contents of the `DiscretePath` when the `RealPath` coordinates are accessed. To avoid unnecessary updates, a time stamp is used for both path types to evaluate synchronization.

5. Functionality of Local path planning

The global path was created using data from the occupancy grid at the time of generation. When following this path, the local planner ensures that the robot also maneuvers reactively around obstacles that are previously unmapped at global path planning time, as well as inaccurate occupancy values due to sensor errors. If the robot does bump into obstacles, the robot is commanded to move away from the obstacle and regenerate a new global path.

5.1 Reactive Path Traversal – Dynamic Window Approach

For reactive path traversal, the dynamic window approach [6] is adopted. It first uses a `DynamicWindowSampler` object to calculate a dynamic window, centered around the current linear and angular velocities and bounded by the amount of velocity increase possible during the next main loop cycle at the maximum acceleration, thereby producing a range for feasible linear and angular velocities for the next command. A tunable number of evenly spaced candidate linear and angular velocities are sampled from the $\mathbb{N}^{v \times \omega}$ space to form (v, ω) pairs. The sample space dimension is set as 20×20 for this project.

To evaluate the best sample, each velocity pair is imaginarily held constant during a tunable prediction horizon, set at 2s for this project, which could span and predict several upcoming main control cycles. Each pair would then produce a sample trajectory of an arc, representing a close prediction of the robot's motion if it were to execute any given (v, ω) pair during the next prediction horizon. Each of such trajectories is then simulated and discretized by a `TrajectoryGenerator` object, using a tunable time increment.

Each trajectory is then evaluated based on the following three criteria: obstacle avoidance by an `ObstacleEvaluator` object, global path proximity by a `PathTracker` object, and finally, the magnitude of velocity. Physically, this means that the ideal velocity pair should lead to a trajectory that maximizes distance to any obstacle along itself, minimizes heading deviation from the global real path, and also provides the maximum possible velocity to allow for reaching the destination safely via the global path as fast as possible. Each criterion produces a score for each trajectory, and the highest total scorer is selected by a `TrajectoryEvaluator` object to be the velocity command to the robot during the next control cycle.

The obstacle evaluator also employs the brush fire algorithm that computes the distance to the nearest obstacle or unknown cell for all cells within the occupancy grid, but it is implemented as a lightweight version that operates without the intricate dependencies of the `RobotLocalMap` class in the global planning module. Given a simulated trajectory, it first evaluates whether each discretized point is within the robot radius to any obstacle or unknown cell. If this is the case, the trajectory is automatically rejected as it would lead to a collision. Otherwise, it finds the minimum distance to the nearest obstacle or unknown cell of all discretized points. This is normalized by division by the diagonal length of the world, which represents the longest possible straight line drawn in the room. It also compares the shortest braking distance, which is calculated with the highest deceleration, i.e. negative of the highest maximum acceleration, against the minimum clearance to any obstacle out of all points along a trajectory. If the braking distance exceeds the minimum clearance, this trajectory is also deemed inadmissible and automatically rejected.

The path tracker first extracts the end pose of every trajectory. It then computes the point on the real path closet closest to the robot's current position, and then imaginarily traverses forward on the path until a dynamically adjusted lookahead distance is reached, where this distance is proportional to the magnitude of the robot's current velocity. The path point at the end of the lookahead distance is selected as a reference point, and its angle from the robot's current position to this position is calculated to represent the directional change that the robot will have to make to stay on the path after it continues ahead for the prediction horizon. Next, the last pose on a given trajectory is subtracted from this angle, representing the heading error between this simulated trajectory and the robot's corrected future heading. This heading error is then subtracted from 1, so that the larger the heading error, the lower the heading score is. Now, the intent of the dynamic lookahead becomes clear – the faster the robot is moving, the further it has to look down the road, such that any sudden turns ahead can be spotted early and prepared for. The same logic applies to selecting the reference point used to compute directional change at a small distance ahead along the global path. Lastly, the magnitude of the linear velocity in this given sample is normalized by division by the maximum velocity limit of the robot to favour fast movement within dynamic constraints.

After each individual score and the Boolean admissibility indicator for all trajectories are calculated, they are combined into a total score with three tunable weights for each score. Tuning the weights has proven to be a crucial step in optimizing the robot's movement, as the relative magnitude of each weight dictates how much the robot favors obstacle avoidance, path tracking, and fast movement, respectively. The finalized weights from this project are 0.75 for obstacle avoidance, 0.18 for path tracking, and 0.08 for fast movement. The dominant weight for avoidance is to ensure the safety of the robot's movement, and the low velocity score came from the observation that during testing, fast velocities led to severe drift and error of laser processing. These weights were tuned simultaneously with the velocity and acceleration limits, the proportional control constant K for the robot's current velocity that determines the dynamic lookahead distance, and the minimum clearance of the robot.

Additionally, the team encountered the problem of the local planner running into local minima during simulation. This occurs when the robot is a small distance (roughly 0.3m) from its current destination and the frontier selected borders an obstacle, usually a wall or a series of close obstacles surrounding or lying closely ahead of the robot. This causes virtually all candidate trajectories to traverse occupied or unknown cells, and the local planner rejects all of them. This is widely known as one of the flaws of the dynamic window approach algorithm, as well as many others. To resolve, a default drift of a low linear velocity of 0.2 m/s and an angular velocity of the last angular velocity command before the local minimum was added to allow the robot to drift slowly along its last known path direction, until sufficient open cells have been scanned, and it can come close enough to the intended destination to generate a new global path.

5.2 Bumper Recovery

The local path planning module is augmented by a bumper recovery layer. While bumper data is continuously monitored, the bumper recovery maneuver is triggered by the rising edge of any one of the left, middle, and right bumpers. Upon triggering, the module commands the robot to back off by a tunable distance, then rotate by a tunable angle (finalized at 15 degrees) to the opposite side of the obstacle. If the middle bumper is triggered, then rotate by 45 degrees either left or right, and the direction executed is determined by a random seed. After such evasive maneuvers, the robot then reevaluates its frontiers, removes the current frontier from its list of potential destinations, and replans a path to a new frontier.

6. Functionality of Navigation Robot and Navigation Environment

`NavigationRobot` and `NavigationEnvironment` classes were used to integrate all components of the path planning node into a functional workflow. The `NavigationEnvironment` class contains everything pertaining to mapping the given room, including a global map, ROS interface, and all robot instances. It is responsible for providing a central point for all components to interface with one another.

The `ros_interface` class contains ROS subscribers that continuously receive odometry, bumper, occupancy grid, visit count grid, as well as frontier list information published by the occupancy grid node. For bumper information, it implements a rise edge detection mechanism that only triggers bumper recovery and replanning when the previous bumper state is released, and the current state is pressed, thereby avoiding repeated triggering when the robot is still in contact with an obstacle while performing evasive maneuvers. For the odometry data, upon program initialization, it listens to the transform from the odometry frame to the occupancy frame broadcast by the occupancy grid node and stores it internally, then uses it to transform every stream of incoming odometry data to the occupancy frame.

As for the `NavigationRobot` class, it contains all information pertaining to an individual robot, including velocity and size parameters unique to the robot, `RobotLocalMap`, `DiscretePath` and `RealPath` instances, its current destination, and `unique_ptr` instances containing `PathInterpolator` and `PathPlanner` objects, such that different derived members of these objects can be used interchangeably, and memory can be automatically managed.

The mapping application is run through the `NavigationEnvironment`'s `mapSurroundings` method, which first rotates the robot to map the visible surroundings. A list of frontiers is then obtained from the mapping node, and the closest accessible point to this frontier is added to a locally stored priority queue of destinations, sorted based on proximity. A global path is then generated to the closest destination. If no valid path exists to this frontier, the next closest frontier is evaluated. Once the global path is generated, the DWA local planner is used to evaluate possible trajectories of the robot and move the robot along the planned path until the proximity to the destination is within a certain threshold. At this point, new frontiers are obtained, and a new global path is generated. This process is repeated for 10 minutes.

7. Objectives vs. Reality

7.1 Results and discussion

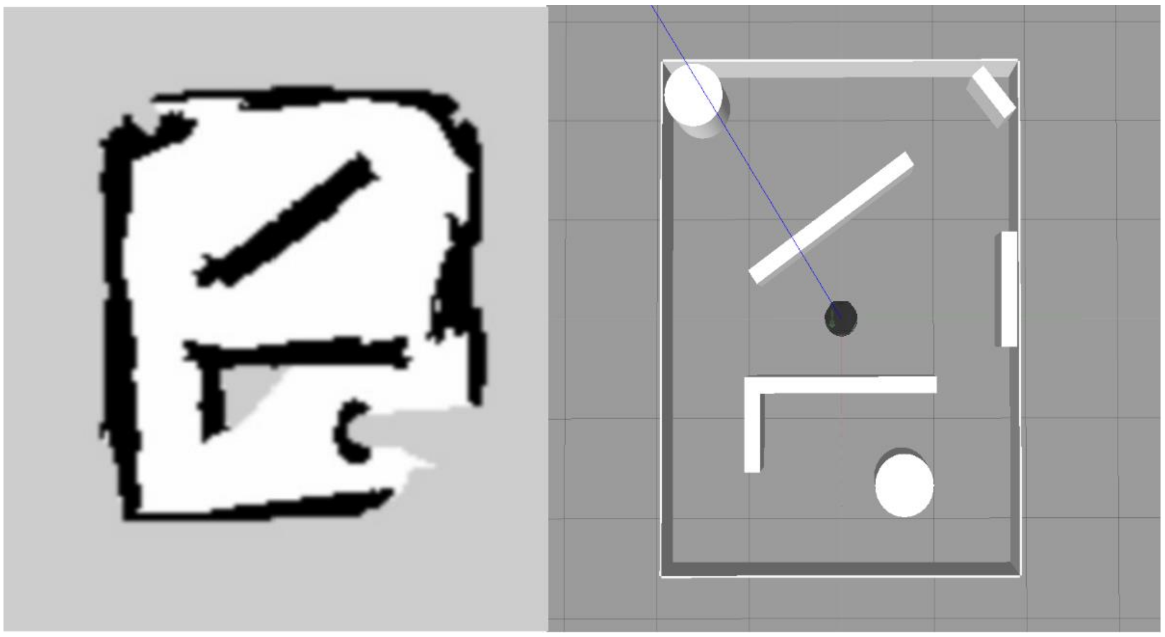


Figure 5. The result of robot exploration. The resulting occupancy map is on the left, and the simulated environment is on the right with the black TurtleBot in its spawn position. Note: the images are not in the same scale.

Figure 5 shows the most complete occupancy map after robot exploration from simulation testing. The map captures approximately 90% of all features in the simulated environment, including walls, cylinders, corners, etc. However, remaining flaws in the path planning module discussed in the following sections and inherent laser scan resolution limitations, such as rough edges and barrier penetrations, do exist that prevent the map from being described as accurate and smooth.

7.2 Limitations and potential improvements

Future extensibility of our project was highly prioritized during development. We have programmed our package to handle arbitrary map sizes and resolutions, as well as multiple robots spawned at random positions on the map, either via runtime user input or the launch file. However, the current work assumes a single robot spawned at the center of the Gazebo world frame, which is half the map width and height in the occupancy grid frame. It is a promising direction for improvement to include multiple robots in the same simulated environment, which may speed up exploration and increase map accuracy as robots validate each other's work.

The accuracy of maps generated with our algorithm cannot match that of existing ROS packages such as GMapping. We have observed problems such as noisy edges around obstacles, hollow areas inside objects, penetration of barriers, and mismatches between the occupancy map and the simulated environment. We consider our laser data-processing and cell-marking algorithms as the culprits. To improve consistency and accuracy, improving the filtering algorithm, such as using convolutional kernels, or improving the cell marking algorithm, such as assigning probabilities rather than marking cells at each scan, should significantly reduce the impact of noisy sensors and movements of the robot.

As for global path planning, the robot's ability to generate an optimized path is currently limited by the resolution of the global map. Thus, the first step toward improving this stage would be to optimally refine the map resolution in a way that balances accuracy and computational efficiency. This is especially relevant when considering that obstacle distances generated by brush fire are calculated by the shortest discretized path distance, rather than the absolute Euclidean distance.

For the local path planning, due to limited development time, the robot occasionally finds itself wedged at a cylindrical or prismatic obstacle, and it is unable to rotate out of it, creating a high amount of slipping and sometimes shaking. Displacements caused by these motions are not captured by corresponding increments in the Turtlebot's differential drive, introducing egregious errors in laser processing and severely distorting the map. To resolve this, a "stuck detector" must be added, where if the robot's position remains unchanged for a tunable duration of time, a forced re-evaluation of frontiers and replanning of the global path is triggered, and any frontier that is within the proximity of this previous obstacle is to be automatically discarded. If this were successfully implemented, the team believes that a near-complete mapping of the room is highly achievable. Moreover, due to limited development time, the team has only conducted intensive tuning and simulation testing in one map, shown in Figure 5.

Additionally, next steps could involve experimenting with different path generation algorithms, such as JPS for faster planning, or RRT* for simultaneous global planning and motion. Other interpolation algorithms could also be used, such as cubic spline interpolation to allow for continuous velocity and acceleration profiles.

8. Conclusion

We have created a ROS package in C++ for autonomous exploration of a simulated indoor environment using a mobile ground robot (TurtleBot platform). The robot navigates in an unknown space, avoids obstacles, and incrementally builds a 2D occupancy map of the entire room using the frontier exploration methodology. The occupancy grid module generates maps from laser and bumper sensors; the global path planning module handles path planning to approach the target, while the local path planning module maneuvers the robot and performs emergency obstacle avoidance. With a successful occupancy map generated as a result, the team also observed problems such as inaccurate alignment of the map with the simulation environment, unexplored regions, coarse surfaces, and being stuck near obstacles due to sub-optimal data processing and exploration algorithms. In future versions, such issues could be resolved with more advanced filtering techniques, increased map resolution, and more thorough debugging for edge cases.

9. References

[1] T. Stogiannopoulos and I. Theodorakopoulos, "Curved Text Line Rectification via Bresenham’s Algorithm and Generalized Additive Models," *Signals,* vol. 5, no. 4, 2024.

[2] University of Toronto, *TurtleBot Technical Manual for MIE 443H1S,* Toronto: University of Toronto, 2025.

[3] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97*, Monterey, 1997.

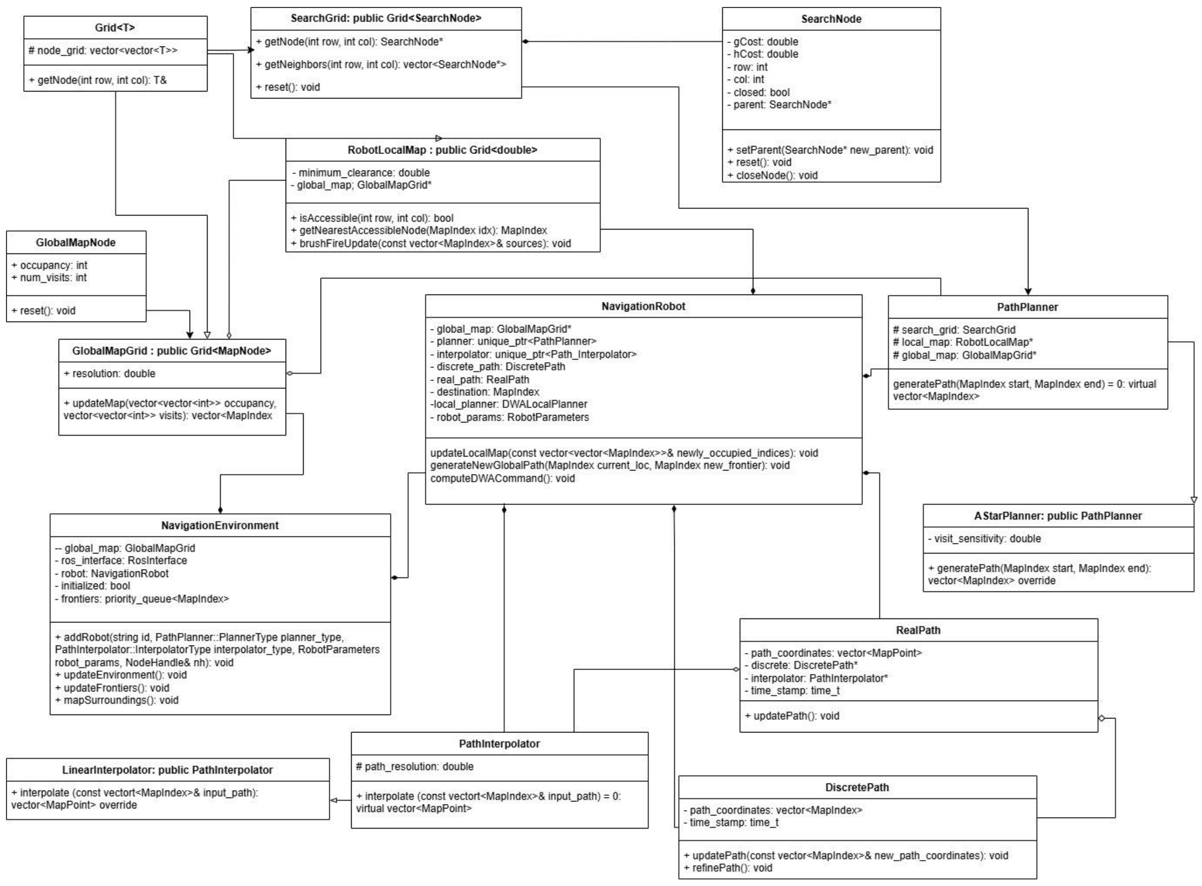
[4] H. Choset, "Robotic Motion Planning: Potential Functions," [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Chap4-Potential-Field_howie.pdf. [Accessed 12 November 2025].

[5] "A* Search Algorithm," Geeks For Geeks, 23 July 2025. [Online]. Available: https://www.geeksforgeeks.org/dsa/a-search-algorithm/. [Accessed 7 November 2025].

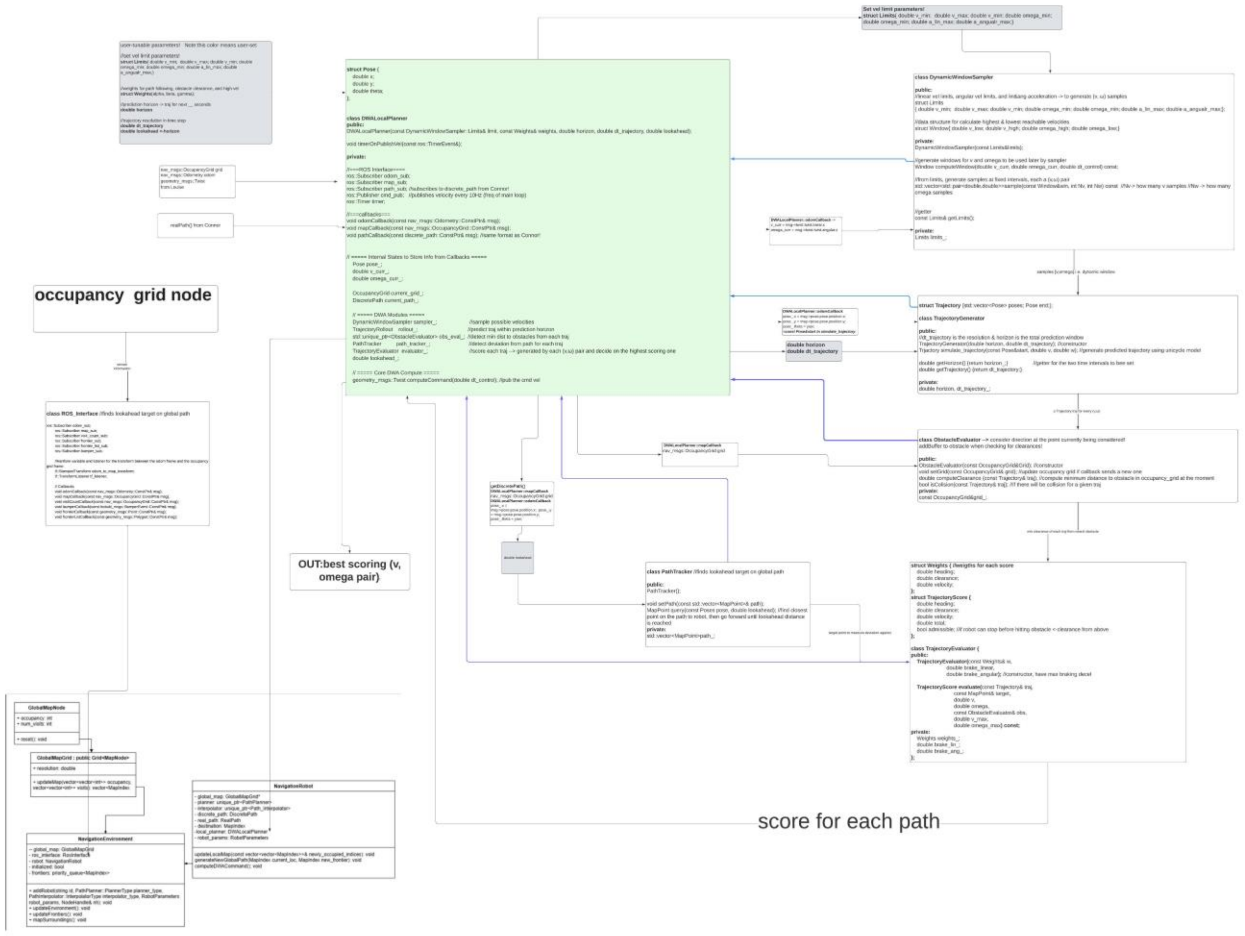
[6] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," IEEE Robotics & Automatation Magazine, vol. 4, no. 1, pp. 23–33, Mar. 1997.

10. Appendices

10.1 Global Path Planning UML Class Diagram


Grid<T>
# node_grid: vector<vector<T>>
+ getNode(int row, int col): T&
SearchGrid: public Grid<SearchNode>
+ getNode(int row, int col): SearchNode*
+ getNeighbors(int row, int col): vector<SearchNode*>
+ reset(): void
SearchNode
- gCost: double
- hCost: double
- row: int
- col: int
- closed: bool
- parent: SearchNode*
+ setParent(SearchNode* new_parent): void
+ reset(): void
+ closeNode(): void
RobotLocalMap : public Grid<double>
- minimum_clearance: double
- global_map; GlobalMapGrid*
+ isAccessible(int row, int col): bool
+ getNearestAccessibleNode(MapIndex idx): MapIndex
+ brushFireUpdate(const vector<MapIndex>& sources): void
GlobalMapNode
+ occupancy: int
+ num_visits: int
+ reset(): void
GlobalMapGrid : public Grid<MapNode>
+ resolution: double
+ updateMap(vector<vector<int>> occupancy, vector<vector<int>> visits): vector<MapIndex
NavigationRobot
- global_map: GlobalMapGrid*
- planner: unique_ptr<PathPlanner>
- interpolator: unique_ptr<Path_Interpolator>
- discrete_path: DiscretePath
- real_path: RealPath
- destination: MapIndex
-local_planner: DWALocalPlanner
- robot_params: RobotParameters
updateLocalMap(const vector<vector<MapIndex>>& newly_occupied_indices): void
generateNewGlobalPath(MapIndex current_loc, MapIndex new_frontier): void
computeDWACommand(): void
PathPlanner
# search_grid: SearchGrid
# local_map: RobotLocalMap*
# global_map: GlobalMapGrid*
generatePath(MapIndex start, MapIndex end) = 0: virtual vector<MapIndex>
AStarPlanner: public PathPlanner
- visit_sensitivity: double
+ generatePath(MapIndex start, MapIndex end): vector<MapIndex> override
NavigationEnvironment
-- global_map: GlobalMapGrid
- ros_interface: RosInterface
- robot: NavigationRobot
- initialized: bool
- frontiers: priority_queue<MapIndex>
+ addRobot(string id, PathPlanner::PlannerType planner_type, PathInterpolator::InterpolatorType interpolator_type, RobotParameters robot_params, NodeHandle& nh): void
+ updateEnvironment(): void
+ updateFrontiers(): void
+ mapSurroundings(): void
RealPath
- path_coordinates: vector<MapPoint>
- discrete: DiscretePath*
- interpolator: PathInterpolator*
- time_stamp: time_t
+ updatePath(): void
PathInterpolator
# path_resolution: double
+ interpolate (const vectort<MapIndex>& input_path) = 0: virtual vector<MapPoint>
LinearInterpolator: public PathInterpolator
+ interpolate (const vectort<MapIndex>& input_path): vector<MapPoint> override
DiscretePath
- path_coordinates: vector<MapIndex>
- time_stamp: time_t
+ updatePath(const vector<MapIndex>& new_path_coordinates): void
+ refinePath(): void

10.2 Local Path Planning Class Diagram


occupancy grid node
class ROS_Interface
class DWALocalPlanner
struct Pose
class DynamicWindowSampler
struct Trajectory
class TrajectoryGenerator
class ObstacleEvaluator
class PathTracker
struct Weights
struct TrajectoryScore
class TrajectoryEvaluator
double horizon
double dt_trajectory
OUT:best scoring (v, omega pair)
GlobalMapNode
GlobalMapGrid
NavigationRobot
NavigationEnvironment
score for each path