Autonomous Navigation Of A Simulated Robot

Using The ROS MoveBase Node

Name: Roberto Akankwasa Sanchez

Student ID: 22012351

ABSTRACT

Autonomous navigation is a vital aspect of robotics, with applications in manufacturing, security, agriculture, and logistics. However, integrating various modules to achieve autonomy can be challenging. Therefore, this project focused on implementing and evaluating an existing solution for autonomous navigation proposed by the Robot Operating System (ROS). Specifically, it aimed to utilize the MoveBase node to enable a mobile robot to navigate from its current location to a target goal position on a known map while avoiding obstacles in its path.

The TurtleBot Burger Model served as the platform for mapping the environment and testing localization and path planning. For path planning, the A* algorithm was employed, while the Dynamic Window Approach (DWA) was used for obstacle avoidance. Additionally, a particle filter, Adaptive Monte Carlo Localization (AMCL) was implemented for localization. The implementation of the MoveBase node was conducted using ROS, and all tests were carried out in the Gazebo simulation environment. RViz was used to visualize the processes of path planning, localization, and map-building.

The significance of this project was in the investigation of the efficiency of the MoveBase package proposed by ROS for achieving autonomous navigation. The performance of MoveBase node was evaluated in a simulated environment and provided valuable insights into its effectiveness and reliability.

Key words: Robot Operating System (ROS), MoveBase, Adaptive Monte Carlo Localization (AMCL), Dynamic Window Approach (DWA), gmapping, A *, Dijkstra, costmap, local planner, global planner.

I. INTRODUCTION

A. Background and motivation

Autonomous robots are increasingly being used in various industries such as transport, manufacturing, and even day-to-day domestic activities. These robots can perform tasks with little to no human intervention and utilize algorithms and sensors to navigate and interact with the environment.

To achieve autonomy in robotics multiple features, need to be considered simultaneously such as path planning, sensor integration, localization, obstacle avoidance, robot control, etc. However, implementing each of the modular features into one system tends to lead to errors during the integration phase with many systems failing to achieve autonomy.

This is where the Robot Operating System (ROS) comes in, providing the necessary tools that allow for the development and testing of robots in a simulated environment that closely mimics real-world conditions as seen in Figure 1. One of the crucial tools provided by ROS is the MoveBase node in the navigation stack. This tool allows a robot to move from its current location to goal pose while avoiding any obstacles which essentially is autonomous navigation.

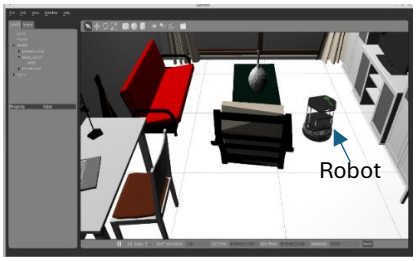

Figure 1: TurtleBot in Demo environment

B. Objectives

For this project, an autonomous robot was developed using ROS, Gazebo, and TurtleBot as the primary robot model. The robot was equipped with path planning and obstacle avoidance capabilities. The MoveBase node was a crucial component in implementing the navigation system. The main objectives of the project were as follows:

- Create a virtual environment that accurately simulates the robot's operational setting:
- Generate map of the environment
- Implement a path planning algorithm to help the robot navigate to reach target points.
- Implement obstacle avoidance to ensure that the robot followed a collision free path
- Implement a localization algorithm that uses sensor data to continuously update the robot's position.
- Testing and evaluation of the system's efficiency in different known environments

C. Scope and limitations

The project focused on developing a simulation-based autonomous robot designed to navigate around a simulated environment. The primary areas of investigation included the design and implementation of navigation and localization algorithms, the integration of obstacle avoidance, and the evaluation of the robot's performance in various simulated scenarios.

The study encompassed the following key aspects:

- **Navigation algorithms**. Implementation of pathfinding algorithms like A* and Dijkstra's for efficient path planning
- **Obstacle avoidance.** After a path was planned, new obstacles were introduced to evaluate the robot's ability to adapt its path based on the changes.
- **Localization Algorithm.** Implementation of an algorithm to enable real-time position updates and accurate tracking within the simulated environment.

However, the project excluded aspects of physical robot hardware and real-world testing. The study focused on the autonomous capabilities of the robot in a controlled simulated environment using the MoveBase node.

II. LITERATURE REVIEW

Mapping

To navigate autonomously, map of the robot's environment was needed. This map was used for tasks such as path planning, localization, and obstacle avoidance.

In most cases, a map was created from scratch. To do so, **SLAM (Simultaneous Localization and Mapping)**, [1] was needed, which aided in building a map of the unknown environment while also keeping track of the location of the robot on that map.

ROS provided a series of mapping packages that were made available to create maps of the environment.

- **OctoMap**. Mainly used for 3D mapping and generated a detailed map of the environment. [2]
- **Hector SLAM**. This was a feature-based algorithm that detected and tracked features

in the robot's environment to build a 2D map using the Lidar data. [3]

- **Gmapping**. This created a 2D occupancy Grid map using the laser and pose data from the robot. [4]

Navigation

Various algorithms exist including A* [5], and Dijkstra [6], that have aided with path planning in robotics. A* was the most used for its efficiency in finding the shortest path to a target goal position. Dijkstra algorithm, although optimal poses to take a toll computationally.

In ROS, development of autonomous robots was done using the MoveBase Node which was part of the navigation stack. The main function of this node was to move the robot from its current position to the target position.

In [7] , the author mentioned the key components, **global and local planners, recovery behaviors, and cost maps** needed by the MoveBase node to perform its task, and when a goal pose was received the node it used these components to generate a velocity command as output which was sent to the /cmd_vel topic of the robot.(Figure 2)These commands caused the robot to move.

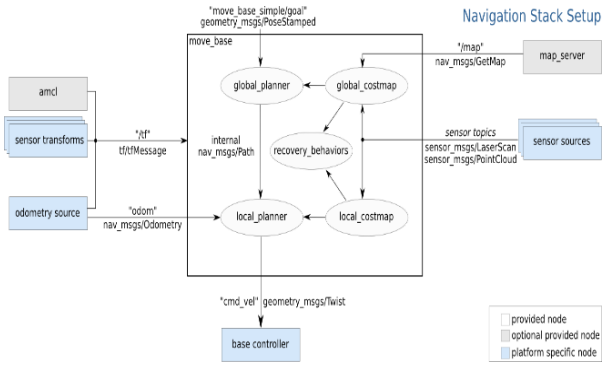

Figure 2:Interaction between the different components required for autonomous navigation

Global and local planner

In the **move_base node,** once a new target goal position was received it was sent to the global planner that was responsible for calculating safest path to that target goal position. The global planner determined the path before the robot began moving which meant that any unexpected obstacles during motion were not accounted for. Commonly used global planners included the Dijkstra and the A*.

The path planned by the global planner was then sent to the local planner that was responsible for producing the velocity commands to send to the robot's mobile base. It considered the **laser and odometry** readings and executed segments of the global path plan. In [8], the authors go through how the global and local planners were integrated to allow the robot to adjust its path and keep the robot from colliding with obstacles while still striving to reach the goal pose.

Commonly used local planners included **Dynamic Window Approach** (DWA) **local planner**, Figure 3,which evaluated multiple potential trajectories based on samples of possible velocities. The best trajectory was chosen based on obstacle avoidance, movement in the direction of the goal pose, and speed efficiency.

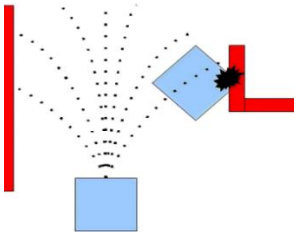

Figure 3: Demonstration of DWA

Another local planner is the **Time Elastic Band (TEB) local planner** (Figure 4)used a method called elastic band to deform the robot's path to avoid obstacles. The trajectory was optimized in both time and space to achieve faster and more efficient navigation.

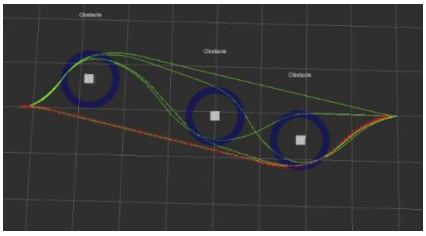

Figure 4: Demonstration of TEB

Cost maps

A cost map was a type of map that represented areas with free space and areas that are obstacles. Each cell of a cost map had an integer value in the range of 0-255. [9]

There were two types of cost maps: a global cost map created from a **static map** and a local cost map created from **sensor readings**. The global cost map was used by the global planner, and the local cost map was used by the local planner.

Recovery behaviours

During navigation, the robot might encounter an error or get stuck. The ROS navigation stack provided a solution to the problem using recovery behaviours

and in [10] , the author provided an overview of how they allow the robot to get unstuck and continue with its navigation path.

Clear cost map removed obstacles within a specific region of the robot's map which entailed that the local map took up the same state as the global cost map. Whereas the **Rotate recovery** attempted to rotate the robot by 360 degrees to allow the robot to find an obstacle-free path.

Localization

During navigation, a robot needed to know where it was. Localization enabled the robot to determine its position within its environment. Some of the common techniques include.

Kalman filter

This was a parametric Bayes filter that utilized recursion to estimate the state of a system and provided an optimal solution. The Kalman filter operated under the 3 assumptions; that all distributions were Gaussian, motion and observation models were linear, and the state space of the system was continuous. [11]

However, in robotics, motion was non-linear which rendered the Kalman filter impractical. This was where extensions such as **the Extended Kalman filter (EKF)** which linearised non-linear functions and the **Unscented Kalman filter (UKF)** that took sigma points to approximate the Gaussian distribution. Both the EKF and UKF were suitable for non-linear systems, which in this case robotics.

Particle filter

This non-parametric Bayes filter utilized a set of particles to represent the posterior state. The particle filter used a set of weighted particles, each representing a hypothesis of the robot's state. [12]

Firstly, a prediction was made based on the motion model, then using the observations model, the particles were weighted, and an update was made. The particles were then resampled based on their **importance factor**. The density of the particles approximated the posterior state.

METHODOLOGY

Simulated Environment

Using Gazebo, three sets of custom environments were created. A single environment was used during the initial mapping, path planning, and localization process; however, during the testing phase, two other environments were mapped and used to test the robot.

The two new environments were created to add different levels of complexity to the maps which were then used to test the MoveBase node

Robot model

Initially, an attempt was made to design a custom URDF file for the robot model as seen in Figure 5 . However, a series of errors were encountered when the ROS Teleop package was being used to manually control the robot.

Since Robot design was not one of the main objectives of this project, the existing robot model, TurtleBot3, was utilized.

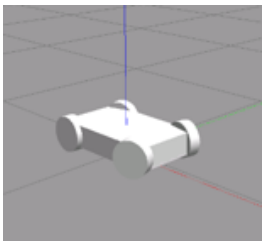

Figure 5: Initial Robot model

The TurtleBot robot burger model (Figure 6) was used because it was equipped with the necessary sensors, which were ideal for mapping, navigation, and localization. The TurtleBot was also a differential drive robot and consisted of two independent wheels which made navigation easier.

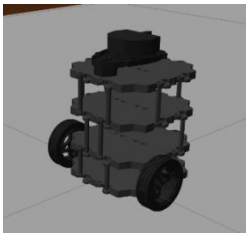

Figure 6: TurtleBot, Burger

Mapping

The ROS (Robot Operating System) navigation stack included the **gmapping package**, which implemented a SLAM algorithm. The slam_gmapping node from the gmapping package was used to create a 2D map by reading data from the robot's laser and its transforms, converting this information into an Occupancy Grid Map.

Once the slam_gmapping node was launched, the Teleop package was used because it allowed for navigation around the environment to build the map.

The 2D occupancy grid map built was sufficient because it captured the necessary spatial information for the robot, which was operating in a 2D flat environment. This

Navigation

The MoveBase node was responsible for the navigation of the robot, it took the goal pose and outputs the necessary velocity commands to navigate the robot from its current position to the goal pose. However, to achieve autonomous navigation, the following yaml files were prepared and fine-tuned to suit the navigation system.

- global_planner parameters
- local planner parameters
- common_costmap parameters
- global_costmap parameters
- local_costmap parameters

Localization

To achieve localization in ROS the AMCL (Adaptive Monte Carlo Localization) package was used. This package provided the **amcl node** that used the Monte Carlo Localization (MCL) algorithm, to keep track of the robot in a 2D space. [13]. The AMCL node subscribed to data from the laser, the map and the transforms from the robot then published the robot's estimated position in the map.

Similar to the MoveBase node, the AMCL node had set of parameters such as the minimum and maximum particles that were configured and fine-tuned to achieve localisation.

IMPLEMENTATION

Robot Model and Environment

The TurtleBot Burger model was used to implement the autonomous navigation of this project. All its features such as the wheel descriptions and the lidar were defined in the urdf file. The simulation environment was created in Gazebo and consisted of walls. The object placements and environment descriptions were all defined in the .**world** file.

A launch file was used to launch the TurtleBot robot model and start Gazebo with the custom simulation environment. The robot was spawned at a declared start position.

Mapping

The 2D occupancy Grid map was created using slam_gmapping node from the gmapping package based on LIDAR scans from the TurtleBot.

The configuration of the parameters in Table 1 to use slam_gmapping was important to generate a proper map. [14] Without a proper map, the robot would not be able to navigate the environment. Some of the key requirements to build a proper map were good laser and odometry data. With this data, the slam_gmapping node then transformed each laser reading to the Odom frame.

To map the environment, gmapping was launched with the TurtleBot and the teleoperation package from ROS. The robot was moved through the environment. During this process, the map was being built and was visible using Rviz. The **map_server node** was used to save the current map being built.

Table 1: List of the relevant Gmapping parameters

Parameter	Value	Description
Base_frame	base_link	Frame for robot reference
Odom_frame	odom	Frame for odometry data
Scan_topic	scan	Topic for lidar scans
maxUrange	6.0	The max usable lidar range for gmapping
maxRange	8.0	The max range of lidar that was considered
Map_update_interval	2.0	How often was the map updated
linearUpdate	0.2	The linear motion required before the map was updated
angularUpdate	0.4	The angular rotation required before the map was updated
resampleThreshold	0.5	Determined when particles were resampled
particles	90	Defined the number of particles used in SLAM
Delta	0.05	Defined the grid resolution of the map
xmin ymin xmax ymax	-10.0 -10.0 10.0 10.0	Defined the map boundaries

Localization

A separate package was created which contained a launch file that launched the AMCL node . Within this launch file the map_server node was started to provide the generated map to the AMCL node. A series of parameters were configured as seen in in Table 2. These parameters affected how the AMCL node processed the sensor data and updated the robot's pose.

Table 2: List of relevant parameters used for localization

Parameter	Value	Description

use_map_topic	true	Indicated that the AMCL subscribed to the /map topic
odom_model_type	Diff	Indicated the robot had a differential drive
odom_frame_id	Odom	Indicated the frame with odometry.
min_particles	500	Maximum particles used for the filter
max_particles	2000	Minimum particles used for the filter
update_min_d	0.25	The linear distance that the robot had to move to perform a filter update.
update_min_a	0.2	The angular distance that the robot had to move to perform a filter update.
resample_interval	1	Defined how often particles were resampled
laser_max_beam	60	Number of lidar beams the node sampled per scan
laser_max_range	12.0	The max range of lidar the AMCL considered
laser_z_hit	0.5	Probability of a correct measurement
laser_z_max	0.05	Probability of a max range reading in the event of no obstacles seen
laser_z_rand	0.5	Probability of a random measurement

Navigation

The A* path planning algorithm was implemented to ensure that the shortest path was found. Compared to Dijkstra which considered all possible paths to the goal, A* took into consideration the general direction of the target goal position, which made the path planning process quicker and more efficient.

To integrate the A* algorithm into the system, the **MoveBase** node was implemented, which handled the path planning and execution of movement to follow the planned path. The following configuration files were set up to implement the **MoveBase** node.

Global planner

The global_planner was configured to implement the A* algorithm. The necessary parameters were configured as seen in Table 3 which allowed the A* algorithm to be implemented and further optimised.

To implement the A* algorithm, the use_dijkstra parameter was set to false, which meant that Dijkstra was not used but rather A*.

Table 3: List of relevant parameters for the global planner.

Parameter	Value	Description
allow_unknown	false	Determined whether the planner was allowed to generate a path through unknown areas
use_dijkstra	false	Specified whether to use Dijkstra or not
default_tolerance	0.1	Defined the distance from the target that was considered as “reaching the goal”
use_quadratic	true	Determined the cost calculation model used by the A*
use_grid_path	false	Specified whether the planner followed the grid lines or free space

Local planner

The Dynamic Window Approach (DWA) was used because it ensured that the robot dynamically modified the planned path in the event of new obstacles. The list of parameters in Table 4 were configured and fine-tuned to ensure that the robot took the best collision-free path.

Table 4: List of relevant parameters for the local planner

Parameter	Value	Description
acc_lim_x	2.5	The maximum linear acceleration
acc_lim_theta	2.2	Max angular acceleration
max_vel_x	1.0	Max linear velocity
min_vel_x	0.0	Min linear velocity
max_vel_theta	1.0	Max angular velocity
min_vel_theta	-1.0	Min angular velocity
holonomic_robot	false	The robot could not move sideways, hence the value false
yaw_goal_tolerance	0.1	Rotation of the robot from the goal position
xy_goal_tolerance	0.2	Distance of the robot from the goal position

latch_xy_goal_tolerance	false	Indicated that the robot would continuously satisfy the goal tolerance.
dwa	true	Indicated that DWA was used

Global cost map

The global costmap was created using the data from the static map. Some of the key parameters, Table 5, that were configured included the rolling_window, which was set as false, which ensured that it was initialized from the static map.

Table 5: List of global costmap parameters

Parameter	Value	Description
global_frame	map	Set the coordinate frame for the global costmap
rolling_window	false	Indicated that the costmap was initialized from the static map

Local cost map

The local cost map was built using the Lidar data from the Lidar data from /scan topic, hence why the rolling_window parameter for the local costmap was set to true. Table 6

Table 6: List of local costmap parameters

Parameter	Value	Description
global frame	Odom	Set the coordinate frame for the local costmap
rolling_window	true	Indicated that the costmap is not initialized from the static map

Common cost map

This was a combination of both the local and global cost map parameters, Table 7. It ensured consistency in the global and local cost maps.

The costmap layers defined how the map handled static and dynamic obstacles. The static layer, Table 8 ensured that the robot used the preloaded map for navigation. The obstacle layer, Table 9 utilised real-time Lidar data in order to identify new obstacles in the map. The inflation layer, Table 10 was utilised because it ensured that a safety zone was applied around the obstacles to prevent collisions.

Table 7: List of common costmap parameters

Parameter	Value	Description
robot_radius	0.4	Defined the circular approximation of the robot's size, TurtleBot
footprint_padding	0.1	Defined the size of the padding added to provide a safety margin when near obstacles
robot_base_frame	base_link	Indicated the TF frame for the robot's base
update_frequency	5.0	The rate at which the costmap updated based on sensor data
publish_frequency	5.0	The rate at which the costmap published updates
transform_tolerance	0.3	The max delay allowed between transform updates
resolution	0.05	Defined the grid cell size

Table 8: List of parameters for the static layer

Parameter	Value	Description
map_topic	/map	Indicated that /map topic was used for navigation
subscribe_to_updates	True	Allows the static map to get real-time updates.

Table 9: List of parameters for the obstacle layer

Parameter	Value	Description
observation sources	laser	Indicated that the laser was used for obstacle detection
laser	**data_type**:LaserScan **clearing**: true **marking**: true **topic**: scan **inf_is_valid:** true **obstacle_range**: 4.5	All data was read from the /scan topic, and the lidar was used to mark new obstacles. The max range for detecting obstacles was also defined.

Table 10: List of parameters for the inflation layer

Parameter	Value	Description

Inflation_radius	0.2	Defined the zone around obstacles that prevented the robot from getting too close.
Cost_scaling_factor	5.0	Controlled how quickly the cost increased near an obstacle

A launch file was created to launch the whole system and all the different parameters from the local and global planners and the local, global and common cost maps were loaded.

RESULTS AND DISCUSSION

Environment development results

The building editor from Gazebo was used to create simulation environments for example in Figure 7 as well as the test environments. Gazebo provided tools and additional props that were used as obstacles.

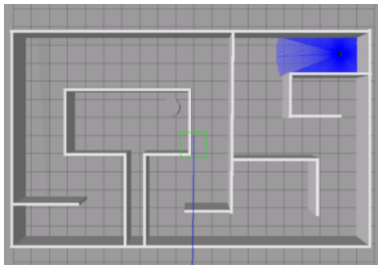

Figure 7:Initial simulation environment

Mapping results

Mapping was critical to achieve autonomous navigation. The map in Figure 8 was generated using gmapping which utilized the Lidar data from the **/scan** topic to determine whether a space was free or occupied.

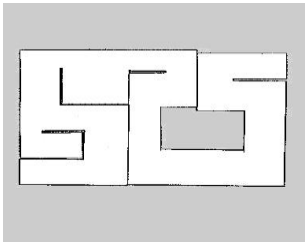

Figure 8: Map of the first environment

The TurtleBot was manually navigated around the simulated environment and a 2D occupancy grid map was accurately built and represented the simulated environment. However, increasing the range and number of Lidar beams used could have improved the mapping accuracy but would slow down the mapping process due to the computational load.

Localization results

The localization was achieved using the Adaptive Monte Carlo Localization node (AMCL) from the ROS Navigation stack. It was used to estimate the robot's position in real-time.

When the Node was launched, a series of random guesses known as particles were generated, as seen in Figure 9. Each particle represented a possible estimate of the robot's current pose. As the robot observed its environment and took readings via the Lidar, it eliminated the particles that did not match the readings, and those particles that looked more probable were given higher weights and resampled, which led to a refined estimate of the robot's position.

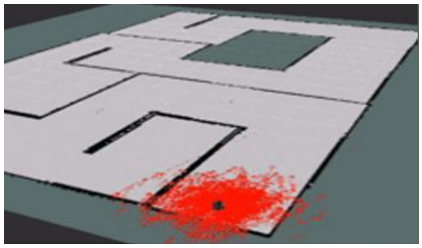

Figure 9: Robot's estimated pose before movement

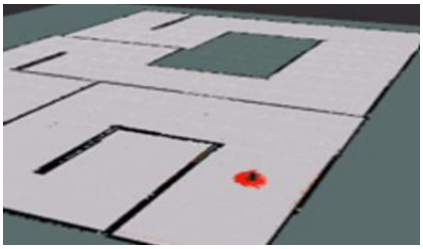

Figure 10: Robot's estimated pose after movement

The process led to most of the particles to converge in the most probable pose of the robot as seen in Figure 10. Essentially, the more the robot moved, the more the data was collected from the lidar, hence the higher the localization precision. However, the AMCL node struggled to estimate the robot's pose in environments with little to no features to be used as observations.

Path planning with the local and global planner

The 2D Goal pose tool from rviz was used to send a goal pose to the MoveBase node. When received, the global planner planned and generated the safest and fastest path from its current position to the target goal as seen by the **green line** in Figure 11.

Once the global planner determined the best possible path, that path was sent to the local planner. The local planner executed segments of the global path and provided velocity commands to move the robot along that path as seen by the **red line** in Figure 11.

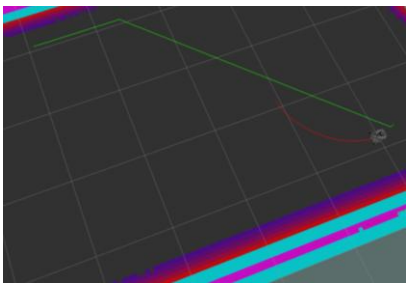

Figure 11: Example of global path and local path execution

In certain situations where the goal pose was significantly far from the robot's position, the global planner was unable to determine a viable path, which was caused by slightly restrictive costmap configurations and limited time for a path to be determined before the process was aborted.

Testing with other environments

A series of different known maps of environments with different levels of complexity were used to test the MoveBase node, as seen in Figure 12 and Figure 13. The robot struggled to navigate in environments with more complexity because it encountered difficulties such as narrow passages and many obstacles, which prevented the local planner from finding a feasible trajectory. As a result, the navigation process was aborted.

In simple environments (Figure 12) where obstacles were fewer, the robot successfully navigated from its current position to the goal pose

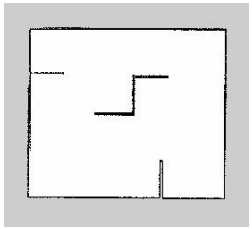

Figure 12: Map of Second Environment

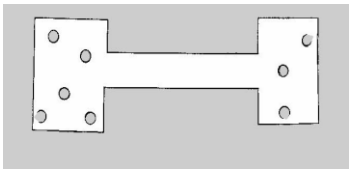

Figure 13: Map of Third Environment

Obstacle Avoidance results

To test the obstacle avoidance capabilities of the robot, an obstacle that was not in the global cost map was introduced into the environment as seen in Figure 14.

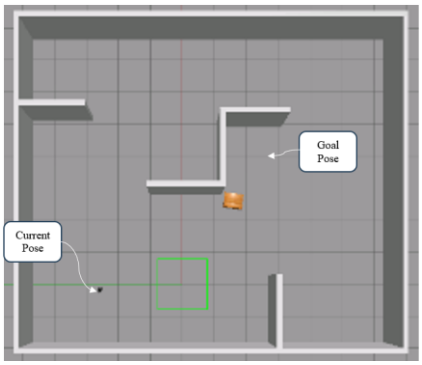

Figure 14: Environment with obstacle introduced

Using the Dynamic Window Approach local planner, the robot successfully navigated around obstacles to maintain a collision free path as seen in Figure 15 where the path was planned, Figure 16 where an obstacle was encountered while navigating along that path and Figure 17 where the robot navigated around the obstacle.

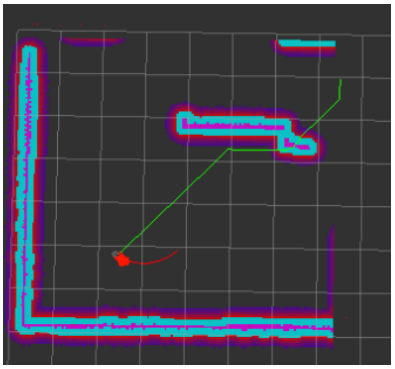

Figure 15: Path planned

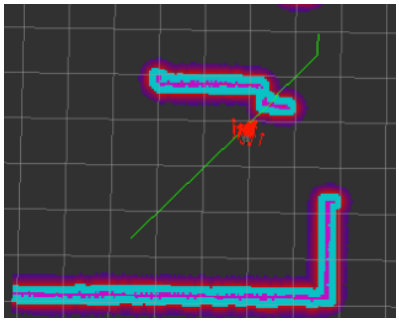

Figure 16: Obstacle encountered along global path

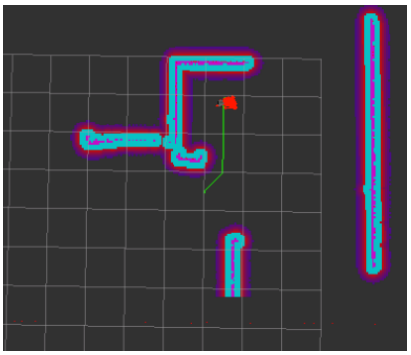

Figure 17: Obstacle Avoided

However, as more obstacles were added, which increased the complexity of the environment, the robot's navigation performance slowly began to degrade

In such situations the velocity and acceleration parameters of the local planner (Table 4) required further tuning to allow the robot to navigate around obstacles more effectively while closely following the global path.

Planned path vs Actual path Analysis

Figure 18 illustrated the planned path (blue) and the actual path (orange) that was followed by the robot. The planned path represented the ideal trajectory based on the A* algorithm (global planner), while the actual path represented how the robot moved while considering obstacles in the environment.

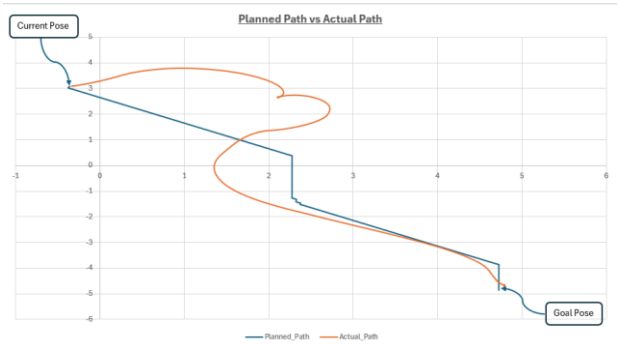

Figure 18: Planned Path vs Actual Path

The deviations from the planned path resulted from the trajectory adjustments made by the DWA (Dynamic Window Approach) local planner, which generated real-time motor commands to navigate to the set goal pose while avoiding any obstacles.

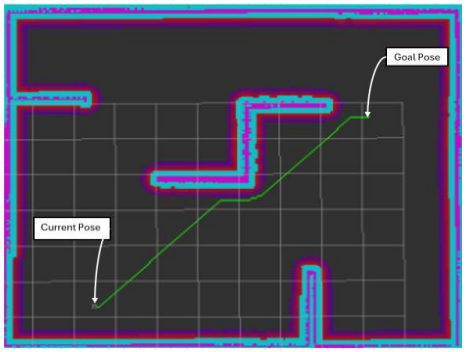

Figure 19: Robot Navigation path in map

The robot successfully navigated and reached its goal despite its deviations from the planned path, as seen in Figure 19. The DWA planner effectively avoided obstacles in its path; however, the deviations from the planned path significantly increased the total distance travelled to the goal. The local planner required further parameter tuning to minimize the deviations and improve path efficiency.

CONCLUSION AND FUTURE WORK

Summary

The project successfully implemented the MoveBase node to achieve autonomous navigation using ROS, Gazebo, and the TurtleBot. The A* path planning (global planner), DWA local planner, AMCL node, global and local costmaps and the occupancy grid map were integrated to allow the robot to navigate through its environment. The system was tested in Gazebo, with a series of known maps where the TurtleBot successfully followed its planned paths and avoided obstacles, and the process was visualized in Rviz. The performance of the system was relatively effective, however the increase in the complexity of the environment led to a decrease in the robot's navigation and obstacle avoidance capabilities. Tackling this problem would require more parameter tuning of the MoveBase components to improve on how the robot perceived its environment in order to achieve optimal navigation.

Lessons learned

Navigation using the MoveBase node from ROS required extensive parameter tuning to achieve smooth navigation. A clear understanding of the relevant parameters of each of the elements such as the global and local costmaps, the local and global planners, and the AMCL was a necessity to fine-tune the parameters and improve the navigation. The costmaps played a crucial role in the safe and obstacle-free navigation of the robot. They influenced how the robot perceived its environment. However, real-world deployment would allow the introduction of new challenges, such as sensor noise to further evaluate the efficiency of the MoveBase Node.

Potential future improvements

With the current system based around a simulation, there were a few potential future elements that could be implemented to further explore autonomous navigation in robotics.

- **Real-world implementation**. In the future, the simulation could potentially transition to a physical robot to allow for real-world testing and evaluation.
- **Implement Multiple robots**. Implementing multiple robots would be another area of autonomy that could be explored. This could consist of a fleet of robots coordinating to perform a series of tasks autonomously.

References

[1] P. Robertson, “Introduction to SLAM Simultaneous Localisation and mapping,” in *Introduction to SLAM Simultaneous Localisation and mapping*, MIT, 2005, pp. 11-20.

[2] AdamAllevato, “octomap,” 12 05 2021. [Online]. Available: https://wiki.ros.org/octomap.

[3] M. Almeida, “hector_mapping,” 24 03 2021. [Online]. Available: https://wiki.ros.org/hector_mapping.

[4] GvdHoorn, “gmapping,” 04 02 2019. [Online]. Available: https://wiki.ros.org/gmapping.

[5] V. Garikina, “A comprehensive Guide to A* Search algorithm,” 2024.

[6] A. I. Aramendia, “A Guide to Dijkstra's Algorithm. All you need,” *Medium,* 2023.

[7] BotWhisperer, “movebase-ros,” 20 08 2024. [Online]. Available: https://therobotcamp.com/2024/08/20/movebase-ros/.

[8] Pablo Marin-Plaza, Ahmed Hussein, David Martin and ArturodelaEscalera, “Global and Local Path Planning Study in a ROS-Based Research,” *Hindawi,* pp. 2-3, 22 02 2018.

[9] NickLamprianidis, “costmap_2D,” 10 01 2018. [Online]. Available: https://wiki.ros.org/costmap_2d.

[10] A. Patil, “Recovery Behaviours used in Navigation Stack - ROS,” *Medium,* 21 04 2024.

[11] Sebastian Thrun, Wolfram Burgard and Dieter Fox, “Probalistic Robotics,” in *Probalistic Robotics*, Cambridge, Massech, MIT press, 2005, pp. 33-65.

[12] Sebastian Thrun, Wolfram Burgard and Dieter Fox, “Probalistic Robotics,” in *Probalistic Robotics*, Cambridge ,Massachusetts, MIT Press, 2005, pp. 77-90.

[13] Dieter Fox, Wolfram Burgard, Frank Dellaert and Sebastian Thrun, “Monte Carlo Localization: Efficient Position Estimation for Mobile Robots,” Carnegie Mellon University, Pittsburgh, 1999.

[14] Zuhair A. Ahmed and Safanah Mudheher Raafat, “An Extensive Analysis and Fine-Tuning of Gmapping's Initialization Parameters,” *ResearchGate,* p. 130, 2023.