Intelligent Path Planning Using A*

Final Project - RXR

Group Members

Haris Mashood - 2413298

Ibrahim Ajagbe - 2417387

Objective

The objective of this project was to enhance the path planning capabilities of the Andino robot simulated in a ROS 2 environment. Previously, as part of Learning Assignment 5 [1], Andino relied on a simple straight-line planner that directly connected the start and goal points without considering obstacles. While straightforward, this approach could not navigate around objects or find an optimal route when the direct path was blocked.

We addressed these limitations by implementing a more intelligent path-planning algorithm. By incorporating the A* algorithm, Andino can efficiently search for a path that navigates around obstacles and finds an optimal or near-optimal route from the start point to the destination.

A* Algorithm

The A* algorithm is a popular and well-established search algorithm used extensively in robotics and autonomous navigation. It extends Dijkstra’s algorithm by incorporating a heuristic that estimates the cost to reach the goal, enabling A* to find efficient paths more quickly than brute-force methods [2].

We chose A* over alternatives because it balances optimality and efficiency well. While guaranteed to find the shortest path, Dijkstra's algorithm does not use a heuristic and can be slower in large spaces. RRT and other sampling-based methods may find a path but are not inherently optimal and may introduce unnecessary complexity [3]. A* is both informed and optimal for many problem spaces, making it a solid fit for our relatively structured and grid-based costmap environment.

Code Overview

A* operates on a graph or grid by maintaining two main sets of nodes:

- **Open list:** Contains nodes that are candidates for exploration.
- **Closed list:** Contains nodes that have already been explored.

At each step, A* picks the node from the open list with the lowest combined cost of the path taken so far and the heuristic estimate to the goal. This allows A* to “guess” which paths will likely lead to the best solution faster.

Pseudo Code

```
function A*(start, goal):
    open_set = priority queue containing start
    came_from = empty map
    g_score[start] = 0
    f_score[start] = heuristic(start, goal)

    while open_set is not empty:
        current = node in open_set with lowest f_score
        if current == goal:
            return reconstruct_path(came_from, current)

        remove current from open_set
        for neighbor in neighbors(current):
            tentative_g = g_score[current] + distance(current, neighbor)
            if neighbor not visited or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristic(neighbor, goal)
                if neighbor not in open_set:
                    add neighbor to open_set

    return failure
```

Explanation

Initialization:

Put the start node into a priority queue (*open_set*) and set *g_score[start]* to 0. The *f_score[start]* is calculated as *g_score[start] + heuristic(start, goal)*.

Main Loop:

While there are nodes to explore, pick the one with the lowest *f_score* from *open_set*. If it's the goal, we're done and can reconstruct the path.

Exploring Neighbors:

For each neighbor of the current node, calculate a tentative cost (*tentative_g_score*). If it's better than any previously recorded cost:

- Update *came_from[neighbor]* to remember the path.
- Update *g_score[neighbor]* and *f_score[neighbor]*.
- Add the neighbor to *open_set* if not already there.

If No Path Found:

If we run out of nodes in *open_set* without reaching the goal, no path exists.

Integration with Existing Code

The original solution from Learning Assignment 5 provided a simple create_straight_plan function that returned a direct path from start to goal. This was sufficient for open spaces but failed when obstacles were present.

To integrate A*, the straight-line planner was replaced with the create_a_star_plan function, which:

- Retrieves the global costmap from the **BasicNavigator** interface.
- Converts the costmap's occupancy data into a 2D grid representation that A* can process.
- Computes grid coordinates corresponding to the robot's start and goal poses, accounting for costmap resolution and origin offsets.
- Runs the A* algorithm on this grid to find a navigable path around obstacles.
- Converts the resulting sequence of grid cells back into world coordinates and assembles a Path message that the navigation stack can use.

The planner is no longer restricted to a basic straight-line solution by making these changes. Instead, it actively searches for a collision-free, feasible route, enabling the robot to navigate more complex scenarios.

Testing the A* Algorithm

The algorithm was tested with various navigation goals to test its functionality and ability to create efficient navigation paths. The results varied in different scenarios;

for example, in the figure below, the Andino was given a Nav 2 goal at the other end of the warehouse, which was aborted after a couple of seconds.

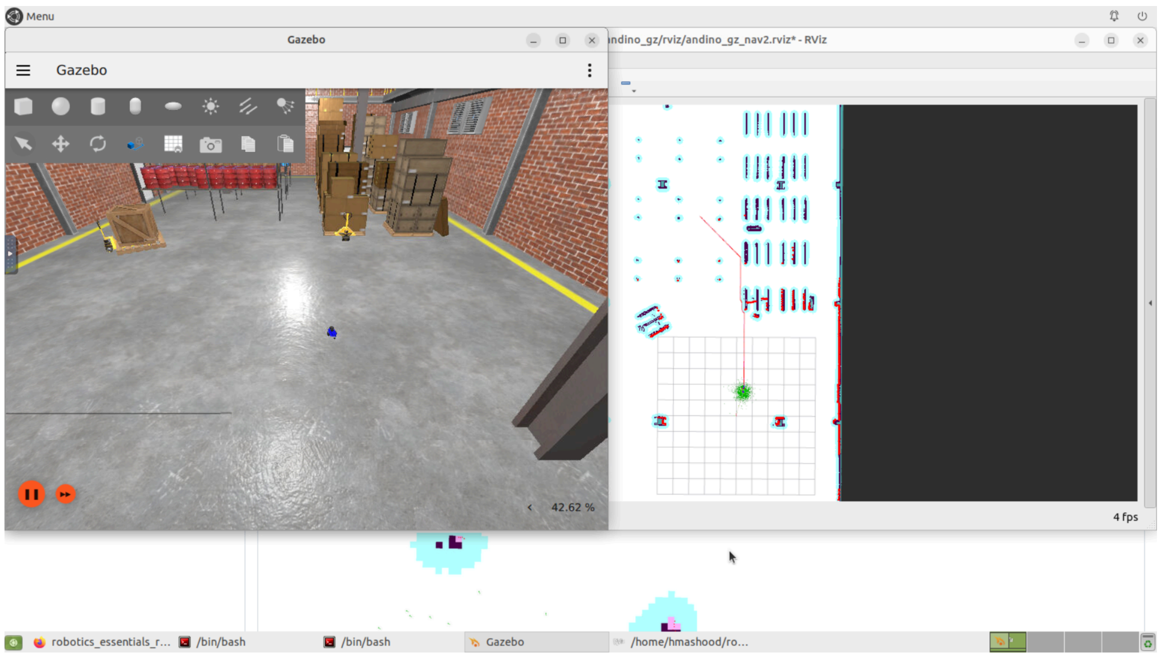
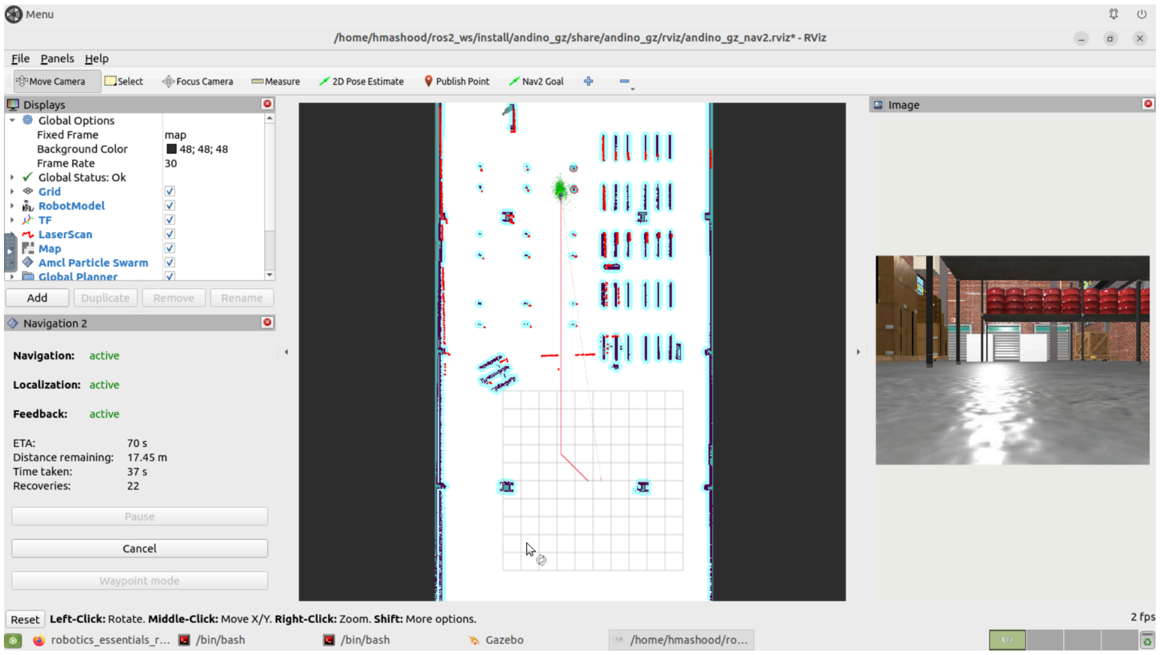


Figure 1: Aborted Nav 2 Goal

But the Andino took the same path when the Nav2 goals were given in succession for smaller distances until the last end of the warehouse. When the Andino reached its destination, the Nav goal for the starting point was given, and the Andino was able to create a plot of a path to the starting point.

While the path was mapped, obstacles were placed in Andino's path to see if it could detect and map a new path, avoiding the obstacles. The obstacles can be seen in Figure 2.



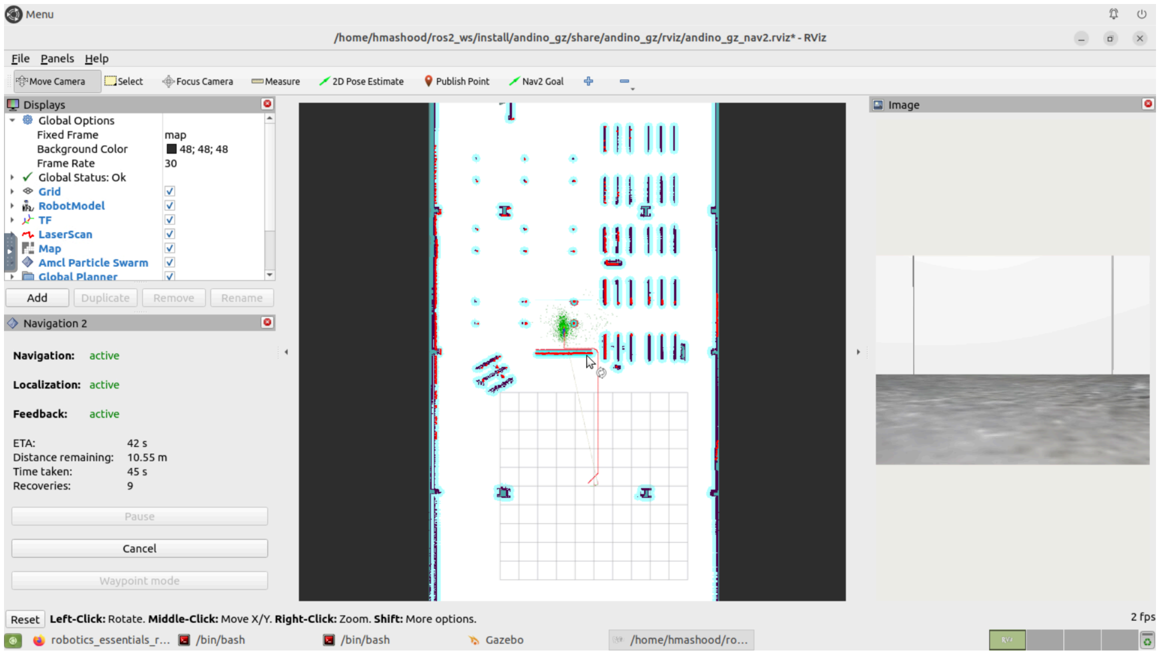
```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
Grid                 ✓
RobotModel           ✓
TF                   ✓
LaserScan            ✓
Map                  ✓
Amcl Particle Swarm  ✓
Global Planner       ✓

Navigation:    active
Localization:  active
Feedback:      active

ETA:                  70 s
Distance remaining:   17.45 m
Time taken:           37 s
Recoveries:           22
```

Figure 2: Andino Path to Start

To test the algorithm even more, the path was completely blocked by placing a third obstacle in Andino's path. The algorithm successfully detected the blocked path and created an alternative one, as seen in Figure 3.



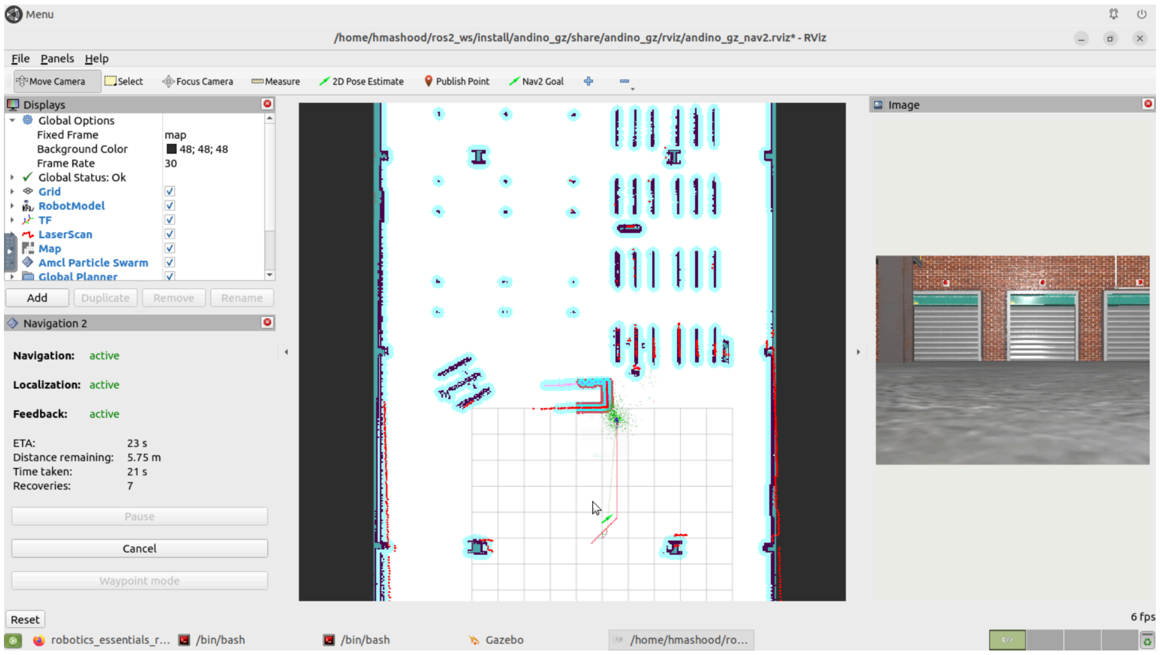
```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
Grid                 ✓
RobotModel           ✓
TF                   ✓
LaserScan            ✓
Map                  ✓
Amcl Particle Swarm  ✓
Global Planner       ✓

Navigation:    active
Localization:  active
Feedback:      active

ETA:                  42 s
Distance remaining:   10.55 m
Time taken:           45 s
Recoveries:           9
```

Figure 3: Updated Path

Figures 4 and 5 show the Andino traversing along the obstacles and reaching its final destination.



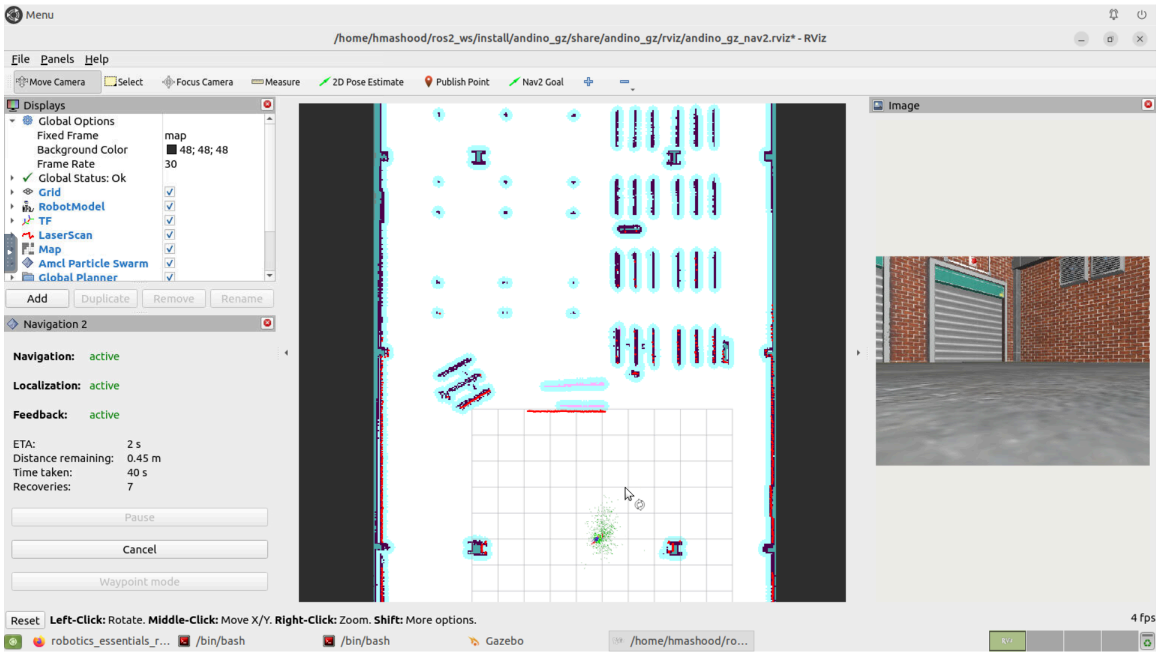
/home/hmashood/ros2_ws/install/andino_gz/share/andino_gz/rviz/andino_gz_nav2.rviz* - RViz

Global Options
Fixed Frame: map
Background Color: 48; 48; 48
Frame Rate: 30
Global Status: Ok

Navigation: active
Localization: active
Feedback: active

```
ETA:                 23 s
Distance remaining:  5.75 m
Time taken:          21 s
Recoveries:          7
```

Figure 4: Navigating Along Obstacles



/home/hmashood/ros2_ws/install/andino_gz/share/andino_gz/rviz/andino_gz_nav2.rviz* - RViz

Global Options
Fixed Frame: map
Background Color: 48; 48; 48
Frame Rate: 30
Global Status: Ok

Navigation: active
Localization: active
Feedback: active

```
ETA:                 2 s
Distance remaining:  0.45 m
Time taken:          40 s
Recoveries:          7
```

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Zoom. **Shift:** More options.

Figure 5: Andino's Nav2 Goal

Conclusions, Resources, & Acknowledgments

Self Learning

For us, ROS 2 was entirely new when we first heard about it in the previous Robotics course. Through this project (and the exercises), we gained invaluable hands-on experience that demonstrated the robust capabilities of ROS 2.

Not only did it reinforce the concepts introduced during the exercises, but it also pushed us to expand our understanding and apply more advanced techniques in robot navigation. Overall, the project was a significant step in building our confidence and skills related to ROS 2.

References

[1] Henki-Robotics. (n.d.). *robotics_essentials_ros2/5-path_planning at main · henki-robotics/robotics_essentials_ros2*. GitHub. https://github.com/henki-robotics/robotics_essentials_ros2/tree/main/5-path_planning

[2] *ROS2: Path Planning Algorithms Python Implementation*. (n.d.). https://robotics.snowcron.com/robotics_ros2/multi_bot_nav_03_intro.htm

[3] Cai, Q. (2024). A comparison between A* and RRT algorithm in path planning for mobile robot. *Highlights in Science Engineering and Technology*, *97*, 282–287. https://doi.org/10.54097/2stv5y97

Use of Generative AI

Some parts of this project's documentation and code were done with the help of ChatGPT. Specifically, it was used to clarify algorithm implementation details, suggest code structuring, and refine documentation language.

Code

The **exercise_ws** can be found in the following GitHub Repository, whereas the **path_planner_node.py** can be found here.