4TM00 Robot motion planning and control – Assignment 2 Report group 11

J. Frydrych, D. Akulovas w

***Abstract*— This work addresses the implementation of minimum-cost navigation over given occupancy grid map for the RoboCyl mobile robot using ROS2 within the Gazebo simulation environment. The assignment is divided into three tasks: (1) Safe Navigation Costmap where a costmap is being generated based on occupancy map in a way that results in higher cost being associated with grids close to obstacles, and lower cost being assigned to unoccupied space; (2) Search-Based Path Planning in which using the calculated costmap a safe navigation path is calculated to given goal position; (3) Sensor-Based Path Following which consists of safely following calculated reference path while avoiding obstacles. The designs aim for modularity, reusability, and adherence to safety constraints, ensuring reliable performance even in complex scenarios. Results demonstrate the ability to safely navigate around the given occupancy grid map highlighting both pros and cons of implemented solutions.**

I. Introduction

This assignment focuses on implementing minimum-cost navigation for the 'Robocyl' mobile robot using ROS2 in the Gazebo simulation environment. The work is divided into three main parts: generating a safe navigation cost map, calculating a reference path, and safely following that path while avoiding obstacles. The cost map is derived from an occupancy grid map, where higher costs are assigned to areas near obstacles and lower costs to free space. Using this cost map, a search-based algorithm calculates the optimal path to a specified goal position. Finally, the robot follows the reference path as accurately as possible. The implementation prioritizes modularity and reliability, while adhering to safety constraints.

II. Safe Navigation Cost map

In the first part of the assignment, we implemented a ROS2 node that receives an occupancy grid map and generates a safe navigation cost map for path planning. Our solution uses exponential decay from obstacles to assign costs to all nodes. Any node that is near an obstacle will have a higher cost than ones that are far away. Initially, obstacles are stored in a binary matrix where "1" values correspond to occupied cells, and unoccupied ones are represented as "0" values.

The occupancy grid contains values representing cell states: 0 – free state, 100 – fully occupied (obstacle). We convert the grid into a binary occupancy matrix where each cell is classified as either obstacle or free. The occupancy threshold is a user-defined parameter to determine the obstacle cutoff.

We compute the Euclidean distance of each free cell to the nearest obstacle using the distance transform. Obstacles are treated as 0 (source points), and free space is treated as 1. The result is the distance matrix D:

$$D[i,j] = \text{distance\_transform\_edt}(1 - \text{binary\_occupancy\_matrix})$$

The function calculates the Euclidean distance for each cell (i, j) to the nearest obstacle.

The grid-based distances $D[i, j]$ are converted to meters by multiplying with the grid resolution r, which represents the size of each grid cell:

$$d[i,j] = D[i,j] \cdot r$$

where: $d[i, j]$d – metric distance for cell (i, j), r – grid resolution (size of each cell in meters).

The cost $C[i, j]$ at each cell is calculated using an exponential decay function. This ensures that cells closer to obstacles have higher costs, and costs decay as the distance increases:

$$C[i,j] = \max\left(C_{\min}, C_{\max} \cdot e^{-k \cdot d[i,j]}\right)$$

where: $C[i, j]$ – cost at cell (i, j), C_{max} – maximum cost (here: 100), C_{min} – minimum cost (here: 1), k – decay rate (controls how quickly costs decrease with distance), $d[i, j]$ – metric distance to the nearest obstacle.

The computed cost values $C[i, j]$ are clipped to ensure they remain within the valid range of the `int8` data type. The cost values are then converted to 8-bit integers for compatibility with the 'OccupancyGrid' topic:

$$C[i,j] = \text{clip}\left(C[i,j], (-127,127)\right)$$

One of the main drawbacks of this solution is that the calculation of the cost map only occurs once, which does not allow for real-time adaptability. We are also forced to fine-tune the parameters for each robot and situation, as the calculation of safe distance to obstacles is absent and we rely on decay rate to define impassable grid points. This could be solved by introducing a parameter like a safety margin, which would be calculated based on the robot's dimensions, allowing for faster setup.

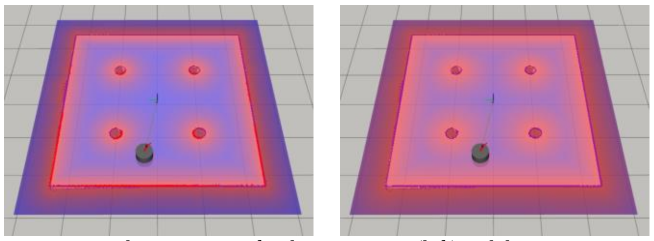
Fig 1. Resulting costmaps for decay_rate = 2 (left) and decay_rate = 1 (right)

Adjusting the decay rate would influence the final path planning. For example, choosing smaller decay_rate value would discourage robots from attempting to move in between close together obstacles encouraging moving around clumps of obstacles. The influence of different decay_rate values has been shown in figure 1.

III. Search-Based Path Planning

In the second part, we were tasked with generating a safe path across the grid map leading to the goal. This was done using the cost map along with the A* path-finding algorithm. By following the general direction towards the goal (using Euclidean distance as the heuristic), the code identifies the lowest-cost nodes near the starting position (the robot's location) and propagates towards other nodes in the direction of the goal. This eventually maps out an optimal path. A* was chosen over Dijkstra because it is faster and does not require calculating the cost of all possible paths in the cost map.

The cost function balances the real cost, and the estimated heuristic, it is defined as:

$f(n) = g(n) + h(n)$

where: $f(n)$ - total estimated cost to reach the goal, $g(n)$ - actual cost to reach node n , $h(n)$ - heuristic cost estimate to the goal.

Euclidean distance is used as the heuristic, which ensures efficient exploration by prioritizing nodes closer to the goal:

$h(u,v) = \sqrt{(r_u - r_v)^2 + (c_u - c_v)^2}$

where (r_u, c_u) are the grid indices of node u , and (r_v, c_v) are the grid indices of the goal node v.

The edge weights for horizontal and vertical neighbors are calculated by:

$$w_{u,v} = \frac{c_u + c_v}{2}$$

For diagonal neighbors:

$$w_{u,v} = \frac{c_u + c_v}{2} \cdot \sqrt{2}$$

where c_u and c_v are the costs of nodes u and v .

The coordinates are converted from the global to the grid by:

$$i = \frac{y - y_0}{\text{resolution}}, \quad j = \frac{x - x_0}{\text{resolution}}$$

A major limitation of this implementation is the slow computational speed. The complexity of the calculations makes real-time path recalculation impractical, limiting adaptability to dynamic obstacles or rapidly changing goals. The path is recalculated when the robot has moved significantly, ensuring the robot stays on an optimal path while reducing computational load. While the implementation does not yet adapt to changing goal positions, this can be easily added to the goal callback function.

To improve computational speed, random sampling methods (e.g., RRT) could be used instead, or path-planning could be combined with real-time sensor data, such as laser scans, to enhance responsiveness to changes in the environment.

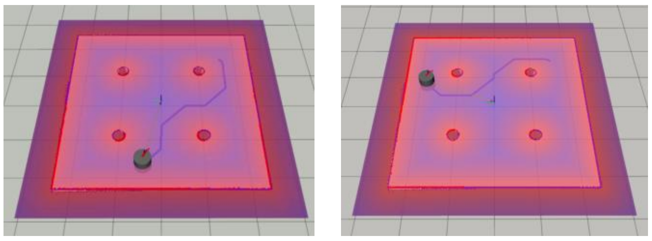
Fig 2. Resulting path generation

IV. Sensor-Based Path Following

In the final part of the assignment a simple path following approach has been implemented. Using the previously generated path the robot sequentially moves through provided waypoints until the final goal is reached. Logic for this algorithm can be represented with following pseudocode:

Path following logic:

```
1: If path has been created
2:     Choose first waypoint as current waypoint
3:     Calculate distance between robot and current waypoint
4:     If distance < threshold
5:         Chose next waypoint
6:         If no more waypoints
7:             Goal reached, stop robot
8:     Else
9:         Apply proportional controller linear velocity
```

This algorithm being part of the *timer_callback* function is being repeatedly called with set frequency. It is important to notice that in this case the path is being created only once, at the beginning of simulation. Due to significant computation time of path planning real time updates would lead to robot moving path created based on its past position, often forcing it to move backward, away from its goal.

This algorithm has two key parameters that adjust the path following execution. The first one, *waypoint_threshold*, defines distance in meters at which current waypoint would be considered “reached”, selecting next one to move to. Increasing this distance can lead to robot “cutting corners” and following path with less accuracy but with smoother trajectory. As we are not using any sensor obstacle avoidance here, this can lead to collisions with obstacles, therefore tuning this parameter carefully is needed to ensure both smooth and safe operation. The second parameter, *velocity_Kp*, is used in proportional velocity controller using following equations:

$$velocity_x = dis\tan c\, e_x \cdot velocityKp$$
$$velocity_y = dis\tan c\, e_y \cdot velocityKp$$

Choosing larger values for velocity_Kp would increase the robot's speed, but excessively high values could lead to undesired oscillations around the generated path and overall, less stable operation. The speed of the robot's movement is also constrained by the frequency of the timer_callback function, as the robot might move faster than new waypoints are being selected.

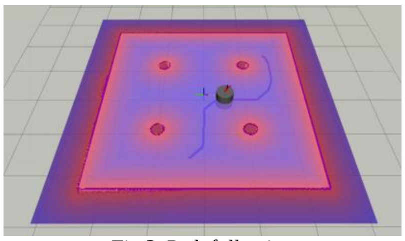
Fig 3. Path following

V. Conclusion

This assignment successfully implemented a minimum-cost navigation system for the RoboCyl robot using ROS2 and Gazebo. We generated a safe navigation cost map, computed optimal paths with the A* algorithm, and followed these paths using a simple proportional controller. The system demonstrated effective navigation and adhered to safety constraints, though limitations included static path planning and high computational cost, affecting real-time responsiveness. Future improvements could focus on dynamic obstacle avoidance, real-time path recalculation, and computational optimizations. Overall, the implementation achieved reliable and smooth navigation, with potential for further refinement.