ROBOTICS : 276A HW5 REPORT

PID1: A69034019 Name: Anandhini Rajendran

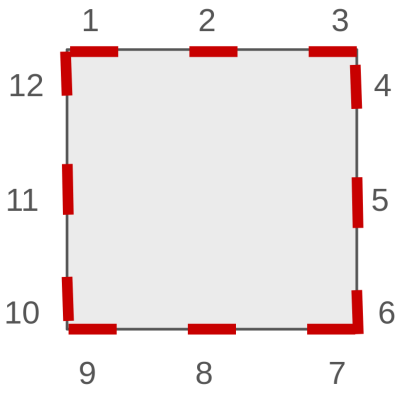
PID2: A69033245 Name: Yueqi Wu

Problem Description

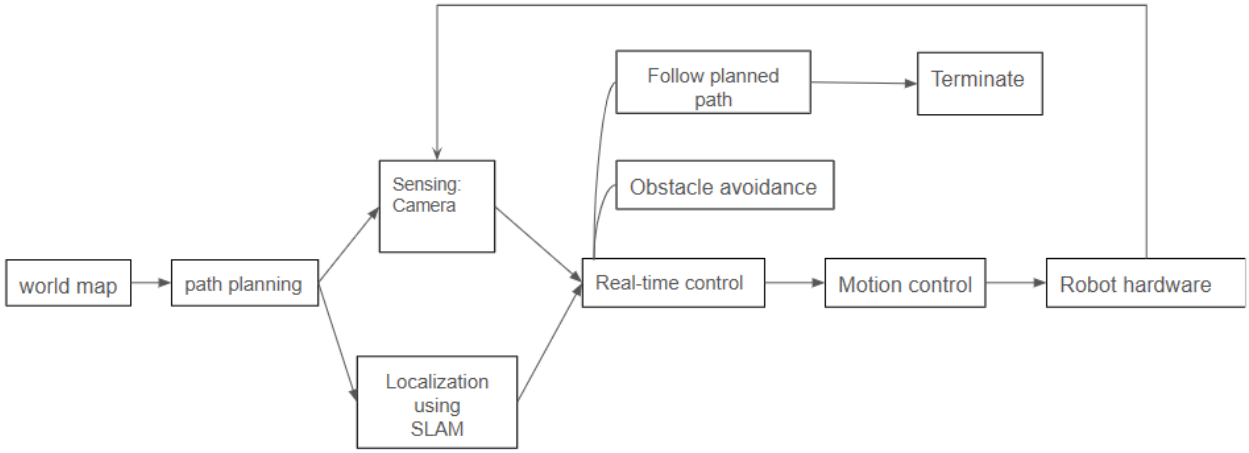
We are required to design a Roomba-like system which navigates a map and provides full coverage.

Environment

We set up a 3m * 3 m (approx) environment with landmarks on each side. The whole setting including where the landmarks are placed is shown in the following picture:



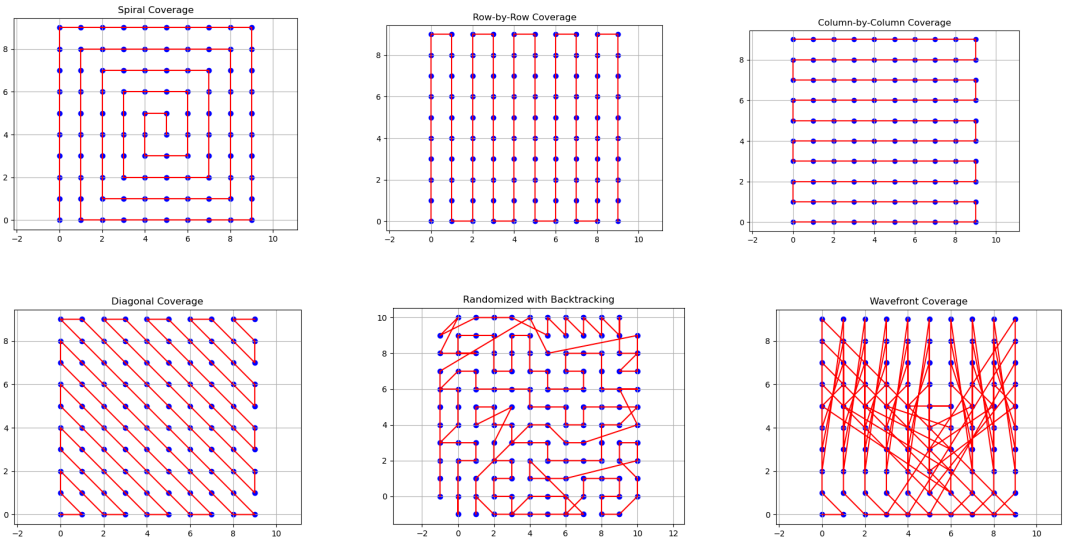
Roomba-like System



The image above shows the structure of our system. The system architecture enables a robot (e.g., a Roomba) to autonomously navigate and clean an environment. The architecture ensures modularity and scalability, allowing the system to adapt to various environments and requirements. The motivation of this system is to integrate sensing, localisation, planning, and control in a closed feedback loop from all the previous homework.

Components Overview:

1. **World Map**: Provides a pre-built or dynamically updated representation of the environment, including walls, furniture, and obstacles. This is critical for efficient navigation and planning. In the cases where we use SLAM, we don't have the map and build the map as the robot moves as done in homework 3.
2. **Path Planning**: Determines the optimal route for the robot to follow based on the map to achieve complete area coverage while avoiding obstacles. It can be done static(using a predefined map) or dynamically included when encountering obstacles. There are many ways to plan the path. While a random path is simple and easy to implement, it takes infinite time for full coverage and lacks efficiency, hence not the ideal solution for systematic and complete navigation. Based on this, we decided to use non-random algorithms for path planning, considering the following two scenarios from the perspective of whether obstacles are present in the environment:
    - For no obstacle-based environment: We can do a row-by-row, column-by-column, spiral coverage, diagonal coverage, and other traversal solutions as shown in the below graphs. These guarantee coverage for areas in the grid.
    - For an environment with obstacles: We can use algorithms like modified row-by-row as given in the behaviour for coverage and avoidance section of the report, spiral spanning tree and way front path from the robotics toolbox.



3. **Sensing (Camera)**: It detects obstacles and boundaries, captures environment details in real-time, and provides data for SLAM and control systems. In the real world, sensing can be done using a camera, LiDAR, GPS etc. For the homework, we used the robot's camera to detect April tags.

4. **Localization Using SLAM**: Simultaneously Localizes the robot and Maps the environment using camera data - in our case it's the April tag vector, ensuring precise movement even in dynamic environments. This can be done using existing libraries or code as done in homework 3. Tuning the Q, and R based on the setup for the robot is essential for good localization.
5. **Real-Time Control**: Combines the planned path with sensor data to adjust the robot's state in real time.
6. **Obstacle Avoidance**: Ensures the robot can dynamically navigate around unforeseen obstacles without deviating significantly from the planned path. This can be done static(do path planning for a predefined map) or dynamically.
7. **Motion Control**: Converts high-level navigation commands into motor commands for the robot's wheels. In our case its the set_four_motors() function in the mpi_twist_controller_node.py
8. **Robot Hardware**: Executes the commands to physically move the robot and provide information for sensing by the camera.

This system architecture ensures robust operation through continuous environmental awareness, real-time adaptability, and systematic task execution.

Representation

We considered the Grid representation since:

1. A grid divides the environment into uniform cells, making it easy to represent and visualize the world. Each cell can be marked as a free space or an obstacle. This binary representation is straightforward and easy to manage programmatically. Also easier to plot the proof for area-based coverage.
2. Many path-planning algorithms like A* and Dijkstra are designed to work well with grid-based maps. These algorithms can easily expand paths from one cell to adjacent cells based on cost or heuristics. The uniformity of grids simplifies the computation of costs, neighbours, and traversal paths.
3. Grid representation allows straightforward collision checks by marking cells within the robot's radius or near obstacles as occupied or unsafe. This ensures that the robot avoids unsafe zones by assigning high costs or prohibiting traversal through these cells.

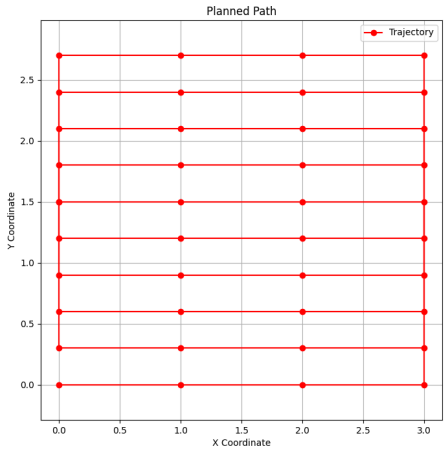
Algorithm

Our algorithm does not utilize a SLAM module. Instead, we set the process as following:

1. Predefined Map: A predefined map of the environment is provided to the robot according to the settings as shown in the Environment section of the report.
2. Path Planning: We use a sliding window-based coverage algorithm.
3. Control System: The robot's movement is controlled using a PID controller.
4. Sensor-AprilTag Correction: The system periodically corrects the robot's position once in 0.8s, using AprilTag-based sensing if it detects April tags.
5. Coverage plot: Finally we plot the path traversed and current_state of the robot to give the proof of coverage.

Initially, we implemented a row-by-row and a spiral coverage algorithm for area coverage. However, the row-by-row approach resulted in significant drift over time due to cumulative errors, which adversely affected the coverage accuracy. For the spiral coverage method , when the robot reaches the center, the distance it needs to move becomes very small, resulting in such slow movement that the wheels almost stop turning.

To address this, we propose a sliding window algorithm, where the robot alternates between covering the i-th row and the (n/2+i)-th row in each pass, instead of sequentially covering the i-th and (i+1)-th rows. This approach mitigates drift by distributing the coverage effort more evenly and allowing for better slide of the robot between widely separated rows. We define the grid size and the overlap based on the dimensions of the robot. The overlap and the choice of algorithms can be adjusted to meet specific real-world requirements. Path planned using the sliding window path:



Waypoints: [[0.0, 0.0], [1.0, 0.0], [2.0, 0.0], [3.0, 0.0], [3.0, 1.5], [2.0, 1.5], [1.0, 1.5], [0.0, 1.5], [0.0, 0.3], [1.0, 0.3], [2.0, 0.3], [3.0, 0.3], [3.0, 1.8], [2.0, 1.8], [1.0, 1.8], [0.0, 1.8], [0.0, 0.6], [1.0, 0.6], [2.0, 0.6], [3.0, 0.6], [3.0, 2.1], [2.0, 2.1], [1.0, 2.1], [0.0, 2.1], [0.0, 0.9], [1.0, 0.9], [2.0, 0.9], [3.0, 0.9], [3.0, 2.4], [2.0, 2.4], [1.0, 2.4], [0.0, 2.4], [0.0, 1.2], [1.0, 1.2], [2.0, 1.2], [3.0, 1.2], [3.0, 2.7], [2.0, 2.7], [1.0, 2.7], [0.0, 2.7], [0.0, 1.5]]

VIDEO:
https://www.youtube.com/watch?v=XjPuSuoEeIU
The video only shows a few iterations since completing the whole process will take a lot more time. Follow the same motion for multiple iterations.
The below video is another run for the same motion: https://www.youtube.com/watch?v=1n_Eeh99ZWM

Behaviours for Coverage and Avoidance:

There are two ways to solve the robot navigation problem:

1. Use a random motion: But this method takes infinite time to guarantee full coverage
2. Have a logical path planned using the map provided.

We use the second method to traverse the waypoints and cover all areas on a given map. For coverage without obstacles, we have implemented a sliding window-based approach as explained in the previous section.

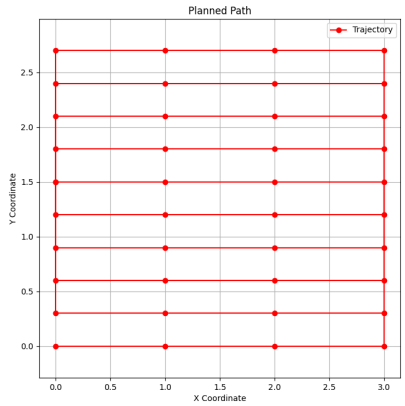
For coverage with obstacles, we can use one of the below algorithms:
1. Modified Row-by-Row Algorithm
We used the A* algorithm combining with the row-by-row algorithm to ensure coverage while avoiding obstacles.
Algorithm:

1. The input for the algorithm is a grid with obstacles defined in the grid.
2. We traverse the grid row by row. At the start of each row, we check if any obstacles are detected in the current row. If it's detected, we use path-planning algorithms like A* to navigate around the obstacle.
3. After navigating around the obstacles it resumes traversal and moves to the next row when it reaches the end of the row.
4. Repeat the row traversal until we reach the end.

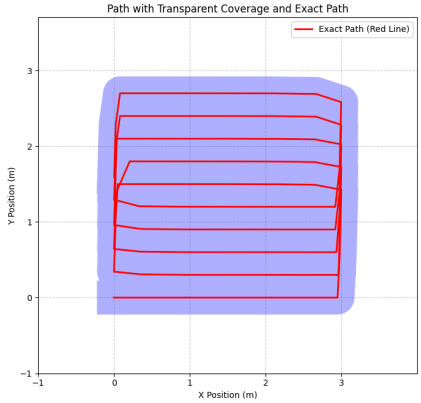
2. Other algorithms for obstacle avoidance.
The below methods are good obstacle avoidance algorithms which are robust and handle edge cases.

a. Ref: https://github.com/rodriguesrenato/coverage-path-planning
    i. The movement model is based on a grid representation where each cell corresponds to a specific position. The robot can move to adjacent cells—up, down, left, or right—with predefined movement costs for each action. The obstacles are numbered 1 in the grid and the free spaces as 0.
    ii. The system has two algorithms: The first is a coverage search algorithm that covers all free cells in the grid. The second is the closest unvisited search algorithm, which uses the A* search method to efficiently target the nearest unvisited cell.
b. Ref: https://ieeexplore.ieee.org/abstract/document/1013479 - Spiral Spanning Tree Coverage Path Planning: In this method, the robot constructs a spanning tree over the free space of the grid. Starting from an initial position, it follows the tree in a spiral manner, covering all reachable areas. The spanning tree guides the robot around obstacles, ensuring that each area is covered without unnecessary repetition or omission.

Coverage Guarantee Proof



planned path - sliding window



Actual trajectory and coverage of the robot

The second graph shows the actual trajectory covered by the robot using the current state of the robot. The path shown in red shows the actual trajectory of the robot when performing the sliding window. The transparent blue region shows the coverage area corresponding to the robot's trajectory. The thickness of the transparent path (equal to the robot's width) shows the area covered by the robot when traversing.

Sliding Window Motion:

- The robot's motion is designed such that its coverage window overlaps a little with adjacent rows. This ensures that every part of the grid is covered, with no gaps between rows. We aim for the least overlap to ensure an energy efficient solution.
- The transparent blue region in the first plot demonstrates the robot's physical coverage area, accounting for the sliding window effect. This plot proves the complete coverage of the area.
- Boundary and Edge Cells: The trajectory starts and ends at the edges of the grid.
- Mathematical Completeness: The Sliding window width guarantees that no cell is skipped in the vertical direction. The trajectory guarantees that the robot moves to every grid row systematically.
- Redundancy and Overlap: The overlapping nature of the sliding window provides redundancy, ensuring complete coverage even if the trajectory has slight inaccuracies. The overlap can be programmed according to the needs of the real world problem.

Mathematical Proof:

Alternating Row Coverage:

- At each step, the robot covers the i-th row and then the (n/2+i)-th row. As it iterates through all indices from 0 to (n/2−1), every row from 0 to n−1 is covered once.

Sliding Window Advantage:

- Drift in the i-th row does not propagate directly to the (i+1)-th row, as the robot instead switches to a distant row (n/2+i).

- This spatial separation allows for drift corrections during the return transition between distant rows.
- Periodic Correction: Repeated transitions between widely spaced rows allow for drift compensation using localization methods (e.g., AprilTag corrections).

Conclusion:

By employing a **sliding window approach**, the trajectory guarantees 100% coverage of the grid. The window overlap ensures no gaps between rows, and every cell is visited. This method is both mathematically rigorous and visually demonstrated in the plots.