Group 11 - D. Akulovas, J. Frydrych

4TM00 Robot motion planning and control – Group Project Report

***Abstract*—** This work presents the implementation of a robot navigation framework for the TurtleBot3 Waffle Pi mobile robot in the Gazebo simulator. The system's objective was to reach a finite number of goal positions in a known environment. An additional challenge was intermittent global localization, which introduced complexity to the system. The solution addressed five key aspects: (1) Odometry, used to achieve accurate pose estimation at a higher rate; (2) Safe Navigation Costmap, where a costmap was generated based on the occupancy map; (3) Path Planning, utilizing the calculated costmap to compute a safe navigation path to the given goal positions; (4) Sensor-Based Path Following, which ensured adherence to the calculated reference path while avoiding obstacles; (5) Multi-Goal Navigation, aimed at repeatedly reaching provided goals using the previously described functionalities. The integration of these components demonstrated the system's ability to safely navigate the given occupancy grid map, with both the advantages and limitations of the implementation being highlighted.

I. Introduction

The ability to navigate safely and efficiently in complex environments is a cornerstone of autonomous robotics. In this project, the focus is on enabling the TurtleBot3 Waffle Pi to navigate a large, office-like environment while dealing with the challenge of intermittent global localization. The global pose updates, provided at a low frequency (0.3 Hz), create significant navigation challenges, necessitating the use of supplementary odometry methods for accurate pose estimation at higher rates.

To achieve this, the framework must integrate several critical components:

1. Odometry for high-frequency pose estimation (e.g., 10 Hz) to bridge the gaps between global pose updates.
2. Costmap generation to account for obstacle proximity during path planning.
3. Path Planning to compute safe, efficient paths to given goals based on the costmap.
4. Path Following to generate velocity commands that safely guide the robot along the planned path.
5. Multi-Goal Navigation to integrate these modules and allow the robot to sequentially reach multiple goal positions.

The framework leverages the ROS2 ecosystem, utilizing tools and resources provided in the project package core_tue4tm00_project. This allows for modularity and reusability of implemented solutions. This implementation uses ROS2 Humble along with Gazebo Fortress Simulator.

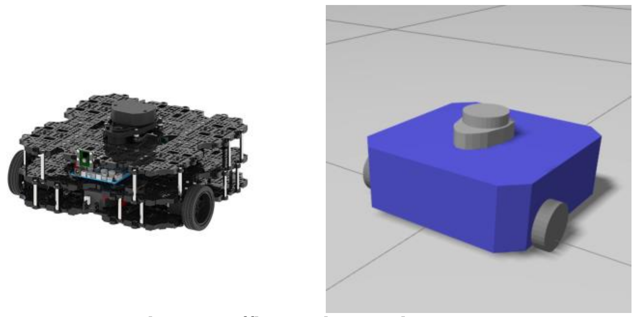
Fig. 1 TurtleBot3 Waffle Pi and its simulation counterpart

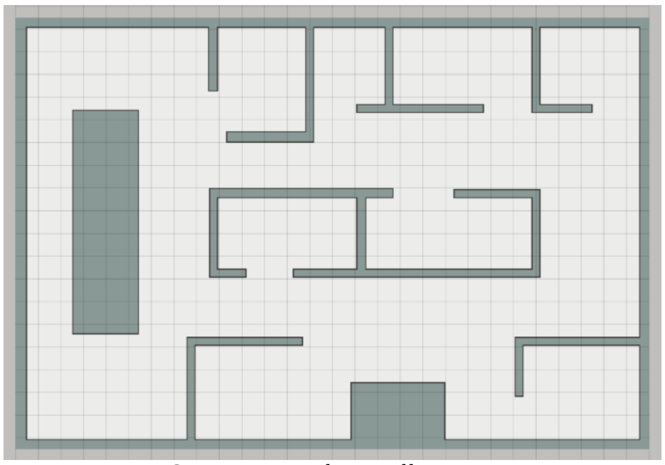
Fig. 2 Occupancy map of given office environment

Project instructions assume the use of a TurtleBot3 Waffle Pi robot platform equipped with 2D 360° Laser sensor. Unlike previous assignments, this robot isn't fully actuated and uses a differential drive mechanism instead. This will introduce an additional challenge when it comes to path following. The environment in which this robot is supposed to navigate is a large, office-like environment with multiple rectangular rooms, corridors and door-like openings.

II. Odometry

Odometry is a critical component of the robot's navigation system, providing high-frequency pose estimation to bridge the gaps between global pose updates. The TurtleBot3 Waffle Pi uses a differential drive mechanism, requiring careful integration of velocity commands to estimate the robot's position and orientation over time.

The odometry is calculated using the robot's commanded velocities v_x (forward linear velocity), v_y (linear velocity in the lateral direction, which is zero for differential drive), and ω_z (angular velocity about the vertical axis). These velocities are integrated over discrete time intervals Δt to update the robot's pose (x, y, θ), where x and y are the robot's Cartesian coordinates, and θ is its orientation.

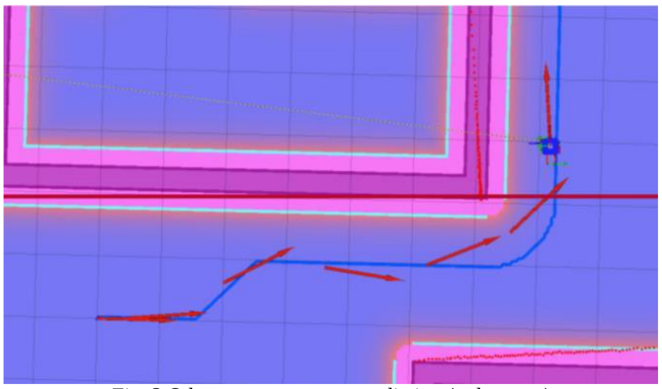
Fig. 3 Odometry movement prediction (red arrows)

The pose update equations are given by:

$$x_{t+1} = x_t + v_x \Delta t \cos\theta_t - v_y \Delta t \sin\theta_t$$
$$y_{t+1} = y_t + v_x \Delta t \sin\theta_t + v_y \Delta t \cos\theta_t$$
$$\theta_{t+1} = \theta_t + \omega_z \Delta t$$

To ensure numerical stability, the orientation θ is normalized to lie within the range $[-\pi, \pi]$ after each update:

$$\theta = (\theta + \pi)\, mod(2\pi) - \pi$$

This process is implemented in the "Odometry" node, which subscribes to velocity commands (cmd_vel) and periodically integrates them to update the robot's pose at a configurable rate (in this case, 10 Hz). The estimated pose is published as a "PoseStamped" message on the "pose_out" topic. The figure 3shows some of the odometry points visualized in RViz.

To mitigate integration errors, odometry updates are triggered only when the elapsed time Δt is within a valid range. Additionally, while the robot's velocities are assumed to be accurate, drift and errors accumulate over time due to wheel slippage, imperfect calibration, and sensor noise. These errors are corrected periodically using global localization updates provided by the "pose_in" topic at a lower frequency (e.g., 0.3 Hz).

III. Costmap

Due to the modularity and reusability of solutions from previous assignments, costmap generation remains mostly unchanged. However, improvement in the form of additional clearance parameter has been introduced. This ensures more flexibility in balancing safety and efficiency of paths planned based on generated costmaps.

Calculations begin with the received occupancy grid that contains values representing cell states: 0 – free state, (0,1) – uncertain, 1 – fully occupied (obstacle) and -1 – unknown. This grid is converted 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.

With the inclusion of clearance, the costmap calculation considers not only the distance to obstacles but also an additional safety margin. In binary occupancy matrix obstacles are treated as 0 (source points), and free space is treated as 1. The result is the distance matrix D, where distance to nearest obstacle is calculated for each free grid cell (i,j):

D[i,j] = distance_tranform(1-binary_occupancy_matrix)

Afterwards the metric distance is converted to meters by multiplying by grid resolution r *and* adjusted by subtracting the clearance parameter:

$$D[i,j] = \max(D[i,j] \cdot r - c,\ 0)$$

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. It is also necessary to fine-tune the parameters for each robot and situation, as the calculation of safe distance to obstacles is absent, and the decay rate is relied upon to define impassable grid points. This issue could be addressed by introducing a parameter like a safety margin, calculated based on the robot's dimensions, allowing for faster setup.

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 clearance parameter serves as a way of an additional safety margin and could be based on robot's size. Fine tuning these parameters has a large impact on resulting paths planned over calculated costmap.

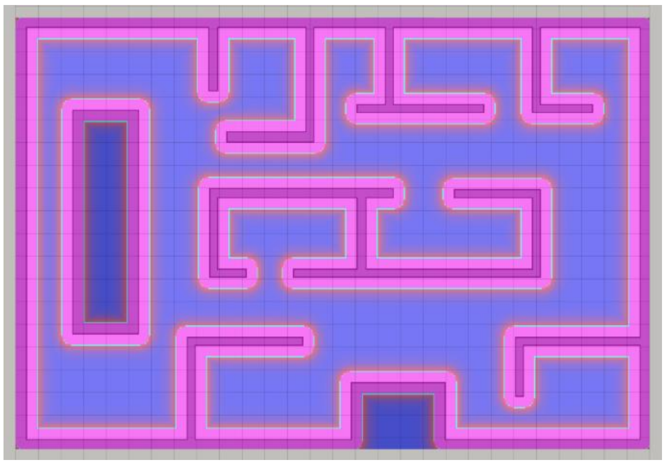
Fig. 4 Generated Costmap, decay_rate=1.0, clearance=0.2

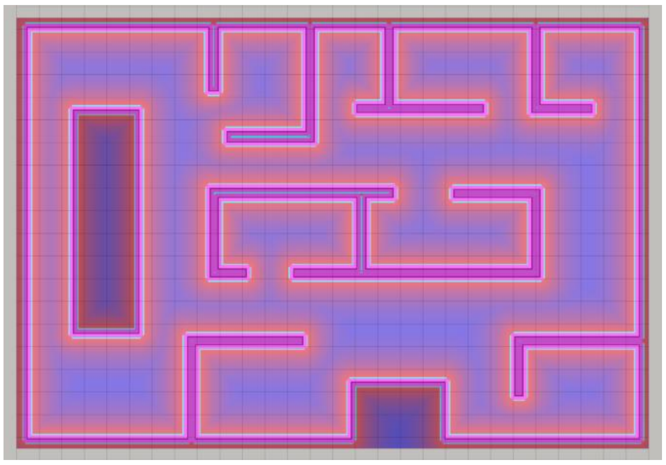
Fig. 5 Generated Costmap, decay_rate=4.0, clearance=0.5

The influence of different decay_rate and clearance values has been shown in figures 4 and 5.

IV. Path Planning

This part of the project focuses on generating a safe path through the environment to the specified goal position. Implemented solution is based on previously calculated costmap and 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, particularly during path recalculation. 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.

A great example of costmaps influence on generated paths can be shown between goal 1 and goal 2. Depending on cost decay rate and clearance parameters in costmap calculations, two different paths were identified. The first one, called “Safe” leads through the lower part of office environtment, where corridors are wider. Additionally, the generated path leads almost exactly in between walls ensuring maximal distance from all obstacles, and therefore higher safety.

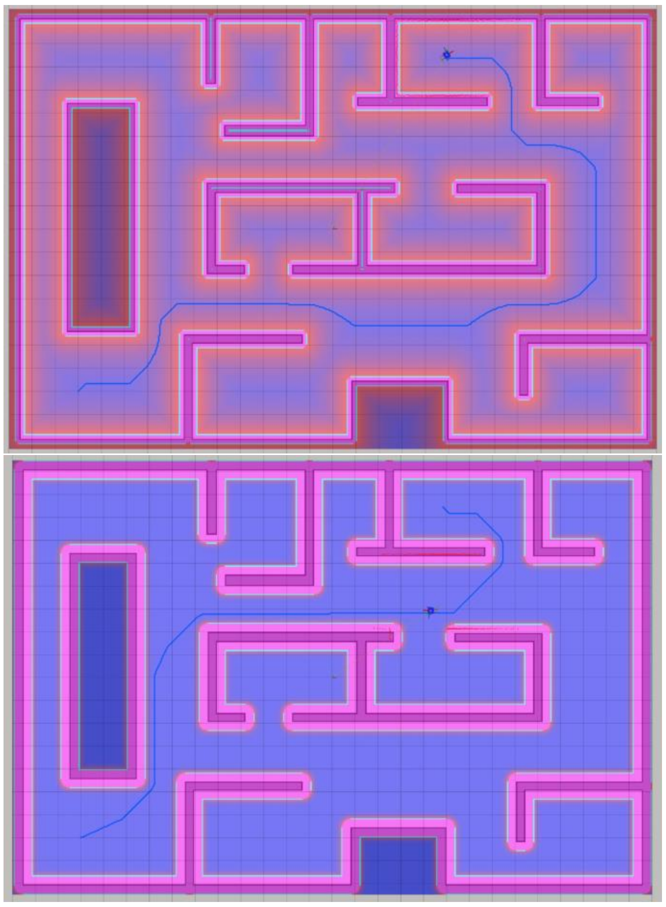
Fig. 6 "Safe" and "Efficient" paths between goal 1 and goal2

The other generated path, called "Efficient" leads through the top part of the environment where corridors are narrower. Costmap distribution also allows path planning algorithm to stick much closer to the walls resulting in smaller distance to cover. This comes at risk of hitting the walls, especially if corner cutting occurs.

To combat that, clearance parameter have been increased resulting in collision free, but less safe path being generated.

V. Path Following

The path-following component is responsible for safely navigating the robot along a planned path while avoiding obstacles and ensuring adherence to the desired trajectory. This node uses real-time odometry, costmap data, and pre-computed paths to generate velocity commands for the TurtleBot3 Waffle Pi.

Implemented path following uses a look-ahead, pure-pursuit approach where the robot selects a waypoint dynamically based on its current position and a configurable look-ahead distance. This ensures smooth navigation while maintaining responsiveness to environmental changes.

The high-level process can be summarized as: determine the robot's progress toward the final goal, identify the next waypoint on the path, maintaining the look-ahead distance, use the local costmap to ensure the look-ahead point and its surrounding area are free of obstacles, generate linear and angular velocities to align the robot with the look-ahead point.

The look-ahead point is selected from the current path to ensure the robot navigates smoothly without abrupt turns. The algorithm calculates the Euclidean distance to each path waypoint, starting from the robot's current position. The first waypoint that lies beyond the configured look-ahead distance $(d_{lookahead})$ is selected.

The distance to each waypoint is calculated as:

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2}$$

where: x_i, y_i - coordinates of the i^{th} waypoint,
x_r, y_r - current robot position.

If the robot is close to the goal (distance d_{goal} < goal tolerance), the final waypoint is selected as the look-ahead point.

To ensure safe navigation, the robot evaluates the local costmap around the look-ahead point. The surrounding area is checked within a corridor defined by the configurable corridor radius $r_{corridor}$. A grid cell in the costmap is considered blocked if its cost exceeds a defined threshold C_{block}.

Grid indices are computed from world coordinates using:

$$row = \frac{y - y_{origin}}{resolution}$$

$$col = \frac{x - x_{origin}}{resolution}$$

The robot stops if any cell within the corridor radius exceeds the block threshold.

The robot's velocity is computed to align its heading toward the look-ahead point. The desired heading θ_d is:

$$\theta_d = arctan2(y_t - y_r, x_t - x_r)$$

where x_t, y_t are the coordinates of the look-ahead point. The yaw error $\Delta\theta$ is:

$$\Delta\theta = \theta_d - \theta_r$$

where θ_r is the robot's current orientation. The angular velocity ω_z is computed as:

$$\omega_z = K_p \cdot \Delta\theta$$

and clipped to a maximum value:

$$\omega_z = \max(-\omega_{max}, \min(\omega_z, \omega_{max}))$$

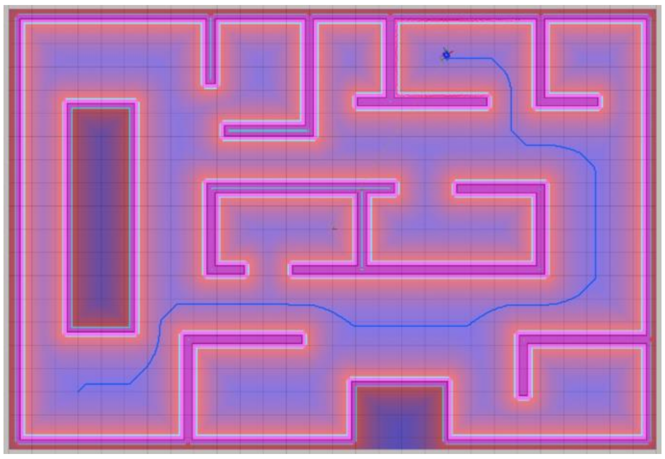
Fig. 7 Calculated path between starting position and goal 1

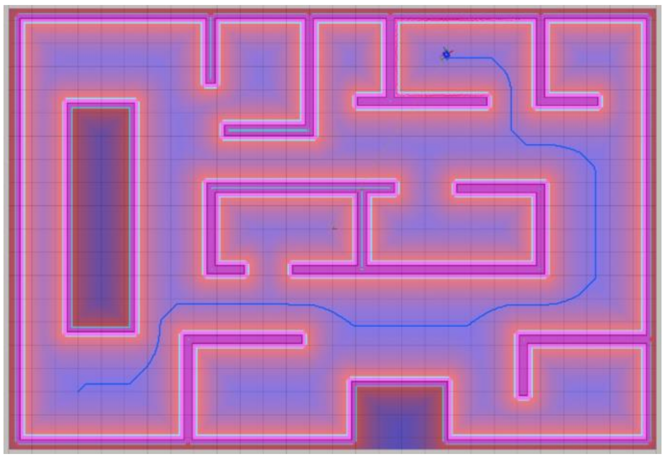
Fig. 8 Calculated path between goal 1 and goal 2

The linear velocity v_x is adjusted based on the magnitude of the yaw error:

$$v_x = v_{linear} \cdot \max(0, 1 - |\Delta\theta|)$$

These velocity commands are published as a 'Twist' message to the '/cmd_vel' topic, ensuring the robot adjusts its motion dynamically while staying within safe operating limits.

The path-following algorithm includes some safety mechanisms:

1. Collision Avoidance: The robot stops if the costmap indicates an obstacle in the look-ahead corridor.
2. Goal Tolerance: The robot stops once it is within a pre-defined tolerance of the goal position.
3. Empty Path Handling: The robot halts if no valid path is received or if the path is cleared during navigation.

This approach has some limitations. The robot relies on a static costmap for collision checking, which limits its responsiveness to dynamic obstacles. This could be addressed by incorporating real-time sensor data (e.g., laser scans) to dynamically update the costmap.

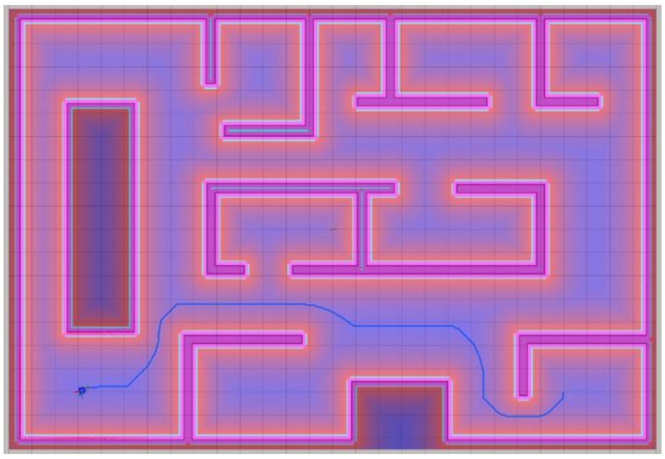
Fig. 9 Calculated path between goal 2 and goal 3

Furthermore, the behavior of the algorithm depends on tuning parameters such as the look-ahead distance, corridor radius, and angular speed gain. Improper tuning can result in suboptimal navigation, oscillatory behavior or collisions.

Finally, If the path becomes blocked, the robot halts but does not attempt to re-plan. Integrating a recovery behavior or re-planning module could improve robustness.

VI. Multi-Goal Navigation

The multi-goal navigation component enables the TurtleBot3 Waffle Pi to sequentially reach multiple goal positions in the environment. This functionality is achieved by dynamically integrating the previously implemented nodes – odometry, costmap generation, path planning, and path following.

When a new goal position is provided, the system generates a new path based on the current occupancy map and costmap. The path-following algorithm then takes over, ensuring safe navigation along this newly planned trajectory. Upon reaching a goal, the system verifies if additional goals remain in the sequence and, if so, repeats the process for the next goal.

The general workflow goes like this: a new goal is received as a 'PoseStamped' message, the path planning module computes an optimal path to the goal using the latest costmap, the robot executes the path-following logic to navigate toward the goal, while dynamically avoiding obstacles, once the robot reaches the goal (based on a configurable tolerance), it halts and prepares for the next goal.

Currently, the robot pauses between goals and does not consider dynamic re-planning during navigation. This could be addressed in future iterations by incorporating a continuous goal-handling mechanism or integrating a recovery behavior to handle blocked paths.

VII. CONCLUSION

The implemented navigation framework for the TurtleBot3 Waffle Pi successfully addressed the challenge of autonomous navigation in a known office-like environment with intermittent global localization. The system integrated key components such as high-frequency odometry for pose estimation, costmap generation for obstacle avoidance, A* path planning for optimal route calculation, and pure-pursuit path following for trajectory tracking. Multi-goal navigation was also achieved, enabling the robot to sequentially reach multiple target positions.

The strengths of the solution lie in its modularity and the effective integration of its components. The use of odometry to bridge gaps between global localization updates ensured accurate pose estimation, while the costmap generation, enhanced with a clearance parameter, provided a flexible mechanism to balance safety and efficiency. The A* algorithm produced optimal paths, and the pure-pursuit approach enabled smooth and acceptably precise path following. These features collectively allowed the robot to navigate complex environment and reach multiple goals reliably. The system's modularity also facilitates future improvements, as individual components can be refined or replaced without overhauling the entire framework.

However, the implementation faced notable limitations. The static nature of the costmap restricted the system's ability to adapt to dynamic obstacles, as it was generated only once and not updated in real time. The computational cost of the A* algorithm made frequent path re-planning impractical, limiting the system's responsiveness to changes in the environment. Additionally, the reliance on intermittent global localization introduced potential inaccuracies over time, particularly in scenarios involving wheel slippage or sensor noise. These constraints highlighted the need for more dynamic and computationally efficient solutions to improve adaptability and robustness.

To address these limitations, several improvements are proposed. Real-time updates to the costmap using sensor data, such as laser scans, would enhance the system's ability to detect and avoid dynamic obstacles. Alternative path-planning algorithms, such as RRT or DWA, could improve computational efficiency and enable real-time re-planning. Implementing recovery behaviors, such as backtracking or exploring alternative routes, would increase robustness when paths are blocked. These enhancements would significantly improve the system's adaptability, efficiency, and reliability, making it more suitable for real-world applications.