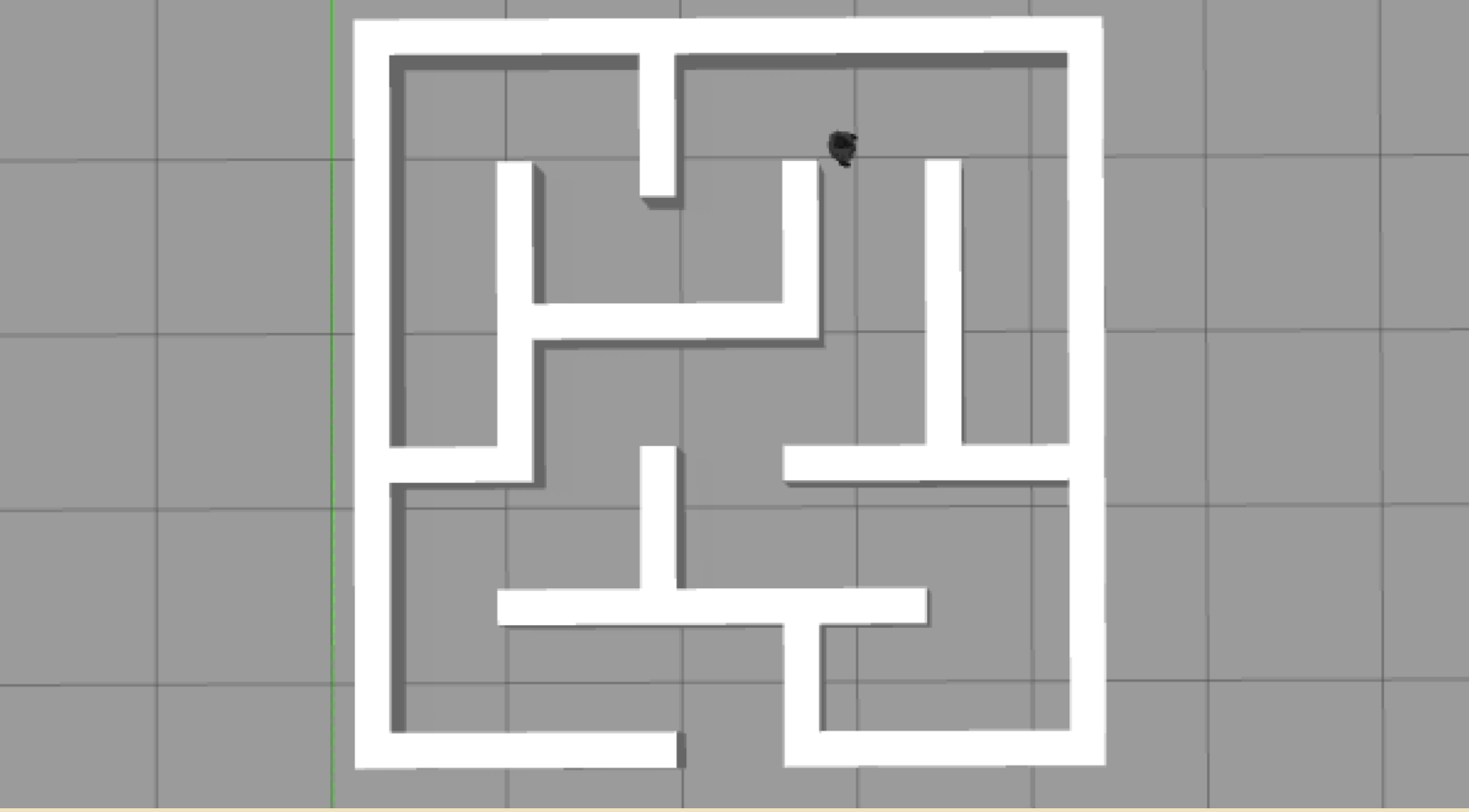


Networked Embedded Systems Group
Chair of Pervasive Computing
Department of Core Informatics
Faculty of Computer Science
University of Duisburg-Essen

March 4, 2025

project group report

Maze Runner Robotics

Soufian Khennousse
Danny Nasra
Sirine Chakchouk

Head of Chair: Prof. Dr. Pedro José Marrón
Supervisor: Carlos Medina
Alexander Golkowski
Simon Janzon
Time frame: Winter term 2024/2025

NES
Networked Embedded Systems

UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Contents

List of Figures

List of Tables

Abstract

Autonomous navigation in both known and unknown environments is a fundamental challenge in robotics due to its applications in industrial automation, autonomous vehicles, and search-and-rescue operations. This report presents the development and evaluation of an autonomous maze-solving robot using the TurtleBot3 Burger platform, ROS1, and the Gazebo simulation environment.

The project is structured into three challenges: (1) the robot has full prior knowledge of the maze layout, (2) the robot knows only the start and goal coordinates but not the maze structure, and (3) the robot has no prior information except the starting position and must explore the environment to locate the goal. For the first challenge, we implemented the A* algorithm to compute the shortest path in a fully known maze, enabling the robot to navigate efficiently. In the second and third challenges, we employed a real-time exploration strategy leveraging LiDAR sensor data and decision-tree-based path planning to autonomously discover and navigate the maze. The experiments were conducted in different simulated maze environments, and execution times were recorded to evaluate performance.

Results demonstrate that the robot successfully completed all challenges within the defined time constraints. The A* algorithm provided optimal pathfinding when the maze was known, while the exploration-based approach proved effective in unknown environments. This study enhances the understanding of maze-solving strategies and provides a foundation for more advanced autonomous navigation systems.

Chapter 1

Introduction and Background

1.1 Introduction

In recent years, the employment of robots has significantly increased in both industrial and social applications [2]. They have become one of the most widely utilized technologies across various domains [3].
In fact, robots come in a range of sizes, from large industrial machines to compact household assistants. Some of these machines are designed with autonomous capabilities, allowing them to operate independently and, in some cases, even mimic human behavior. These are referred to as autonomous robots [3].
Such robots can perform tasks in real world environments with minimal to no human intervention. Their popularity stems from their ability to execute tasks with high precision, accuracy, and consistency [4]. As a result, the integration of autonomous systems into daily life is becoming increasingly prevalent. These systems have diverse applications, ranging from household tasks, such as vacuuming and cleaning robots, to more specialized and industrial roles, including food-serving robots [5], logistics operations [6], and self-driving cars [7].
One of the fundamental aspects of autonomous mobile robotics is navigation. This feature enables a robot to move independently from one location to another, either by following a predefined path or by autonomously exploring an area and also allows it to avoid obstacles [8].
A fascinating field of robotics navigation is autonomous maze solving, in which an autonomous robot attempts to navigate a maze as quickly and effectively as feasible and reach a target location. Maze-solving robots are among the most popular autonomous systems [4].
Successful robot navigation in unknown environments has significant implications for industrial automation, autonomous vehicles, and search-and-rescue missions. Robotic path planning in warehouses, autonomous exploration of disaster-affected areas, and robotic aid in both organized and unstructured environments are just a few of the challenging real-world problems that can be resolved with the help of maze-solving techniques [4].

Given the importance of autonomous navigation and the challenges associated with robotic path planning, we chose to work on this project to gain a deeper understanding of maze solving algorithms and robot navigation strategies. Certainly this project provides insights into real-world path planning problems and serves as a foundation for more advanced navigation systems in robotics research.
We use the TurtleBot3 Burger platform, ROS1 (Robot Operating System), and the

Gazebo simulation environment to program a robot capable of navigating different types of labyrinths and escaping from them using data gathered from the robot's sensors.
The project is divided into three challenges. In Challenge 1, the robot has prior knowledge of the maze layout, including the start and goal coordinates. In Challenge 2, the robot knows only the start and goal coordinates, but has no prior knowledge of the maze structure. In Challenge 3, the robot knows only the start coordinate and must explore the environment to find the goal.

1.2 Related work

Design, implementation, and evaluation of autonomous maze solving robots have been widely investigated over the past few years.
Gatti et al.[9] introduced a probability-driven exploration strategy for unknown maze environments using an occupancy grid map, a wavefront algorithm, and gradient descent for path optimization. While their approach effectively navigates entirely unknown spaces, our project takes a broader perspective by considering multiple scenarios with different levels of prior knowledge. This allows us to evaluate maze-solving strategies in a more comprehensive manner. We both share a common foundation, using TurtleBot3, ROS, and Gazebo for simulation, ensuring a comparable test environment for autonomous navigation research.
Romadhon et al.[10] implement the A star algorithm to optimize pathfinding in a structured maze,to demonstrate its ability to find the shortest route efficiently. Similarly, in Challenge 1 of our project, we use A* for path planning when the maze layout is fully known. While their work focuses on an infrared sensor-based robot navigating a structured environment, our implementation integrates A* within a LiDAR-based navigation system in ROS and Gazebo.
Rule-based strategies like the Wall-Follower and Pledge algorithms have also been explored for maze solving. Alamri et al.[4] enhance the Wall-Follower method to avoid infinite loops.
Narendran et al.[11] evaluate the performance of A* against BFS, DFS, and Markov Decision Processes (MDP) in a structured grid-based environment. Their study highlights the efficiency of heuristic-based planning but assumes a perfect knowledge of the environment with discrete state transitions. Our work extends this by integrating A* within a physics-based simulation using ROS and Gazebo, where real-world sensor constraints and motion dynamics influence navigation accuracy.

1.3 Foundation

As mentioned in the introduction, the robot used in this project is the TurtleBot3 Burger (Figure 1.1), a widely adopted mobile platform known for its compact design, mobility and computational efficiency. It features a maximum translational speed of 0.22 m/s, a rotational speed of 2.84 rad/s, and a 15 kg payload capacity, despite weighing only

1 kg. The robot is powered by a Raspberry Pi 3 Model B/B+ as its main processor and an ARM Cortex-M7 MCU for motor control. Additionally, it is equipped with a 360-degree Laser Distance Sensor (LDS-01), enabling Simultaneous Localization and Mapping (SLAM) for navigation and obstacle detection [1, 12].

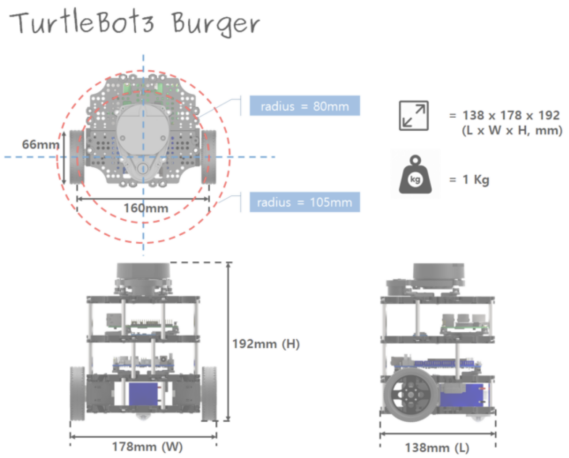


Figure 1.1: Turtlebot3 Burger[1]

To program and control the TurtleBot3, we use the Robot Operating System (ROS1), an open-source robotics framework that provides tools for hardware abstraction, localization, navigation and visualization [13, 14]. ROS follows a publisher-subscriber communication model, where nodes exchange data via topics, facilitating modular and scalable robotic system development.

For visualization and debugging, we use RViz, a key ROS tool that offers 3D representations of sensor data and robot states [15]. It supports real-time visualization of laser scans, transforms, robot models and camera feeds, allowing us to analyze planned trajectories and improve navigation strategies.

The Gazebo simulator plays a crucial role in our project by providing a realistic physics-based environment for testing the TurtleBot3 [16]. It enables the simulation of complex navigation scenarios without the risks and costs associated with real-world experimentation. Its seamless integration with ROS ensures an efficient workflow for designing, testing and refining autonomous behaviors before deployment.

Together, these technologies form the foundation of our project, enabling us to develop and evaluate maze-solving strategies in a controlled, physics-based simulation environment.

Chapter 2

Challenge 1:

In Challenge 1, the robot has complete knowledge of the maze, the start point coordinates and also the goal coordinates. In this chapter, we will explain and discuss the design implementation choices made for this challenge, including the specific approaches and algorithms used. We will then test and evaluate the robot's performance in navigating the known maze and analyze the results. Finally, we will highlight some of the challenges we faced and the limitations of our solution.

2.1 implementation:

For the implementation, we chose to split the implementation into three ROS nodes: **odom_tracker** (map_loader.py), **a_star_planner** (a_star_algorithm.py) and **controller** (tb3_driver.py). These nodes work together to enable the robot to navigate a known maze, compute an optimal path using the A* algorithm and execute motion control to follow the planned path efficiently.

2.1.1 The odom_tracker node:

The **odom_tracker** node is essential to ensure a realistic and efficient maze representation for the robot. It processes raw map data, making it compatible with the navigation system and inflates obstacles to enhance safety and resilience.

2.1.1.1 Map Data:

The Map Loader node is responsible for retrieving and processing the occupancy grid map, which represents the environment in which the robot navigates. It subscribes to the **"/map topic"**, which publishes an **"OccupancyGrid"** message containing:

- Resolution: the size of each grid cell in meters (e.g. 0.05m/cell).
- Width and Height: The number of cells in the grid, defining the map's dimensions.
- Origin: The world coordinates of the bottom-left corner of the map.
- Grid Data: The occupancy values of each cell, stored as a 1D array where:
  - 100 = occupied (wall/obstacle).
  - 0 = free space (navigable area).

– -1 = unknown (unexplored regions).

Since the **"OccupancyGrid"** message stores map data as a 1D list, but the map represents a 2D environment, this node converts it into a 2D array(map_2d) using this expression:

$$\text{map\_2d}[y][x] = \text{map\_data}[y \times \text{width} + x]$$

This conversion is crucial for two key tasks: path planning and wall expansion. For path planning, the 2D representation enables efficient computation of shortest paths. For wall expansion, it simplifies the identification and extension of obstacles for safer navigation.

2.1.1.2 Robot's position:

Tracking the robot's position is essential for successful navigation in the maze. That's why the **odom_tracker** node subscribes to the **"/odom"** topic, which provides real-time odometry messages containing the robot's position and orientation in world coordinates (position = msg.pose.pose.position). In this project, the robot's motion occurs within a defined map, these world coordinates (in meters) must be converted into pixel coordinates within the occupancy grid. This transformation ensures that the robot's position is aligned with the discrete map representation used for navigation. The occupancy grid represents the environment as a discrete grid of cells, whereas odometry data provides continuous world coordinates (in meters). To achieve this, we applied the following conversion:

$$\text{pixel}_x = \frac{(\text{world}_x - \text{map\_origin}_x)}{\text{resolution}}$$

$$\text{pixel}_y = \frac{(\text{world}_y - \text{map\_origin}_y)}{\text{resolution}}$$

pixel_x and pixel_y represent the robot's position in the occupancy grid. world_x and world_y represent the robot's position from odometry and map_origin_x, map_origin_y represents the map's bottom left corner in world coordinates. It defines the transformation between the pixel coordinates in the map and the real-world coordinates.
Once the robot's position is calculated in pixel coordinates, it is published on the **"/robot_path"** topic as a **"Point"** message. This allows the other two nodes (**a_star_planner** and **controller**) to access real time updates of the robot's location within the map.

To ensure accurate position tracking, the Odometry callback only processes data if the map has already been received. If the map is not available yet, the node waits before attempting to convert the coordinates of the robot to avoid errors. In addition to that, to prevent redundant updates or unnecessary processing, we included a flag (odom_processed) that ensures that the odometry callback does not execute multiple times if not needed. This mechanism optimizes computation by avoiding repeated calculations when they are unnecessary.

2.1.1.3 Start and goal positions:

The start and goal positions are key to the robot's navigation strategy. The start position is manually set in world coordinates, while the goal position is received from the **"maze_goal"** topic.
To align with the occupancy grid, both positions are converted from world to pixel coordinates using the same transformation as the robot's real-time position. The resulting pixel coordinates are then published for use by other nodes.

2.1.1.4 Wall extending:

To improve the safety of the robot's navigation and avoid collisions, we decided to increase the maze's walls. This is achieved by expanding the walls and adding extra obstacle layers around them. By restricting path planning near walls, this approach reduces the risk of the robot getting stuck and also decreases the computation time of the A* algorithm. In detail, the wall expansion is performed by scanning through the entire 2D occupancy grid and identifying cells marked as 100 (obstacles). Once an obstacle is detected, the algorithm modifies its neighboring cells within a specified radius (wall_extension), marking them as occupied (100). It also ensure that the extended cells remain within the valid map range and that only cells marked as 0 (free space) are converted into walls. The function **add_100_to_the_next_points()** is responsible for this task.

Since expanding walls could accidentally block the robot's starting position or goal, we used the function **check_if_start_or_end_100()** to prevent that these two important points will be changed to obstacle (100). This ensures that the robot can still start and reach its goal without being blocked by expanded walls.
Once the walls are expanded, the modified map is converted back into a 1D occupancy grid and published. So this updated map ensures that path-planning and navigation nodes receive the modified grid with expanded walls.

2.1.2 The a_star_planner node:

Finding the best route from the robot's starting position to the desired location on the map is the responsibility of the **a_star_planner** node. It uses the A* algorithm, which effectively avoids obstacles while choosing the shortest path. This node subscribes to the processed map, start and goal positions from the node **odom_tracker** and publishes the computed path for execution by the robot's movement system.

2.1.2.1 Overview of the A* Algorithm:

A* is a graph traversal and path search algorithm initially proposed by Hart et al.[17]. By leveraging heuristics, A* significantly enhances real-time performance, making it

faster and more efficient in pathfinding scenarios [18]. As a weighted graph-based search method, A* finds the most efficient path from a starting node to a goal by minimizing a cost function, such as distance or travel time. It does this by systematically exploring possible routes, prioritizing those that appear most promising. At each step, the algorithm evaluates both the actual cost of reaching a node and an estimate of the remaining cost to the goal. By continuously refining its search and expanding the most promising paths first, A* ensures an optimal and efficient solution, making it a widely used approach in navigation and robotics [18].

Before selecting the A* algorithm for path planning in our maze-solving robot, we conducted research to identify the most effective algorithms for structured maze navigation. Several studies, including those by Brasil et al.[19] and Romadhon et al.[10], provide empirical evidence of A*'s efficiency in such environments. Unlike Dijkstra's algorithm, which explores all nodes uniformly, or greedy best-first search, which can lead to suboptimal paths, A* balances exploration and optimality by considering both the actual path cost and an estimated distance to the goal.
Comparative studies further demonstrate that A* consistently delivers optimal paths while maintaining lower memory consumption and faster execution times than alternative approaches such as BFS and DFS, which either require excessive node expansion or risk getting stuck in inefficient paths. While Markov Decision Processes (MDP) offer sophisticated decision-making for uncertain environments, they introduce unnecessary computational complexity for our static and known maze structure[11].
Additionally, research by Romadhon et al.[10] highlights how hardware constraints, such as wheel slippage and battery instability, can impact real-world robot performance. However, since our implementation is in simulation using TurtleBot3 and Gazebo, we avoid these limitations, ensuring A* operates under ideal conditions. Considering these factors, A* provides the best trade-off between computational efficiency, optimal pathfinding and practical feasibility, making it the most suitable algorithm for our maze-solving robot.

2.1.2.2 Implementation of the A* Algorithm:

The A* algorithm is implemented in the **AStar** class (a_star_algorithm.py), which operates using the **Node** class. Each node represents a position in the grid and stores the following properties:

- position: The (x, y, z) coordinates of the node in the grid.
- parent: The previous node in the path, used for path reconstruction.
- g_cost: The cost from the start node to the current node.
- h_cost: The estimated cost from the current node to the goal.
- f_cost: The total estimated cost (g_cost + h_cost).

Each node generates neighbors in 8 possible directions (up, down, left, right, diagonals) ensuring that the neighbor is inside the grid boundaries, it is not an obstacle (100) and also is not already in the closed list that contains nodes that have already been processed and should not be reconsidered.

2.1.2.3 Heuristic Function:

The A* estimates the remaining cost from a given node to the goal node using a heuristic function. This minimizes computational effort while directing the search in the direction of the most promising path. A* can prioritize nodes that are most likely to lead to the shortest path since the heuristic function gives an estimate rather than a guarantee of the precise cost [20].

Since movement is limited to horizontal, vertical and diagonal directions in grid-based pathfinding, we used the **Manhattan** distance heuristic for this implementation.

2.1.2.4 Pathfinding Process:

The algorithm initializes the start node with a g_cost of 0 and an h_cost computed using the Manhattan distance heuristic:

$$h(n) = |x_{\text{goal}} - x_{\text{current}}| + |y_{\text{goal}} - y_{\text{current}}|$$

It then uses a priority queue (heapq), which ensures that nodes are processed in ascending order of their f_cost (f_cost = g_cost + h_cost).
The pathfinding process follows these steps:

1. The node with the lowest f_cost is dequeued.
2. If the goal node is reached, the path is reconstructed.
3. The neighbors of the current node are retrieved, ensuring they are within grid boundaries and not obstacles.
4. The g_cost, h_cost and f_cost of each neighbor are updated.
5. If a neighbor is not in the open list, that contains nodes that are yet to be explored or has a lower f_cost, it is added to the open list.
6. Repeat the process until the goal is reached or no valid path exists.
7. When the path is calculated it is backtraced, converted into real world coordinates and then provided to the **controller** node via the **a_star_planner** node.

2.1.2.5 Handling Edge Cases:

We have taken into consideration and effectively managed a number of edge circumstances to guarantee the A* pathfinding implementation's resilience and reliability.
First of all, before adding a node to the open list, the algorithm verifies that it is within the grid boundaries and free of obstacles (grid[y][x] == 0). This prevents the algorithm from considering out-of-bounds locations or occupied cells, ensuring the computed path remains navigable.
Secondly, we used the closed list that prevents the algorithm from revisiting already processed nodes, avoiding unnecessary computations and infinite loops. Once a node has been fully explored, it is added to the closed list, ensuring that no redundant calculations occur for the same position.
The algorithm logs a warning message and does not proceed with the search if the start or goal node is outside the grid boundaries or is located on an obstacle (grid[y][x] # 0). Additionally, if no valid map data is available the A* algorithm will never be triggered, it is temporarily suspended until a valid map is received. This prevents the system from attempting path planning on an incomplete or missing environment representation.

2.1.3 The controller Node:

The **controller** node is tasked with controlling the robot movement to execute the planned path. It processes odometry data to monitor the robot's position and continuously generates velocity commands. This node ensures efficient path-following while making real-time adjustments based on the robot's orientation and position.

2.1.3.1 Publisher and Subscriber:

The controller node subscribes to the **"/odom"** topic of type odometry to receive real-time updates on the robot's position and orientation. It also subscribes to the **"/real_world_path"** topic to track the robot's actual movement in the real world. Additionally, it publishes to the **"/cmd_vel"** topic to send velocity commands that control the robot's linear and angular movements.

2.1.3.2 Tracking Robot Position and Orientation:

To navigate effectively, the robot must always be aware of where it is and which direction it is facing. For this reason, the robot's position and orientation are continuously updated using data from the **"/odom"** topic, which provides real-time odometry information about the robot's movement. Since odometry messages contain position data in cartesian coordinates (x, y, z) and orientation in quaternion format, some conversions are necessary to properly track the robot's movement. This is achieved through the **odom_callback()** function.
The values **"msg.pose.pose.position.x"** and **"msg.pose.pose.postion.y"** represent

the robot's current position in world coordinates. However, these values are multiplied by -1 to correct the misalignment between RViz and Gazebo's coordinate frames in our setup, which arises due to the absence of GMapping in our implementation.
Additionally, to further align the robot's position with the map's reference frame, adjustment values (-0.75 and +0.75) are applied.These adjustments depend on the robot's initial starting position and how the coordinate system is set up in the navigation environment.
Regarding the robot's orientation, it is represented as a quaternion. However, quaternions are not intuitive for 2D navigation, as direct angle calculations (such as turning toward a goal) become complex. To simplify this, we convert the quaternion into Euler angles (roll, pitch, yaw) using the **euler_from_quaternion()** function. The yaw angle (rotation around the z-axis) is particularly important for navigation, as it determines the robot's heading direction.

2.1.3.3 Handling the Planned Path:

Once the A* algorithm computes an optimal path (Figure 2.1), it is published as a **"PoseArray"** message and received by the **real_path_callback()** function in the **controller** node. This function stores the path as a list of waypoints, enabling the robot to move smoothly from one waypoint in the path to the next. So the Robot can adjust dynamically and continue navigating through the path.

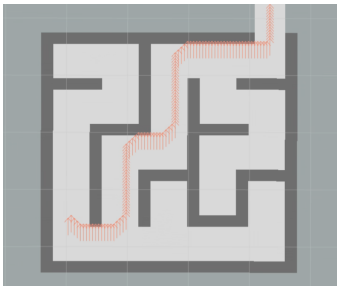


Figure 2.1: Planned path in Rviz

2.1.3.4 Motion Planning and Execution:

After tracking the robot position and orientation and storing the path to the endpoint we will start with the algorithm that we designed to ensure that the robot follows the precomputed path efficiently while dynamically adjusting its movement based on real-time feedback.
The movement logic is based on two key computations: distance to the goal and angle to the goal. The **get_distance_to_goal()** function calculates the **Euclidean** distance between the robot's current position and the next waypoint, determining how far it

needs to travel. The **get_angle_to_goal()** function computes the angular difference between the robot's current heading and the direction of the goal, ensuring the robot turns correctly before proceeding forward.
The **move_to_goal()** function is responsible for generating velocity commands based on these computations. It follows a three-step approach to ensure smooth navigation. First, it aligns the robot, second, it moves the robot forward and third, it transitions to the next waypoint when the robot is 0.22 meters away from the current temporary goal. The robot moves at a constant speed of 0.22 m/s, while its angular velocity is adjusted dynamically based on the angle difference between its current orientation and the next temporary goal.
The execution of this motion planning system is handled by the **run()** function, which acts as the main control loop. Running at 10 Hz, it continuously calls **move_to_goal()**, processes movement commands and publishes them to the **"/cmd_vel"** topic, ensuring the robot remains responsive to real-time position updates.

2.2 Evaluation:

To evaluate the implementation in Challenge 1, tests were conducted on a MacBook Pro equipped with an Apple M3 Pro chip, featuring a 12-core CPU, 18-core GPU and 18GB of RAM. This hardware configuration ensured smooth simulation performance, enabling precise measurement of the robot's execution time while running the navigation system. For Challenge 1, the maximum allowed time to complete the task was 150 seconds. To assess the robot's efficiency, the challenge was tested three times, each using a different maze (Map 8, Map 9 and Map 10). Gazebo was utilized to visualize the robot within the simulated environment and to monitor its movement throughout the maze. The test results are presented in Table 2.1.

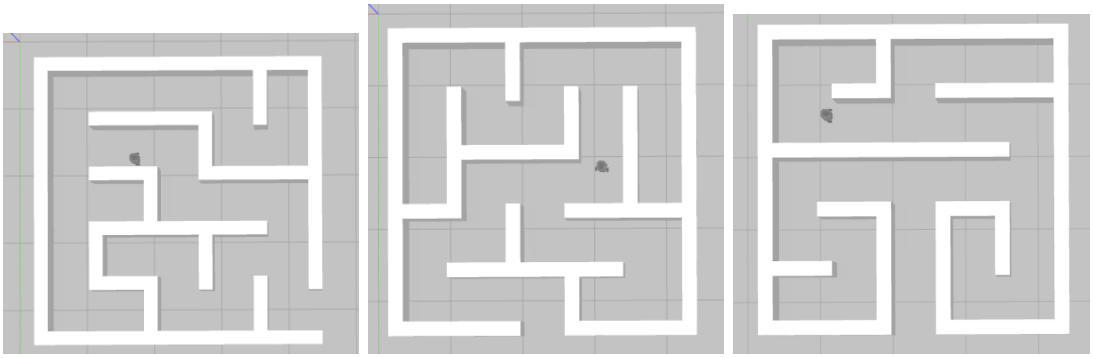
Map	**Time (seconds)**
Map 8	36.80
Map 9	54.27
Map 10	44.32

Table 2.1: Execution time of the robot in different maps in Challenge 1

With these results, the robot successfully completed the challenge within the given time limit, demonstrating effective path planning and execution. Below, Figures 2.2a, 2.2b, and 2.2c illustrate the robot navigating in each of the tested maps.

2.3 Challenges:

During the implementation of Challenge 1, several challenges emerged that required careful adjustments and problem solving. One of the primary issues was with the



(a) Robot in Map 8 (b) Robot in Map 9 (c) Robot in Map 10

Figure 2.2: Visualization of the robot navigating different maps in Gazebo

odom_tracker node, where we had to expand the walls in the occupancy grid to ensure that the robot did not move too close to them, preventing it form getting stuck at corners. In addition, finding the optimal balance between angular and linear speed was challenging, as the robot needed to complete the navigation task within a specific time limit while maintaining stable movement.
Additionally we had to resolve an offset issue in the odometer system, specifically understanding and adjusting the 0.75 unit shift between the coordinate system and the maze model to ensure accurate positioning. Overcoming these challenges allowed us to refine our approach and improve the overall efficiency of our navigation system.

2.4 Limitations:

While implementing Challenge 1, we identified several limitations that influenced the design and execution of our navigation system.
First, all the mazes used for testing were created within the fourth quadrant of the coordinate system. This meant that if we attempted to run our code with a maze positioned differently, it would not work correctly without modifications.
Additionally, the offset between the coordinate system and the Gazebo maze model was treated as a fixed value (0.75). While this worked well for our specific setup, it could require adjustments in different environments or with other maze configurations.
Furthermore, after implementing the A* algorithm, we realized that the path planning process could be accelerated by using a bidirectional search approach with two threads instead of one. One thread would start from the initial position while another would start from the goal merging into one when they meet. This approach could potentially reduce the computation time by half. However, due to time constraints, we only implemented a single-threaded version that searches for the optimal path from the start to the endpoint.

Chapter 3

Challenge 2 and 3:

In Challenge 2 the robot is provided with only the start and goal position and has no prior knowledge of the maze that he will navigate. In Challenge 3 the robot only knows the start point. The robot has to explore the maze in both of the exercises so we have merged both of them into one implementation with a small difference.

3.1 Implementation:

3.1.1 Node Class:

The Node class is a fundamental structure within our maze navigation system, primarily responsible for representing individual waypoints in the exploration and pathfinding process. Each instance of Node encapsulates positional information (x, y), relationships with adjacent nodes (node_1, node_2, node_3), and tracking attributes such as parent (for backtracking). Additionally, number_of_roads is maintained to determine available movement directions at any given node. This class plays a crucial role in mapping out crossroads and decision points within the maze. As the robot navigates, nodes are instantiated dynamically to record intersections where multiple paths are available. The parent-child relationship allows the system to backtrack efficiently if an incorrect path is taken or a dead end is encountered.

3.1.2 Autonomous Movement Execution:

Our algorithm that is responsible for the autonomous movement execution of the robot is presented in the **move()** function. It uses just the laser reading to calculate distances to the walls, to get new temporary goals, and to recognize new roads, if any. It creates nodes and links them as a tree to go back or to decide in which way it will move. The algorithm works as follows:

1. Check if there are any roads in the left or right side: check if the walls are more than 0.58 meters away from the robot, since the roads always have 0.6 as width
2. Check if the current position is more than 0.6 away from the last position you find a road at: if true then set the coordination information of this node and define how many roads (right/left/front) it has.
3. Choose the direction based on the right-hand rule.

4. If the direction left (-1) or right (1) then rotate 90 degree and move forward. If it is (0), then move forward. It may be not found any road to go to or all the roads are already visited at this node then it will go back to the parent.
5. Check if the road is closed: if true then go back to the parent in the last cross road. if not, then do nothing
6. Do all the steps from 1 to 5 until you find the way out.

In this **move** algorithm we used methods to ensure that the robot can explore the maze and reach the goal.

3.1.2.1 Get the Distance from a Temporary Goal:

The **get_distance_to_temp_goal()** function calculates the Euclidean distance between the robot's current position and a specified goal position. This is achieved using the standard distance formula:

$$\text{distance} = \sqrt{(x_{\text{goal}} - x_{\text{current}})^2 + (y_{\text{goal}} - y_{\text{current}})^2}$$

This function is essential for navigation, as it helps determine whether the robot has reached a temporary goal or needs further movement adjustments.

3.1.2.2 Move the Robot to the Goal:

The **move_to_goal()** function is responsible for guiding the robot toward a specified goal position by adjusting both its linear and angular velocities based on real-time distance and orientation data. First, it calculates the Euclidean distance to the goal and the angular difference between the robot's current heading and the goal direction. If the angular difference is large, the robot prioritizes rotation in place to align itself with the goal before proceeding forward. Once the robot is sufficiently oriented, it begins moving toward the goal with a speed proportional to the remaining distance. During forward motion, small angular adjustments are applied to maintain an accurate trajectory. As the robot gets closer to the target, it gradually slows down and fine-tunes its heading for precise arrival.

3.1.2.3 Number of open Roads:

To determine the number of open roads available in front of the robot's current position. We used the **number_of_roads()** function. This function checks whether there is enough space in three primary directions: front (direct), left, and right. If any of these directions have a clearance greater than 0.6 meters, they are considered accessible, and the function increments a counter accordingly.

3.1.2.4 Get the new Orientation:

At intersections, the **get_the_new_orientation()** function guides the robot's direction. It combines pre-measured distances with real-time LiDAR data, deciding a path is clear if it extends beyond 0.58 meters. If the intersection has been previously visited and stored as a node, the function returns -2, meaning that no new movement decision is required.
Otherwise, the function utilizes **laser_dist()** function to detect the first angle where the LiDAR sensor identifies an infinite distance (indicating an open space). If the detected open area falls within 45° to 135°, the robot turns left (+1); if between 225° to 315°, it turns right (-1); and if within 315° to 360° or 0° to 45°, it moves forward (0). If no open space is detected through LiDAR, the robot follows a priority-based approach, first attempting to turn right if possible, otherwise moving forward, and turning left as a last resort.

3.1.2.5 Get a new temporary Goal:

We implemented the **get_new_goal()** function to calculate and set a new temporary goal for the robot based on its current position, orientation and proximity to nearby obstacles. The function first computes the difference between the distance to the nearest wall and 0.3 meters, which acts as a safety buffer to ensure that the new goal is correctly positioned relative to detected obstacles. Next, it analyzes the robot's yaw angle to determine whether it is facing left/right or up/down. Based on this orientation, the function adjusts either the x or y coordinate of the temporary goal while maintaining a safe distance from obstacles.

3.1.2.6 Rotate the Robot:

The **rotate()** function is responsible for rotating the robot in place to align it with the direction of the temporary goal before continuing its navigation. Depending on the orientation of the robot (yaw) (Figure 3.1), the function calculates the rotation by adjusting the x or y coordinates. The robot then begins rotating while continuously adjusting its angular velocity based on the difference between its current heading and the target direction. Once the robot's orientation is within an acceptable threshold, it stops rotating. It also calls the **get_new_goal()** method after rotating to make sure that the robot will directly have a new temporary goal to go to.

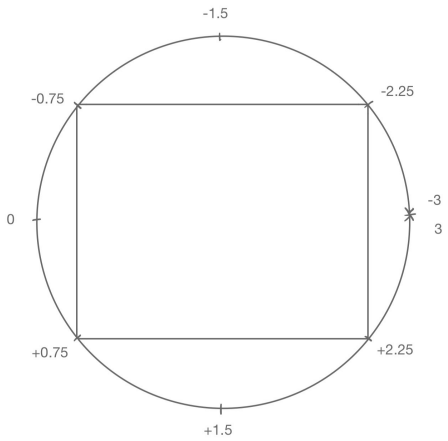


Figure 3.1: Yaw values

3.1.2.7 Robot return to the last Crossroad:

To ensure that the robot can navigate back to the last intersection when needed we used the function **return_to_the_last_crossroad()**. This function retrieves the parent node's coordinates and calculates the Euclidean distance between the robot's current position and the previous crossroads. It then commands the robot to move towards this previous node, publishing velocity commands (cmd_vel) until the destination is reached.

3.1.2.8 Check dead-ends:

We designed the function **check_closed_way()** to detect whether the robot is trapped in a dead point. First, it retrieves distance readings from the robot's laser scanner (distance to the left, right, front) if there are no valid distances it defaults to float('inf'), assuming there is no obstacle detected. And if all three directions (left, right, front) have obstacles closer than 0.4m, the function returns True, indicating a dead-end. Otherwise, it returns False, meaning the robot still has space to move. This function prevents the robot from attempting to move forward when he is in a dead end and it works alongside **return_to_the_last_crossroads()** to find a new path.

3.1.3 Improving Sensor Accuracy with Oscillatory Motion:

Before implementing the **just_once_called()** function, we noticed that the robot sometimes received incorrect coordinate data, causing it to misinterpret its position and surroundings. This led to unexpected navigation errors and unreliable movement. To address this issue, we introduced the **just_once_called()** function, which briefly rotates

the robot clockwise for 1 second and then counterclockwise for 1 second to help recalibrate its sensors. This slight movement allows the robot to refresh its position readings and obtain more accurate data. Since this adjustment is only needed once, the function is designed to be called a single time during operation.

3.1.4 Control Loop:

To ensure that the robot's movement logic is continuously executed, we used the **control_loop()** function, which repeatedly calls the move function. The key difference between Challenge 2 and Challenge 3 lies in the goal that we get from the **"/maze_goal"** topic. If the goal is 0.0 then it indicates it is Challenge 3 and if anything else then it is challenge 2.

3.2 Evaluation:

In Challenge 2 and 3, the tests were conducted using the same hardware configuration as Challenge 1. The robot was tested also on three different maze environments (Map 8, Map 9 and Map 10) and the maximum allowed time to complete the task in this two challenges is 300 seconds. Table 3.1 and 3.2 present the results that we recorded for Challenge 2 and 3:

Map	**Time(seconds)**
Map 8	100,43
Map 9	99,3
Map 10	55,64

Table 3.1: Execution time of the robot in different maps in Challenge 2

Map	**Time(seconds)**
Map 8	95,16
Map 9	108,39
Map 10	55,91

Table 3.2: Execution time of the robot in different maps in Challenge 3

Since the maximum time limit for these two challenges was 300 seconds, the robot successfully completed the task in all tested scenarios. Although execution times were longer than in Challenge 1 due to the lack of precomputed paths, the robot's real-time path planning strategy demonstrated efficiency, as it completed the task well within the allowed timeframe.
Furthermore, the average execution time across the maps in Challenge 2 and Challenge 3 was nearly the same. This consistency is primarily due to the identical implementation

used in both challenges. The minor variations observed between individual runs can be attributed to differences in processor load and resource consumption.

3.3 Challenges:

During the development of our robot's navigation system, we faced several challenges that had to be solved. One major issue was incorrect coordinate data, which caused navigation errors. To fix this, we introduced the **just_once_called()** function, which briefly rotates the robot clockwise and counterclockwise to recalibrate its sensors for more accurate readings. Another challenge was wall avoidance, as the robot sometimes moved too close to obstacles. We addressed this by adding a buffer distance to ensure a safe gap from walls. Additionally, the robot struggled with backtracking in dead-end situations, leading to unnecessary movements. We solved this by implementing the **check_closed_way()** function, which detects confined spaces and triggers **return_to_the_last_crossroads()** to return to the last intersection and find a new path. Lastly, yaw alignment errors caused the robot to drift off course, so we optimized the **rotate()** function with adaptive angular velocity for smoother, more precise rotations. These improvements greatly enhanced the robot's ability to navigate efficiently and avoid unnecessary errors.

3.4 Limitations:

In the mazes that the robot navigates, the wall distance is hard coded to 0.6 meters, which directly influences the calculation of the midpoint. Currently, the midpoint is computed as 0.6/2=0.3 meters. However, if the wall distance were to change, this calculation would no longer be accurate. Since this value is fixed within the code, adapting the system to different environments with varying wall distances would require manual adjustments.

Additionally, our algorithm relies on the left and right distances when generating nodes. As a result, the robot stops, adjusts its direction, and then moves forward when the angle difference to the next temporary goal is too large. If the robot were to move while rotating, it could create incorrect nodes with an inaccurate number of possible paths, potentially leading to the robot getting stuck in a loop.

This limitation prevents the robot from achieving higher speeds, as seen in Challenge 1, where a more efficient navigation strategy was possible.

Chapter 4

Conclusion:

In this project, we successfully developed an autonomous maze-solving robot using the Turtlebot3 Burger platform, ROS1, and the Gazebo simulation environment in three progressively challenging scenarios. Each challenge required distinct navigation strategies, demonstrating the effectiveness of both precomputed path planning and real-time exploration techniques.

For challenge 1, where the maze layout was fully known, we applied the A* algorithm that proved to be an optimal path planning solution, ensuring efficient and collision-free navigation. The robot successfully reached its goal in minimal time, validating the reliability of heuristic-based search methods in structured environments.

In challenges 2 and 3, the robot faced an unknown maze, and that required us to have a more adaptive approach. The robot dynamically mapped its surroundings, identified paths and successfully reached its goals using LiDAR data and the decision-tree-based exploration method.

Throughout the project, we encountered several challenges, including odometry offsets, wall proximity issues and sensor inaccuracies. To overcome these issues, we implemented several algorithmic refinements. Oscillatory motion was introduced to recalibrate the sensors, improving position accuracy. Additionally, wall extension in the occupancy grid was used to prevent the robot from getting too close to obstacles and getting stuck at corners.

Despite these improvements, some limitations remain. Our implementation still relies on predefined maze structures, making adaptation to new environments more challenging.

Future work could focus on integrating more developed exploration strategies, such as reinforcement learning-based decision-making or bidirectional A* search to accelerate path computation. Additionally, implementing real-world testing with hardware beyond simulation could validate the effectiveness of the proposed solutions in a real-world environment. Overall, this project provides valuable insights into autonomous navigation and establishes a strong basis for future advancements in robotic path planning.

Bibliography

[1] ROBOTIS, "Turtlebot3 features," 2025, accessed: March 4, 2025. [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/

[2] J. H. Jung and D.-G. Lim, "Industrial robots, employment growth, and labor cost: A simultaneous equation analysis," *Technological Forecasting and Social Change*, vol. 159, p. 120202, 2020.

[3] V. A. Hajimahmud, A. Khang, V. Hahanov, E. Litvinova, S. Chumachenko, and A. V. Alyar, "Autonomous robots for a smart city: Closer to augmented humanity," in *AI-Centric Smart City Ecosystems*. CRC Press, 2022, pp. 111–122.

[4] S. Alamri, S. Alshehri, W. Alshehri, H. Alamri, A. Alaklabi, and T. Alhmiedat, "Autonomous maze solving robotics: Algorithms and systems," *International Journal of Mechanical Engineering and Robotics Research*, vol. 10, no. 12, pp. 668–675, 2021.

[5] J. A. J. Builes, G. A. Amaya, and J. L. Velásquez, "Autonomous navigation and indoor mapping for a service robot," *Investigación e Innovación en Ingenierías*, vol. 11, no. 2, pp. 28–38, 2023.

[6] Y. Zhang, Y. Zhou, H. Li, H. Hao, W. Chen, and W. Zhan, "The navigation system of a logistics inspection robot based on multi-sensor fusion in a complex storage environment," *Sensors*, vol. 22, no. 20, p. 7794, 2022.

[7] M. Imad, M. A. Hassan, H. Junaid, I. Ahmad, *et al.*, "Navigation system for autonomous vehicle: A survey," *Journal of Computer Science and Technology Studies*, vol. 2, no. 2, pp. 20–35, 2020.

[8] C. Wang, L. Meng, S. She, I. M. Mitchell, T. Li, F. Tung, W. Wan, M. Q.-H. Meng, and C. W. de Silva, "Autonomous mobile robot navigation in uneven and unstructured indoor environments," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 109–116.

[9] F. Gatti, F. Rojas, G. Angeles, R. Contreras, and D. Dobriborsci, "Case study: Autonomous mobile robot exploration and navigation in unknown maze environment," in *2024 10th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, 2024, pp. 1831–1836.

[10] A. S. Romadhon, V. T. Widyaningrum, and A. Dafid, "Implementation of a*(star) algorithm in robots object movement." *Technium*, vol. 16, 2023.

[11] A. Narendran, A. Hothri, H. Saga, and A. Sahay, "Mazesolver: Exploring algorithmic solutions for maze navigation," in *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, 2024, pp. 1–8.

[12] A.-C. Stan, "A decentralised control method for unknown environment exploration using turtlebot 3 multi-robot system," in *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2022, pp. 1–6.

[13] R. Amsters and P. Slaets, "Turtlebot 3 as a robotics education platform," in *Robotics in Education: Current Research and Innovations 10*. Springer, 2020, pp. 170–181.

[14] A.-C. Stan, "A decentralised control method for unknown environment exploration using turtlebot 3 multi-robot system," in *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. IEEE, 2022, pp. 1–6.

[15] H. Maghfiroh and H. Santoso, "Online navigation of self-balancing robot using gazebo and rviz," *Journal of Robotics and Control (JRC)*, vol. 2, no. 5, 2021.

[16] A. V. Babu, A. Krishna M J, S. Damodaran, R. K. James, A. V. Kumar, and T. S. Warrier, "Enhancing mobile robot navigation in turtlebot3 burger: A ros-enabled approach focusing on obstacle avoidance in real-world scenario," in *2024 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, 2024, pp. 1–6.

[17] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[18] P. M. de Assis Brasil, F. U. Pereira, M. A. de Souza Leite Cuadros, A. R. Cukla, and D. F. T. Gamarra, "Dijkstra and a* algorithms for global trajectory planning in the turtlebot 3 mobile robot," in *International Conference on Intelligent Systems Design and Applications*. Springer, 2020, pp. 346–356.

[19] P. M. de Assis Brasil, F. U. Pereira[1], M. A. d. S. L. Cuadros, and A. R. Cukla[1], "Dijkstra and a* algorithms for global trajectory planning in the turtlebot 3," in *Intelligent Systems Design and Applications: 20th International Conference on Intelligent Systems Design and Applications (ISDA 2020) held December 12-15, 2020*, vol. 1351. Springer Nature, 2021, p. 346.

[20] X. Jing and X. Yang, "Application and improvement of heuristic function in a* algorithm," in *2018 37th Chinese Control Conference (CCC)*, 2018, pp. 2191–2194.