Int. j. inf. tecnol.
https://doi.org/10.1007/s41870-025-02500-5

ORIGINAL RESEARCH

Autonomous navigation of ROS2 based Turtlebot3 in static and dynamic environments using intelligent approach

Abhishek Kumar Kashyap[1] · **Kavya Konathalapalli**[1]

Received: 3 December 2024 / Accepted: 3 March 2025

Abstract This study offers a unique strategy for autonomous navigation for the TurtleBot3 robot by applying advanced reinforcement learning algorithms in both static and dynamic environments. With the use of TD3 (twin-delayed deep deterministic), DDPG (Deep Deterministic Policy Gradient), and DQN (Deep Q-Network), real-time object detection, tracking, and navigation can now be done seamlessly by the proposed TD3 algorithms. Additional techniques have been integrated to this project to enhance its mobility performance: ROS 2 (Robot Operating System 2) and LiDAR (Light Detection and Ranging)-based perception. Performance comparison among the above-mentioned algorithms shows that TD3 is the most efficient and robust when exposed to diverse environments. The work further addresses significant gaps in dynamic obstacle navigation and maze resolution, significantly changing the game for robotics applications such as those found in surveillance, human–robot interaction, and inspection. The outcome significantly boosts TurtleBot3's performance and capabilities across various scenarios.

Keywords Robot operating system · Reinforcment learning · TurtleBot3 · Dynamic obstacle

✉ Abhishek Kumar Kashyap
abhishek.kashyap@manipal.edu

[1] Department of Mechatronics, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karnataka 576104, India

1 Introduction

The world of robotics is constantly advancing, with sizeable developments in object tracking and navigation. These skills are important for robots to be able to navigate in a real, self-reliant manner. The aim is to develop an advanced object monitoring device tailored for the TurtleBot3. It is a versatile robotics platform. In many fields like search and rescue operations and warehouse automation, mobile robots are essential for safely and efficiently performing tasks. To effectively navigate in complex environments, these robots need the ability to autonomously track and follow objects. TurtleBot3, a widely used teaching and research robot, offers excellent maneuverability with a payload capacity of up to 15 kg (expandable to 30 kg), and an array of built-in sensors for navigation and localization. This platform uses a Raspberry Pi [1] computer to support ROS 2 and provides a solid framework for building robot software, which facilitates the integration of different components and algorithms. This system can be developed using ROS 2 and Machine Learning algorithms. The project includes the application of the SLAM (Simultaneous localization and mapping) algorithm for navigation in static conditions and the application of a deep reinforcement learning algorithm for conditions with dynamic obstacles. The applications of this program are diverse in terms of modeling, analysis, and human–robot interaction. With a reliable object-tracking system, TurtleBot3 can navigate complex terrain with high accuracy and efficiency, increasing its performance and expanding its capabilities in a variety of environments.

ROS provides an affordable IoT-based waiter robot intended to improve patron interactions and workplace productivity [2]. It can make use of a camera with a high resolution installed on a car, coupled with various

Published online: 25 March 2025

equipment, and is run on an operating system to make it autonomous [3]. To make robots intelligent, the social spider optimization (SSO) method can be implemented in PID (Proportional–integral–derivative controller) controllers [4] to prevent robot fluctuations in speed caused by vehicle position changes [5]. The new online movement and position controller for humanoids in the Virtual Robot Experimental Environment (V-REP) system framework is presented using a nature-based algorithm [6].

2 Literature survey

Author(s)	Methodology	Dataset/ Environment	Key findings	Limitations
Lee and Yusuf [7]	Deep Q-Network (DQN) and Double DQN (DDQN)	Simulated, obstacle-laden environments	DDQN outperformed DQN in terms of reliability and speed	Limited to simulated environments
Algabri and Choi [8]	CNN (Convolutional neural network) trained on color features for indoor human-following	Indoor environment	Accurate human tracking, robust to occlusion	Relies on pre-trained CNN for specific features
Kamble et al. [9]	Object-following robot using AI/ML (Artificial intelligence/ Machine learning) with OpenCV and TensorFlow Lite	Raspberry Pi with camera and sensors	Effective object-following, robust to lighting adjustments	Struggles in complex environments
Park et al. [10]	SLAM combined with motion detection and tracking	Ambiguous environments	Improved SLAM robustness with object tracking	High computational complexity
Eslami and Amjadifard [11]	Model Predictive Control (MPC) for navigation	Dynamic environment	Integrated target tracking and obstacle avoidance	Computational challenges in ambiguous scenarios
Zhang et al. [12]	A* algorithm for dynamic environments	Unknown dynamic environment	Efficient real-time path planning	Limited flexibility in complex situations
Nandkumar et al. [13]	Real-time object detection (RTOD) with CNNs and Haar Cascades	Simulated environment	Cost-effective, reduced dependency on GPS	Vulnerable to lighting and absorption factors
Gulalkari et al. [14]	4-wheel independent steering AGV (Automated guided vehicle) with Kinect and Kalman filtering	Simulated indoor environment	Stable tracking using depth and color data	Limited scalability to outdoor settings
Santiago et al. [15]	Proportional control and artificial potential field (APF) algorithm	ROS 2 system for TurtleBot3	Enhanced route tracking and obstacle avoidance	Requires more robust future implementations
Liu et al. [16]	Path planning using A*, Dijkstra's, RRT (Rapidly exploring random tree), and PRM (Probabilistic Roadmap) algorithms	Simulated environments	Comprehensive assessment of classical and modern methods	Lacks practical implementation details
Kavraki et al. [17]	Trade-offs between grid-based, potential fields, and sampling-based methods	High-dimensional configuration spaces	Highlights algorithm trade-offs in path planning	Limited real-world application insights
Zhang et al. [18]	Genetic Algorithm (GA) for dynamic path planning	Dynamic environments	Flexible and efficient real-time path planning	Computationally intensive in large environments
Zeng and Si [19]	Rapidly-Exploring Random Tree (RRT) algorithm	High-dimensional configuration spaces	Effective for complex, high-dimensional planning	Suboptimal path quality
Amudhavel et al. [20]	Hybrid ACO (Ant colony optimization) and PSO (Particle swarm optimization) algorithm for path planning	Mobile ad hoc networks (MANETs)	Improved path planning using distributed capabilities	Sensitive to hyperparameter tuning
Teimoori and Savkin [21]	Equilateral navigational guidance (ENG) with local barrier avoidance	Dynamic environments	Biologically inspired navigation method	Limited validation in large-scale environments
Khooban et al. [22]	Interval type-2 fuzzy PID control for nonholonomic robots	Simulated environments	Effective trajectory tracking	Requires optimization for real-world applications

The decision to undertake the present project stems from several notable challenges in the field of mobile robot navigation that necessitate attention and resolution. Firstly, existing research on mobile robot navigation often focuses on environments with static obstacles, neglecting scenarios where obscure objects are involved. This oversight presents a significant gap in addressing real-world applications where mobile robots must interact with moving entities. For navigation in an environment with static obstacles, SLAM and NAV2 (ROS 2 Navigation Stack) can be implemented. By using Deep Reinforcement learning algorithms, navigation in a dynamic obstacle environment is made possible. Secondly, while numerous navigation algorithms and techniques have been developed, there remains a lack of exploration specifically tailored to the TurtleBot3 platform. As a result, there is a need to investigate and optimize navigation strategies that align with the capabilities and constraints of TurtleBot3, enabling more effective utilization of this robotic platform in practical scenarios. By performing a comparison analysis on the reinforcement algorithms utilized for this project, one can decide the best algorithms suitable for TurtleBot3. Such capabilities are crucial for applications ranging from surveillance and security to human–robot interaction and service robotics. Therefore, by addressing these challenges and developing a robust tracking and following system for TurtleBot3 in obscure environments, this project aims to fill crucial research gaps and contribute significantly to the advancement of mobile robot navigation technologies.

1. To develop algorithms of reinforcement learning algorithms for object tracking and following using TurtleBot3 within the simulated environment.
2. To use SLAM and NAV2 libraries to simulate autonomous navigation for TurtleBot3 in static environments.
3. To compare multiple algorithms in a static environment and the best among them to test and validate in different static and dynamic environments.

The autonomy pipeline is divided into three main components: perception, planning and control. The perception component involves using 2D planar LiDAR sensor(s) to gather information about the environment, such as identifying cues and detecting obstacles. The planning component involves using an amalgamation of traditional heuristic approaches and modern machine learning algorithms to analyze the gathered data and make decisions about what actions the system should take. The control component involves implementing the necessary software to act on the generated high level behavior decisions or plans to control the robot's movement.

3 Introduction of robotic and simulation platform

Following is a brief summary of the hardware description of the TurtleBot3:

TurtleBot3 (shown in Fig. 1) is a differentially-driven (with a supporting caster wheel) mobile robot platform designed for education, research, and hobby purposes. It is a small, affordable, and customizable robot that can be programmed to perform various tasks. The robot is based on the ROS (Robot Operating System) framework and can be controlled using a laptop or a mobile device. Table 1 presents the features of TurtleBot3 Burger.

3.1 ROS Framework

In terms of the middleware framework used for the development and deployment of the autonomy algorithms for the project (and assignments), our team exploited Robot Operating System 2 (ROS 2), which is the second version of the Robot Operating System (ROS 1). Overall, it is an open-source framework for building robotic software and was designed to address some of the limitations of the original ROS, such as scalability, security, and real-time performance. ROS 2 is built on top of a new middleware layer called Data Distribution Service (DDS), which provides more robust and efficient communication between different components of a robot system. This allows ROS 2 to support a wider range of use cases, from small robots to large-scale distributed systems. That being said, it is to be noted that ROS 2 is a relatively new framework and significantly lacks resources and community support compared to ROS 1 distros.

3.2 Webots simulator

For simulation-based development and testing, the native simulator shipped with the ROS framework called Webots [24] has been used. It is an open-source 3D simulation environment for robotics and autonomous systems. It provides a platform for testing and prototyping robots in a virtual environment before deploying them in the real world. Although it has its limitations in terms of simulation fidelity (both physics as well as graphics), it served as a baseline testing tool for rapid testing and optimization of autonomy algorithms. It supports different programming languages like Python C, C+ + , and ROS. This software is installed in Ubuntu and connected with the ROS 2 with a launch file.

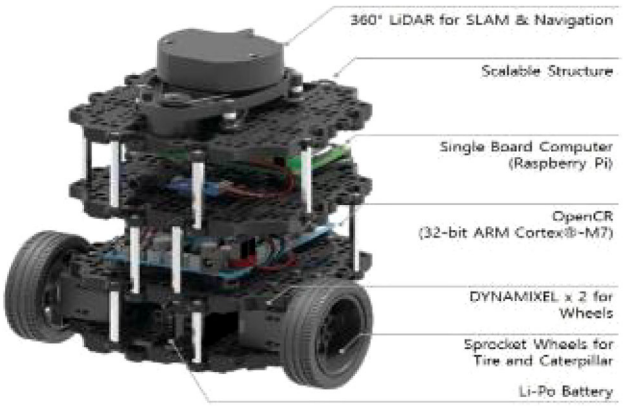

Fig. 1 Representation of Turtlebot3 Burger robot hardware [23]

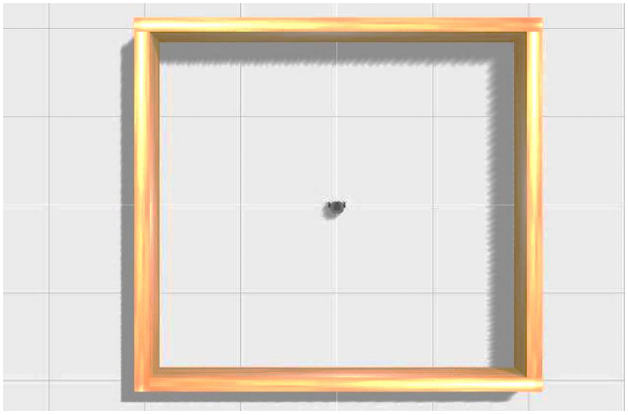

Fig. 2 Static Environment with no obstacles

3.3 ROS visualization

Rviz has become a very important 3D visualization tool that gets used a lot in robotic systems. One cannot debug, test or present research in the field without this tool because it is integrated with ROS. Rviz is known for its unique ability to represent data from a vast range of different types of sensors and sources. Such sensors might include GPS, Sonar sensors, cameras etc. Also, it can visualize camera streams get data from LiDARs, sonar sensors, GPS, robot state and control topics. This helps developers to watch over their robots while they work, as well as see why errors occur in time, leading to better testing during the debugging stage. A significant feature of Rviz is its ability to provide customization for creator purposes such as 3D modeling, laser scans, etc.; interactive menus and markers likewise facilitate easier user interaction. Furthermore, Rviz has a number of other utilities that may be used to analyze and troubleshoot the operations of robots across a wide range of fields. Other sensor simulators, such as cameras and LiDAR, are examples of plug-ins that are available in Rviz. In relation to this, there are also safety assessment tools for path planning. Its capability to integrate with other ROS tools and frameworks is one of Rviz's strengths. A standard interface is provided for accessing and visualizing ROS topic data by Rviz, which makes it easy to integrate with other ROS applications and nodes, hence making it one of its strengths. Hence, a user can extract real-time analysis of the robot data by leveraging other ROS tools and libraries through visualization.

4 SLAM and NAV2 implementation for static environments

4.1 Implementation of SLAM

SLAM (Simultaneous Localization and Mapping) is a technique widely used in robotics to create maps of unknown environments while simultaneously localizing the robot within that environment. To generate a map of the environment using SLAM, we first need to deploy a robot equipped with sensors capable of mapping its surroundings. This can be achieved by running a SLAM (Simultaneous Localization and Mapping) algorithm. These algorithms utilize sensor data, typically from a laser scanner, to build a 2D or 3D representation of the environment while simultaneously estimating the robot's pose

Table 1 Features of TurtleBot3 Burger

Items	Burger
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Maximum payload	15 kg
Size (L x W x H)	138 mm × 178 mm × 192 mm
Weight (+ SBC + Battery + Sensors)	1 kg
SBC (Single Board Computers)	Raspberry Pi
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Actuator	XL430-W250
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01 or LDS-02
IMU	Gyroscope 3 Axis

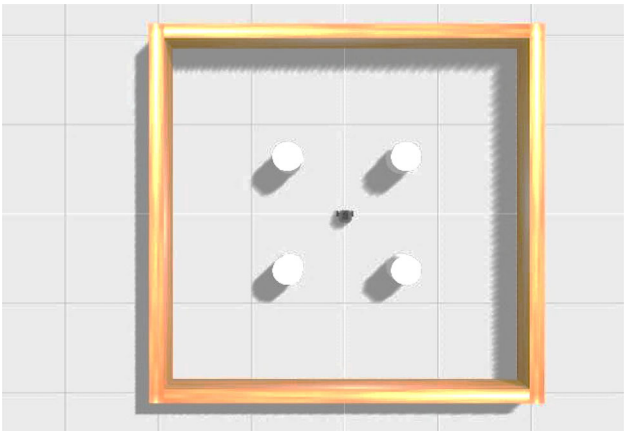

Fig. 3 Static Environment with few obstacles

within that map. Figures 2 and 3, shown below, are the environments with no obstacles and static obstacles, where the SLAM algorithm is to be performed. Continuously mapping the surroundings involves updating maps whenever new data is generated by sensors during the teleoperation of the robot environment. TurtleBot3 can run various SLAM algorithms with the well-known one being GMapping. GMapping is a very popular implementation of the 2D SLAM problem, employing particle filters to create an occupancy grid map and position a robot within the map generated this way. Such an approach is especially applicable when it comes to indoor environments that require only 2D maps.

The primary advantage of GMapping is its accuracy and speed that makes it possible for the algorithm to work as a fundamental solution used in many devices used in practice. This is due to the extensive support the ROS ecosystem has, such as its extensive documentation plus an active community that makes it user-friendly ROS has a large number of users who contribute to its development through forums thus, GMapping benefits from this network of developers. The robot navigation stack uses cost maps; it downloads maps of its environment; robots are representations of the world. The Robot Operating System (ROS) was made as a useful tool for robot application programmers but now also supports this area while facilitating a process through which robots can move from one point in an environment to another without colliding with obstacles. The robot operating system ecosystem provides a lot of help and guidelines on GMapping, which is required to be set up and used by many people who never practiced SLAM algorithms. This is because, for various common situations, a 2D map suffices, and this explains why most TurtleBot3 users still consider GMapping to be a significant tool. Such use cases have been exhaustively checked using different technologies, showing how stable they can be. This is explained and presented in Fig. 4.

TurtleBot also supports various kinds of SLAM. Included among these are:

- Cartographer designed by Google supports 2D and 3D SLAM, making highly detailed maps and very robustly locating itself. As a result of being able to conduct loop closures within it, this algorithm performs particularly well in complicated and changing environments, which may consequently reduce the error margin when Mapping. Due to this feature, the system can recognize places that have been visited before. Cartographers have sensor fusion capabilities that enable the integration of data from different sources, improving the accuracy of the maps they make. However, configuring the cartographer is more difficult, and it demands more processing power resources, especially if you are into 3D mapping, which makes it suitable for applications that require detailed maps but could be difficult to use on low-power platforms.
- Hector Mapping is a stand-out 2D SLAM algorithm that lacks wheel odometry; hence, it is amazing for robots with less or no odometry data. It is largely based on laser scan matching, and it can make maps that are updated at high rate, which makes it good for moving robots quickly. However, its dependence on laser scans may pose some limitations in specific situations despite the fact that Hector Mapping is able to tolerate rapid motions and dynamic environments effectively. Like GMapping, 2D Mapping is one of the constraining factors. It's mostly used in situations in which odometry might fail to provide accurate information, leading to accurate maps being derived without the need for data from wheel encoders.
- Frontier Exploration is an exploration strategy specifically designed for autonomous investigation of unfamiliar environments. It does this by locating frontiers, which are the lines separating already known territory from uncharted territories, and moving the robot to such areas in order to grow the zone already mapped out. This technique is very efficient in exploration activities since it helps to cover new areas, taking little time for mapping out. Frontier Exploration depends on the SLAM algorithm for Mapping. This means that the quality of the SLAM algorithm directly affects its performance. Despite its seamless integration with the ROS navigation stack, its exploration efficiency changes depending on the environment and how well SLAM is implemented.
- Karto is a graph-based 2D SLAM algorithm known for its efficient optimization and loop closure capabilities. It creates accurate occupancy grid maps by optimizing the map based on graph structures, which helps reduce errors and improve consistency. Karto's strength lies in

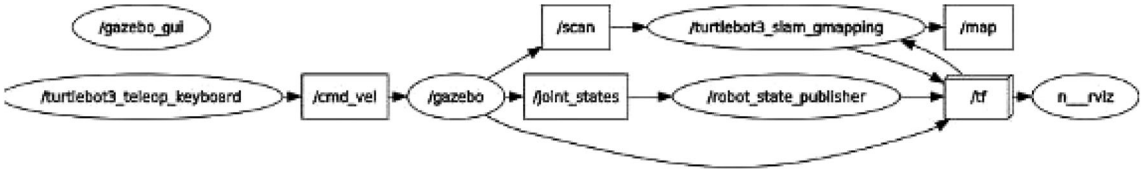

Fig. 4 Rqt_Graph generated for SLAM using G-Mapping algorithm

its ability to perform efficient loop closure, reducing drift over time and enhancing the map's accuracy. However, it is limited to 2D Mapping and does not have as much community support as some other SLAM algorithms, which can make troubleshooting and implementation more challenging. Karto is well-suited for indoor mapping applications where efficient and accurate 2D Mapping is required.

Once the map is generated using SLAM, we will save the map for future reference and use in localization-based navigation. Figure 5 shows the map generated using SLAM for environment with no obstacles and Figure 6 for the environment with Static Obstacles.

4.2 Localization

Localization involves determining the robot's position on the existing map using sensor data and odometry information. Unlike SLAM, localization does not update the map but rather relies on the pre-existing map to determine the robot's position accurately. By saving the map

Fig. 5 Map generated for environment with no obstacles

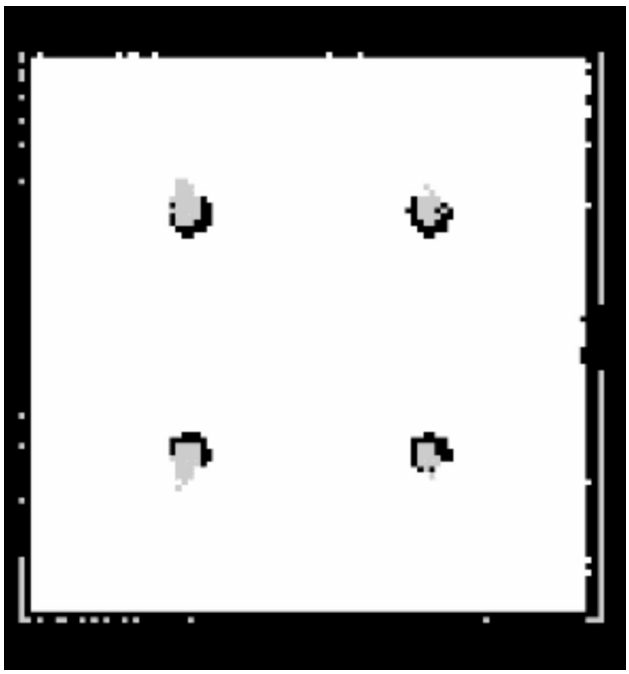

Fig. 6 Map generated for environment with static obstacles

generated through SLAM, we can utilize it for localization-based navigation, enabling the robot to navigate through known environments more efficiently and reliably. Navigation enables a robot to move from the current pose to the designated goal pose on the map by using the map, the robot's encoder, the IMU sensor, and the distance sensor. For its localization requirements, TurtleBot3 mainly uses the AMCL (Adaptive Monte Carlo Localization) method. It is a probabilistic concept that depends on particle filters to determine the pose (position and orientation) of a robot. It essentially retains some likely states (particles) and adjusts them with sensor data and motion models. The algorithm uses particle filters in such a way that every particle corresponds to one potential robot pose. The weights of these particles depend on how the sensor information, like laser scans, fits onto a map with each of those poses. Resampling refers to a technique that involves selecting particles with high probability so that they can help concentrate computing resources on these believed true locations in the next generation. Adaptivity is an outstanding feature since this algorithm can alter the number of particles in response to the degree of likeness between the robot pose and its

Table 2 Parameters of DWA algorithm

Parameter	Description
Inflation_radius	This parameter makes inflation area from the obstacle. Path would be planned in order that it don't across this area. It is safe that to set this to be bigger than robot radius
cost_scaling_factor	This factor is multiplied by cost value. Because it is an reciprocal propotion, this parameter is increased, the cost is decreased
max_vel_x	This factor is set the maximum value of translational velocity
min_vel_x	This factor is set the minimum value of translational velocity
	If set this negative, the robot can move backwards
max_trans_ve	Actual value of the maximum translational velocity. The robot can not be faster than this
min_trans_vel	Actual value of the minimum translational velocity. The robot can not be slower than this
max_rot_vel	Actual value of the maximum rotational velocity. The robot can not be faster than this
min_rot_vel	Actual value of the minimum rotational velocity. The robot can not be slower than this
acc_lim_x	Actual value of the translational acceleration limit
acc_lim_theta	Actual value of the rotational acceleration limit
xy_goal_tolerance	The x,y distance allowed when the robot reaches its goal pose
yaw_goal_tolerance	The yaw angle allowed when the robot reaches its goal pose
sim_time	This factor is set forward simulation in seconds. Too low value is in sufficient time to pass narrow area and too high value is not allowed rapidly rotates. You can watch differences of length of the yellow line in below image

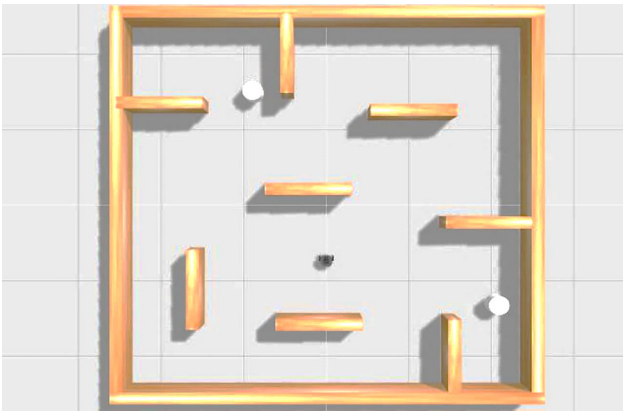

Fig. 7 Representation of environment with dynamic obstacles

current state, hence allowing much processing power to be saved in AMCL. The rationale for selecting AMCL as a localization package for TurtleBot3 is multifold. AMCL has been credited with high precision and consistency in dealing with noisy sensor information, as well as improperly functioning motion models typically faced by actual systems. AMCL is designed to use computational power efficiently by balancing accuracy and time. So, regarding the availability of different sizes of space, such as small indoor spaces or bigger places with many corners inside, this adaptation makes it suitable for various situations. Moreover, being an integral part of the ROS ecosystem adds more weight to its benefits.

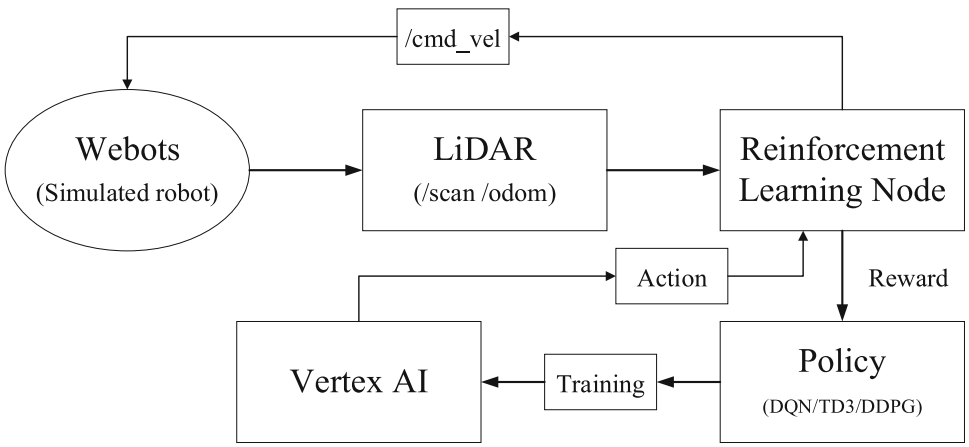

Fig. 8 Workflow of the Reinforcement Algorithm working

Table 3 Description of reward/ penalty parameters

Parameters	Reward / Penalty
Yaw_Reward ('r_yaw')	Penalizes large goal angles with a range of [− 3.14, 0]
Angular Velocity Reward ('r_vangular'):	Penalizes high angular velocities with a range of [− 4, 0]
Distance Reward ('r_distance'):	Rewards reduction in distance to the goal with a range of [− 1, 1]
Obstacle Distance Reward ('r_obstacle')	Penalizes proximity to obstacles with a significant penalty of − 20 if the distance is less than 0.22 m
Linear Velocity Reward ('r_vlinear')	Penalizes deviations from a linear velocity target, calculated as a quadratic function, with a range of [$-2*(2.2^2)$, 0]
Success	A large reward of 2500
Collision	A large penalty of − 2000

AMCL is appropriate for localization in TurtleBot3 and other robots using the ROS framework due to its ready packaging and detailed manuals, making implementation and customization more straightforward and ideal for that purpose. AMCL means a likely localization plan that determines the place and direction of a machine on a map it knows through promoting resembling existing particle groups and adjusting their weights due to constant sensory data changes. The necessary parameters are configured in this file for launching a localization node that subscribes to sensor data, such as laser scans. This subscription is then used to estimate the robot's pose on a given map. In order to achieve the desired accuracy when navigating between two points (Nav2), aspects of this file, such as sensor information, particle filter configuration options, or map material, should be taken into account. TurtleBot3 combines Navigation2 (Nav2) and Adaptive Monte Carlo Localization (AMCL), thus improving from its earlier versions with a more efficient way towards a certain goal. It maintains accurate position awareness through AMCL, which estimates the robot's pose using particle filters and updates them depending on sensor data and motion models for real-time localization, which is important to the successful manner of moving. This localization data is used by Nav2 for path planning using advanced algorithms like DWA, thereby guaranteeing optimum routes in spite of obstacles during movement. This approach makes it possible to automatically move around complicated physical spaces using TurtleBot3 through the combination of real-time sensor feedback with dynamic path planning through ROS 2. The parameters of the DWA algorithm is presented in Table 2.

4.3 NAV2 algorithm

The DWA is a strategy for local path planning intended for mobile robots meant to move about an environment that is undefined and subject to change with no warning whatsoever. It differs from global planners such as A* in that it operates on a local level by taking into account the robot's current state, sensor data, and near environment. It looks at likely paths to follow by trying out what will happen if the robot moves through different places in some time to come. This is done through a "window" with various achievable velocities and direction rates where the path is the best. DWA selects the trajectory that optimizes both reaching the goal and avoiding obstacles, ensuring smooth and safe navigation. This approach is computationally efficient and particularly effective for real-time applications, where quick reactions to changing environments are essential for mobile robots like TurtleBot3. Initial Pose Estimation must be performed before running the Navigation as this process initializes the AMCL parameters that are critical in Navigation. TurtleBot3 has to be correctly located on the map with the LDS sensor data that neatly overlaps the displayed map. The robot will create a path to reach to the Navigation Goal based on the global path planner. Then, the robot moves along the path. If an obstacle is placed on the path, the navigation will use a local path planner to avoid the obstacle. Setting a Navigation Goal might fail if the path to the Navigation Goal cannot be created. If you wish to stop the robot before it reaches the goal position, set the current position of TurtleBot3 as a Navigation Goal.

5 Object tracking and navigating in dynamic environment

TurtleBot3 robot is equipped with sensors and actuators. Key sensors include a LiDAR scanner and odometry sensors that provide crucial data for navigation and obstacle detection in the environment, as shown in Fig. 7. The use of reinforcement learning helps the robot even drive in an unknown environment to track the goals. The workflow of the process is shown in Fig. 8. The robot sends sensor data to ROS2 topics. The /scan topic provides LiDAR scan data, which contains information about the surrounding environment and obstacles. The */odom* topic provides odometry data detailing the robot's position and movement over time.

5.1 DRL Environment node

This node simulates an environment where a DRL agent interacts with a robot. It scans the sensor data (for scanning) and odometry (*/odom*) and then feeds these inputs into the DRL agent generator. Environment is responsible for assigning rewards or penalties depending on the outcome achieved by robot, avoiding obstacles and catching/achieving the target location/destination in most cases. The (DRL) environment for TurtleBot3 that performs obstacle avoidance and goal tracking is set up using Python code. The '*DRLEnvironment*' class is set up as a ROS2 node that subscribes to multiple topics. Some of the topics it subscribes to include *'/scan*' for LiDAR data, *'/odom*' for odometry and '*goal_pose*' for goal positions. In addition, it subscribes to the '/clock' topic used for keeping track of the simulation time and also listens on an '*obstacle/odom*' topic that provides the current position of obstacles. Publishers for '*cmd_vel*' are made available by the class to communicate velocity commands to the robot (as shown in Fig. 8).

The '*goal_pose_callback*' function updates the robot's goal position whenever a new goal is received. On the other hand, the '*odom_callback*' updates the current position of the robot and its heading based on odometry data, calculates the distance and angle to the goal, and then updates the previous position of the robot to keep track of the total distance traveled by the robot. Meanwhile, the '*scan_callback*' processes LiDAR data in order to normalize the distance measurements and detect the closest obstacle. The '*obstacle_odom_callback*' function tracks the positions of dynamic obstacles by computing their distances from the robot. The DRL environment node serves as an intermediary between the robot's real-time data and the DRL agent. The '*step_comm_callback*' function is crucial as it handles the communication between the environment and the DRL agent. It initializes the episode by resetting the environment and providing the agent with the initial state. During each step, it applies motor noise to the actions for robustness, computes the linear and angular velocities from the agent's actions, and publishes these commands to move the robot. The '*get_state*' function constructs the state vector, which includes the robot's sensor readings, the distance and angle to the goal, and the previous actions. It also checks if it has reached the goal, collided with obstacles, timed out, or tumbled so as to look for termination conditions. Based on these conditions, the reward is computed to steer the agent towards the best acting. On reaching the goal or encountering a failure condition, '*stop_reset_robot*' function stops the robot and adjusts the difficulty radius for the next episode. The DRL agent refines its policy through the state and reward information is constantly updated within the DRL environment.

5.2 Reward observation

The reward bot's reward would be computed by the DRL environmental node based on its present circumstances and activities. The goal tracking and avoidance agent in charge of training requires this compensation due to this offering information in terms of how it undertakes movements in between tasks while at the same time evading obstacles. The following code (in HTML) defines the needful distance to the objective and stores it as a global variable '*goal_dist_initial*' to enable calculations of rewards based on distances, which are critical. Besides, it sets '*reward_function_internal*' variable to the wanted reward function depending on the setting in configuration ('REWARD_FUNCTION'). A wrapper "*get_reward*" contains a call of the internally defined reward function. One of the available reward functions, '*get_reward_A*' calculates several constituent values of the reward on the basis of the robot's actions and state. Such components are included in Table 3. The total reward is the sum of these components minus a constant penalty of 1. Additionally, the reward is adjusted based on the outcome of the episode. At the beginning of each episode, the function '*reward_initalize*' is used to set the target goal distance and finally, the script assigns the function specified by '*REWARD_FUNCTION*' to '*reward_function_internal*', presented in Table 3. In error, the function may not exist, leading to an error message. This design provides an opportunity to configure the reward function differently, hence facilitating experiments on varied award strategies.

5.3 Policy

The agent can be trained on three reinforcement algorithms: DQN, DDPG, TD3.

5.3.1 DQN:

Deep Q-Network (DQN) [25] is a reinforcement learning algorithm that combines Q- learning with deep neural networks. The goal of DQN is to find an optimal policy for an agent interacting with an environment, maximizing the cumulative reward over time. The Q-network approximates the Q-value function, which estimates the expected utility of taking a given action in a particular state and following the optimal policy thereafter. In DQN, the Q-value function is parameterized by a neural network, and the target Q-value function is updated periodically to stabilize training. The key components of the DQN algorithm include experience replay and target network updates. Experience replay involves storing the agent's experiences (state, action, reward, next state, and done flag) in a replay buffer and sampling mini-batches of experiences to update the Q-network. This breaks the temporal correlations and leads to better convergence. The target network, a copy of the Q-network, is updated less frequently to provide stable targets during the Q-value updates. In this code, we define an agent for controlling a TurtleBot3 robot. The 'Actor' class represents the neural network architecture used to approximate the Q-values. It consists of three fully connected layers ('fa1', 'fa2', and 'fa3') with ReLU activations. The forward pass through this network computes the Q-values for the given states. If 'visualize' is enabled, the network can also visualize the selected actions and intermediate activations. The 'DQN' class, inheriting from '*OffPolicyAgent*', encapsulates the DQN algorithm. During initialization, two networks are created: 'actor' and '*actor_target*'. The 'actor' network is the primary Q-network, while the '*actor_target*' is the target Q-network, updated periodically using a hard update method. The optimizer for the 'actor' network is also initialized. The '*get_action*' method selects an action based on the current state. During training, the agent uses an epsilon-greedy policy to balance exploration and exploitation. With probability epsilon, it selects a random action; otherwise, it selects the action with the highest Q-value. The '*get_action_random*' method returns a random action. The 'train' method updates the Q-network. It computes the target Q-values using the reward and the maximum Q-value of the next state from the target network. The loss is calculated using Mean Squared Error (MSE) between the predicted Q-values and the target Q-values. Backpropagation is performed to minimize this loss, and the gradients are clipped to prevent exploding gradients. The target network is updated periodically based on the '*TARGET_UPDATE_FREQUENCY*'. This approach allows the TurtleBot3 to learn effective navigation strategies through trial and error, ultimately improving its performance in autonomous tasks. The description of the method is presented in the necessary steps.

5.3.1.1 Algorithm 1: DQN implementation of motion planning

1. Initialization
- Set up a single Q-network using weights that are chosen randomly.
- Establish an objective Q-network that has identical initial weights as the Q-network.
- Set up a waiting period for experience playback.

2. Training Loop

a. For every incident:

b. State Evaluation: Use odometry and lidar to determine the robot's current state.

c. Choosing an Action: Adopt a greedy epsilon strategy (presented in Eq. (1)):****

a. $$a = \begin{cases} \text{random action} & \text{if } \varepsilon > \text{random value} \\ \arg\max_a Q(s,a) & \text{otherwise} \end{cases} \quad (1)$$

d. Communication with the Environment: Perform the action, then watch for the reward and subsequent state.

e. Store Changes: Put the tuple (s, a, s', a') into the buffer for replay.

f. Mini-Batch Selection: Take an example of the replay buffer's changes.

g. Q-Network Update: Use the Bellman formula to calculate the desired Q-value (as presented in Eq. (2)

a. $$Q_{\text{target}} = r_t + \gamma \max Q'(s', a) \quad (2)$$

h. Reduce the average squared error among the target and expected Q-values (presented in Eq. (3)):

a. $$\text{Loss} = \mathrm{E}\left[\left(Q(s,a) - Q_{\text{target}}\right)^2\right] \quad (3)$$

i. Objective Network Improvement: Apply the Q-network parameters to the target network on a regular basis.

3. Output
- A distinct navigation strategy that is best suited for tracking targets and avoiding obstacles.

Table 4 Comparison between RL algorithms based on various aspects

Aspect	DQN (Deep Q-Network)	DDPG (Deep deterministic policy gradient)	D3 (Twin delayed DDPG)
Action space	Discrete	Continuous	Continuous
Core idea	Uses Q-learning with a neural network to estimate Q-values	Combines actor-critic methods with deterministic policy gradient	Enhances DDPG by addressing overestimation bias and stabilizing learning
Networks	Single Q-network	Actor and Critic networks	Actor and two Critic networks (double Q-learning)
Exploration strategy	ε\epsilonε-greedy policy	Noise processes (e.g., OUNoise) added to the action	Adds noise to the target action and clips it
Q-value estimation	Single Q-value estimation	Single Q-value estimation	Double Q-value estimation to reduce overestimation bias
Update frequency	Regularly update both the Q-network and target networks	Regular updates with soft target updates (using τ\tauτ)	Critic updated more frequently; Actor updated less frequently to reduce variance (policy update frequency)
Policy type	Off-policy	Off-policy with deterministic policy	Off-policy with deterministic policy
Target network	Yes, it is used to stabilize training	Yes, soft updates of target networks	Yes, soft updates with two Critic networks
Algorithm complexity	Moderate	High, due to the need for both actor and critic networks and the addition of noise processes	Higher, due to the use of two Critic networks and additional noise in target policy
Learning rate	Varies, typically 10^{-3} to 10^{-4}	Actor: 10^{-4}, Critic: 10^{-3}	Similar to DDPG, with separate rates for Actor and Critic networks
Batch size	Typically, 32–64	Typically, 64–128	Typically, 100
Experience replay size	10,00,000.00	10,00,000.00	10,00,000.00
Policy noise	None	Added through noise processes	Added to target action, typically Gaussian noise (policy noise 0.2, noise clip 0.5)
Overestimation bias	Prone to overestimation bias	Prone to overestimation bias	Mitigates overestimation bias with clipped double Q- learning
Sample efficiency	Moderate	Higher than DQN, leveraging experience replay and actor-critic methods	Similar to DDPG, but with more stable and efficient updates due to double Q- learning

5.3.2 DDPG

The Deep Deterministic Policy Gradient (DDPG) algorithm is an off-policy [26] actor-critic method used for continuous control tasks. DDPG combines the benefits of both Deep Q- Networks (DQN) and deterministic policy gradients, making it well-suited for tasks where actions are continuous rather than discrete. DDPG uses two neural networks called Actor and Critic. One of these networks, the Actor-network, chooses what action the agent should take when given an input state, whereas the other one estimates its quality with respect to the expected reward (Q-value).

5.3.2.1 Algorithm 2: DDPG implementation of motion planning

1. Initialization
- Set arbitrary weights at the beginning of the actor and critic networks.
- Target networks should be configured as duplicates of the source networks.
- Set up the Ornstein–Uhlenbeck (OU) distortion and replay cache for the investigation.

2. Training Loop

a. Monitor the robot's orientation using LiDAR *(/scan)* and odometry *(/odom)*

b. Use classes for each have been set up separately in the source code under different names, such as Actor and Critic, respectively, for both

Table 5 Stability, advantages and disadvantages of RL algorithms

Aspect	DQN (Deep Q-Network)	DDPG (Deep Deterministic Policy Gradient)	TD3 (Twin Delayed DDPG)
Training Stability	Stable for discrete actions but can struggle with continuous actions	Less stable than DQN due to sensitivity to noise and hyperparameters	More stable than DDPG, addressing overestimation bias and reducing variance in policy updates
Advantages	Effective in discrete action spaces Simplicity and ease of implementation Robust against small perturbations	Effective in continuous action spaces Can learn deterministic policies Suitable for high- dimensional action spaces	Reduces overestimation bias with double Q-learning Improved stability and performance over DDPGEffective in continuous actions
Disadvantages	- Inefficient in continuous action spaces - Overestimation bias in Q-values High sample complexity	- Overestimation bias in Q- values - Sensitive to hyperparameters Requires careful tuning of exploration noise	Increased complexity compared to DDPG Requires more computational resourcesSensitive to hyperparameters

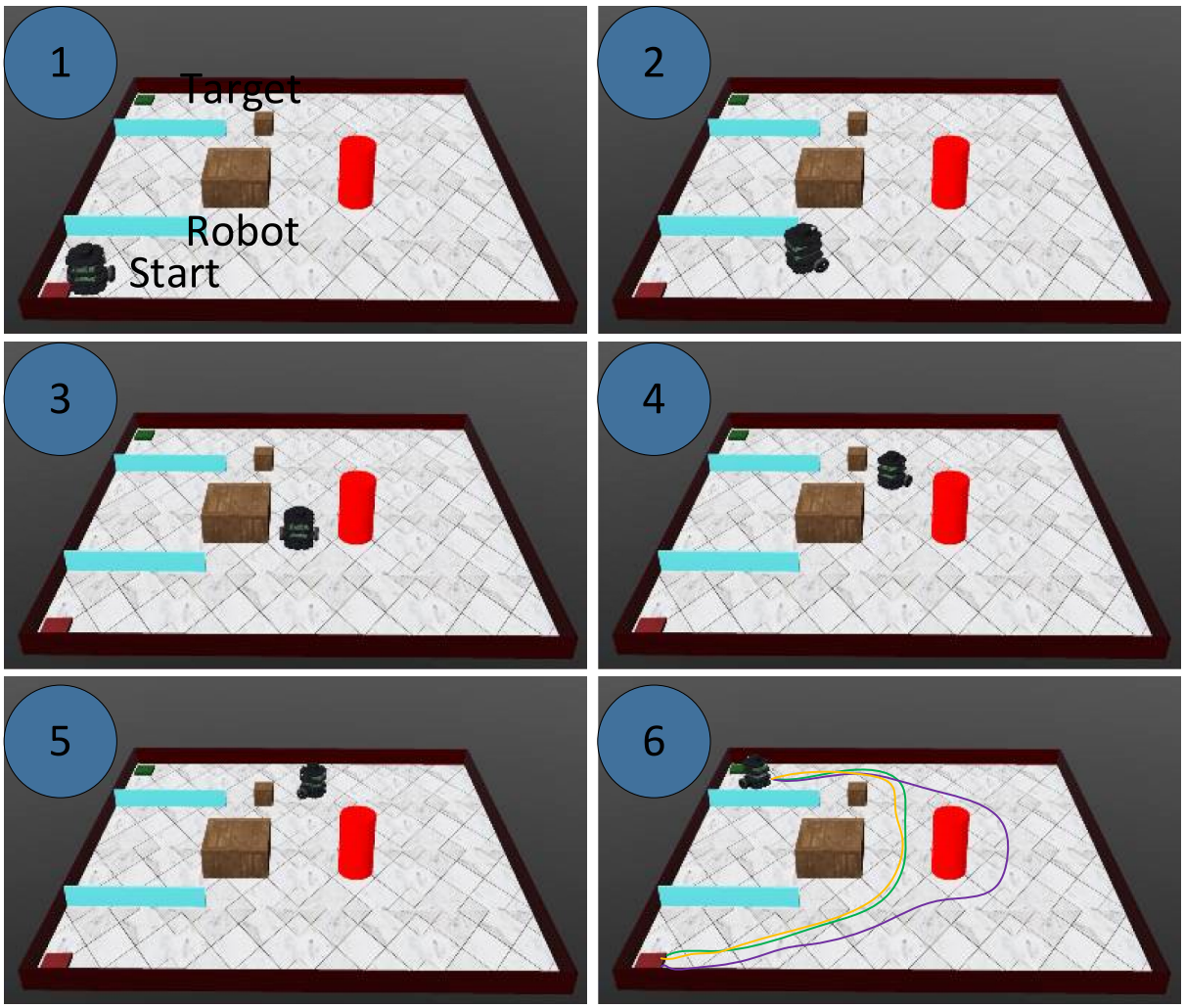

Fig. 9 Implementation of Nav2 in environment with static obstacles using TD3 (Yellow), DDPG (Green) and DQN (Violet)

networks of their kind. The Actor network consists of three fully connected layers presented in Eq. (4). The input layer (fa1) processes the state, followed by two hidden layers (fa2 and fa3) that output the action.

$$\text{action} = \tanh(w_3 \cdot \text{ReLU}(w_2 \cdot \text{ReLU}(w_1 \cdot \text{state} + b_1) + b_2) + b_3) \tag{4}$$

c. Use the Critic network has a more complex architecture, with separate paths for the state and action inputs. Depending on the action, the state input is presented in Eq. (5). It goes through a linear layer (l1), and the action input goes through another linear layer (l2).

$$Q(\text{state}, \text{action}) = w_4 \cdot ReLU(w_3 \cdot [ReLU(w_1 \cdot \text{state} + b_1) \parallel ReLU(w_2 \cdot \text{action} + b_2)] \tag{5}$$

The outputs of these two layers are concatenated and passed through additional layers (l3 and l4) to produce the Q-value estimate. In the TurtleBot3 environment, DDPG is employed to learn a policy that enables the robot to navigate toward a goal while avoiding obstacles.

d. The DDPG class inherits from OffPolicyAgent, initializing the Actor and Critic networks, their corresponding target networks, and optimizers. Ornstein–Uhlenbeck noise (OUNoise) is used to add exploration during training by perturbing the action outputs.

e. In the get_action method, the Actor-network generates an action based on the current state. If training, noise is added to the action to encourage exploration. The action (presented in Eq. (6)) is then clamped to the range $[-1, 1]$.

$$\text{action}_{\text{train}} = \text{clamp}\ (\text{action} + \text{noise}, -1, 1) \tag{6}$$

f. During training (train method), the Critic network is optimized first. The next state's action is predicted using the target Actor-network (presented in Eq. (7)), and the Q-value for the next state-action pair is estimated using the target Critic network.

$$\begin{aligned} Q_{\text{target}} &= \text{reward} + \gamma \cdot (1 - \text{done}) \\ &\cdot Q_{\text{critic\_target}}(\text{state}_{\text{next}}, \text{action}_{\text{next}}) loss_{\text{critic}} \\ &= E\left[(Q(\text{state}, \text{action}) - Q_{\text{target}})\right] \end{aligned} \tag{7}$$

The target Q-value (Q_target) is calculated using the reward, discount factor, and the estimated Q-value of the next state. The Critic network minimizes the difference between its current Q-value and the target Q-value.

g. The Actor network is then optimized to maximize the expected return as predicted by the Critic. This is done by minimizing the negative (presented in Eq. (8)) of the Critic's Q- value output for the predicted action.

$$\text{loss}_{\text{actor}} = -E[Q(\text{state}, \text{actor}(\text{state}))] \tag{8}$$

h. Both Actor and Critic target networks are updated slowly using a soft update mechanism, where the target network parameters (presented in Eq. (9)) are slowly adjusted towards the current network parameters based on a factor (tau).

$$\theta' = \tau\theta + (1 - \tau)\theta' \tag{9}$$

where θ and θ′ in Equation (9) are the parameters of the original and target networks, respectively.

3. Output

- TurtleBot3's optimal navigation strategy in constant motion spaces.

DDPG is ideal for continuous control tasks such as robotic arm manipulation, autonomous driving, and in this case, TurtleBot3 navigation. It excels in environments where actions are not discrete but need to be fine-tuned continuously for optimal performance.

5.3.3 TD3

The TD3 (Twin Delayed Deep Deterministic Policy Gradient) [27] algorithm is an enhancement of the DDPG (Deep Deterministic Policy Gradient) algorithm, designed to stabilize training and improve performance in continuous action spaces. TD3 introduces three key improvements over DDPG: clipped double Q-learning, target policy

Table 6 Evaluation TD3 with respect to DDPG and DQN

Metric	DDPG	DQN	TD3	Deviation (%) of TD3 w.r.t	
				DDPG	DQN
Navigation Time to Target	23.4 s	27.8 s	20.1 s	14.1	27.69
Average Number of Collisions/Episode	2.1	3.4	1.5	28.57	55.88
Success Rate	78%	64%	85%	8.97	32.81
Average Reward per Episode	42.3	37.8	45.9	8.51	21.43

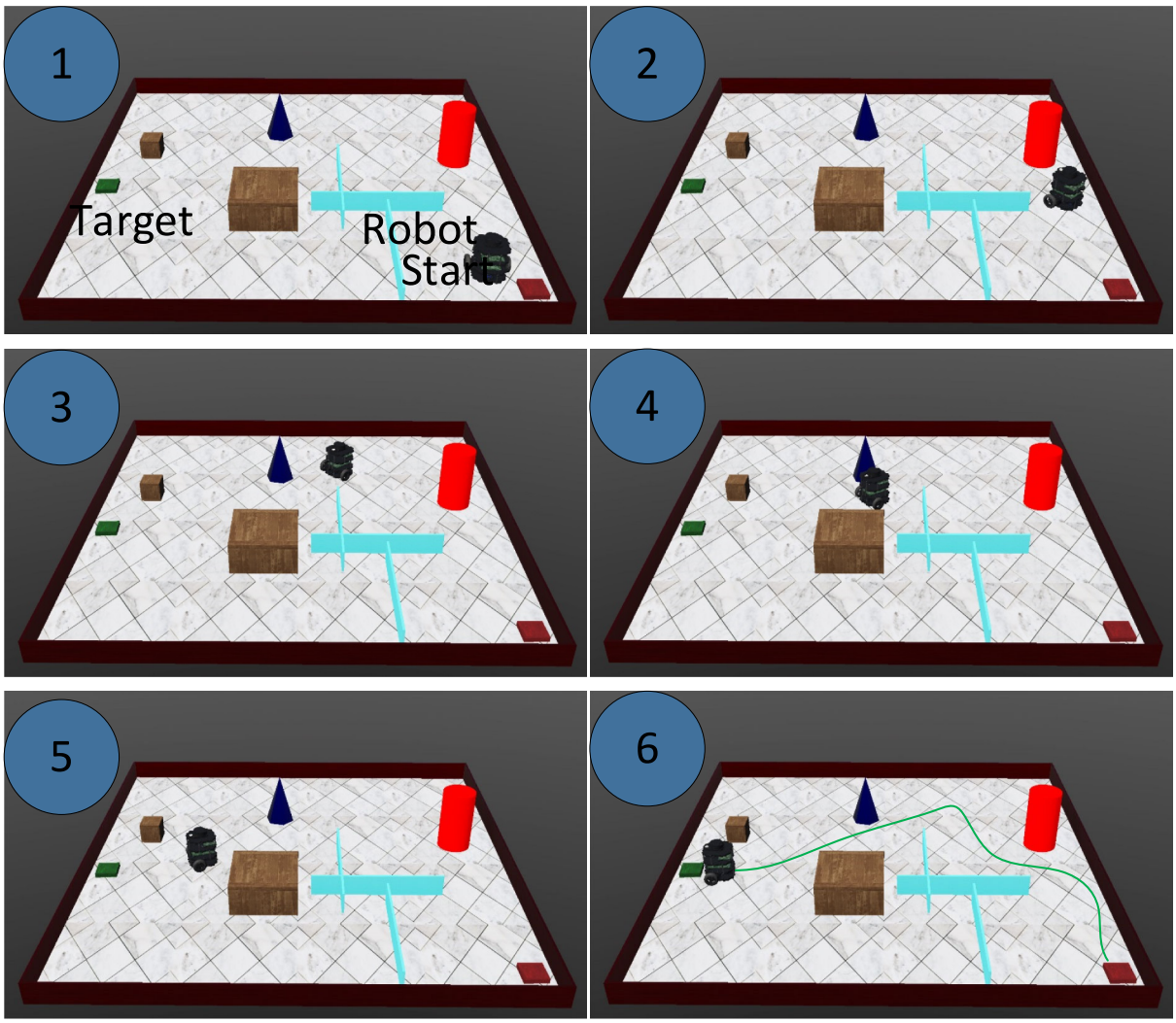

Fig. 10 Implementation of TD3 in environment having more complicated path (static Obstacles)

Table 7 Navigation parameter of path planning using TD3 in static environment

Run	Path length (cm)	Time travel (s)	Average velocity (cm/s)	Collision count	Success rate (%)
1	236	31	7.62	2	100
2	245	33	7.43	1	100
3	257	38	6.77	3	78
4	230	29	7.94	0	100
5	280	42	6.67	3	85
Avg	249.6	34.6	7.286	1.8	92.6

smoothing, and delayed policy updates. TD3 utilizes two Critic networks (Q-networks) to deal with the overestimation bias usually found in value estimates. Thus, reducing the bias effectively, the algorithm computes the minimum of the two Q-values. Through the 'Critic' class, this is put into action whereby there are two separate networks ('Q1' and 'Q2') that approximate the Q-values. Make these values using 'forward' and target actor update is done by 'Q1_forward' method. Target policy smoothing adds noise to the target action so that it remains close to the Q value without exploiting sharp peaks in the value function. We achieve this by adding noise sampled from a Gaussian distribution to the target actions that are obtained by clipping them between the range '*noise_clip*' during training. In the 'train' method, this idea is applied by appending noise to the target actions given by the '*actor_target*' network. Longer intervals between the updates of the policy network, in contrast with those of the

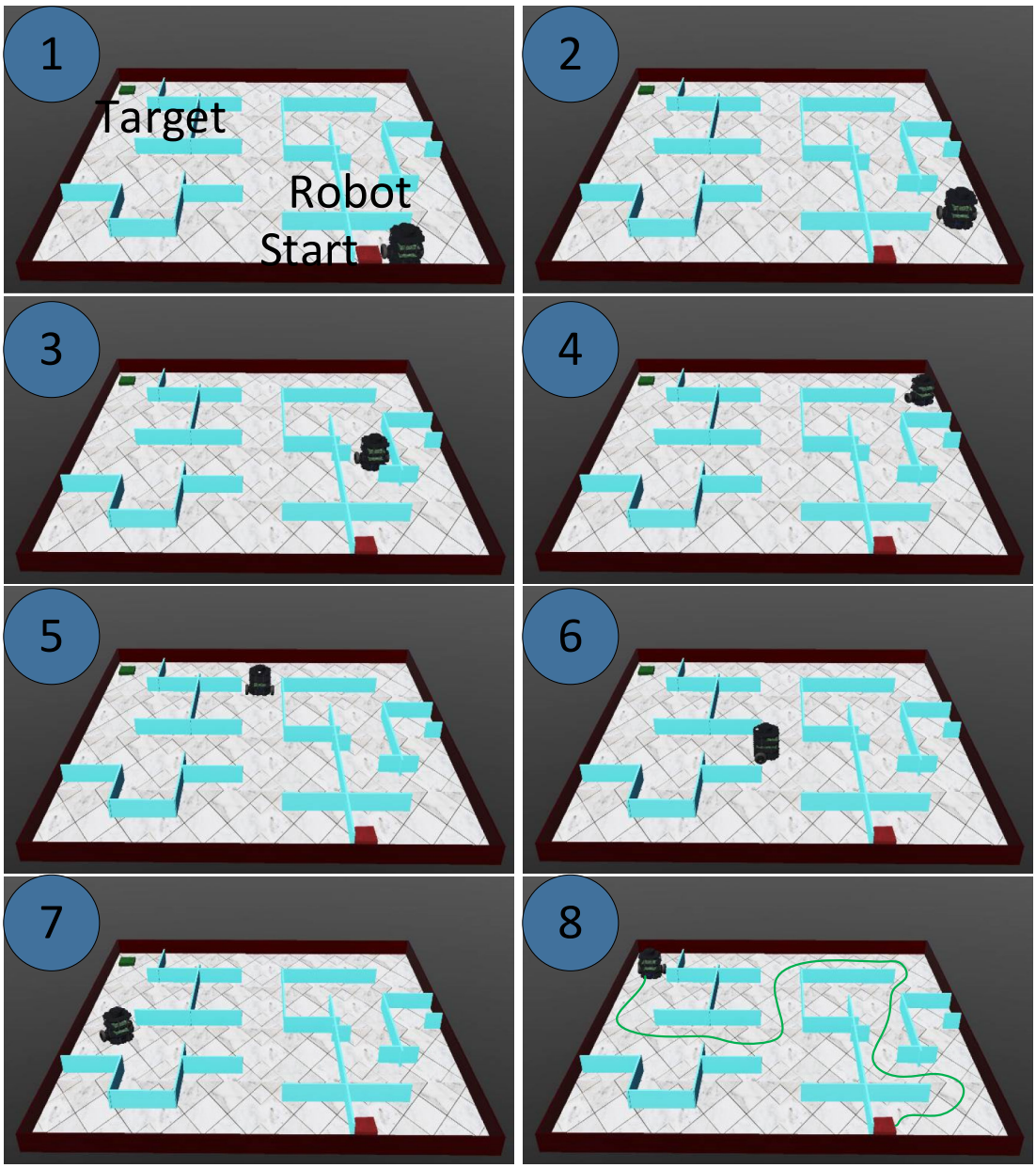

Fig. 11 Implementation of TD3 in the maze environment

Table 8 Navigation parameter of path planning using TD3 in maze environment

Run	Path length (cm)	Time travel (s)	Run into dead ends	Success rate (%)	Reward/Run
1	315	47	2	100	0.87
2	326	53	3	75	0.77
3	295	45	2	100	0.9
4	322	51	3	100	0.82
5	340	49	4	80	0.77
Avg	319.6	49	3	91	0.82

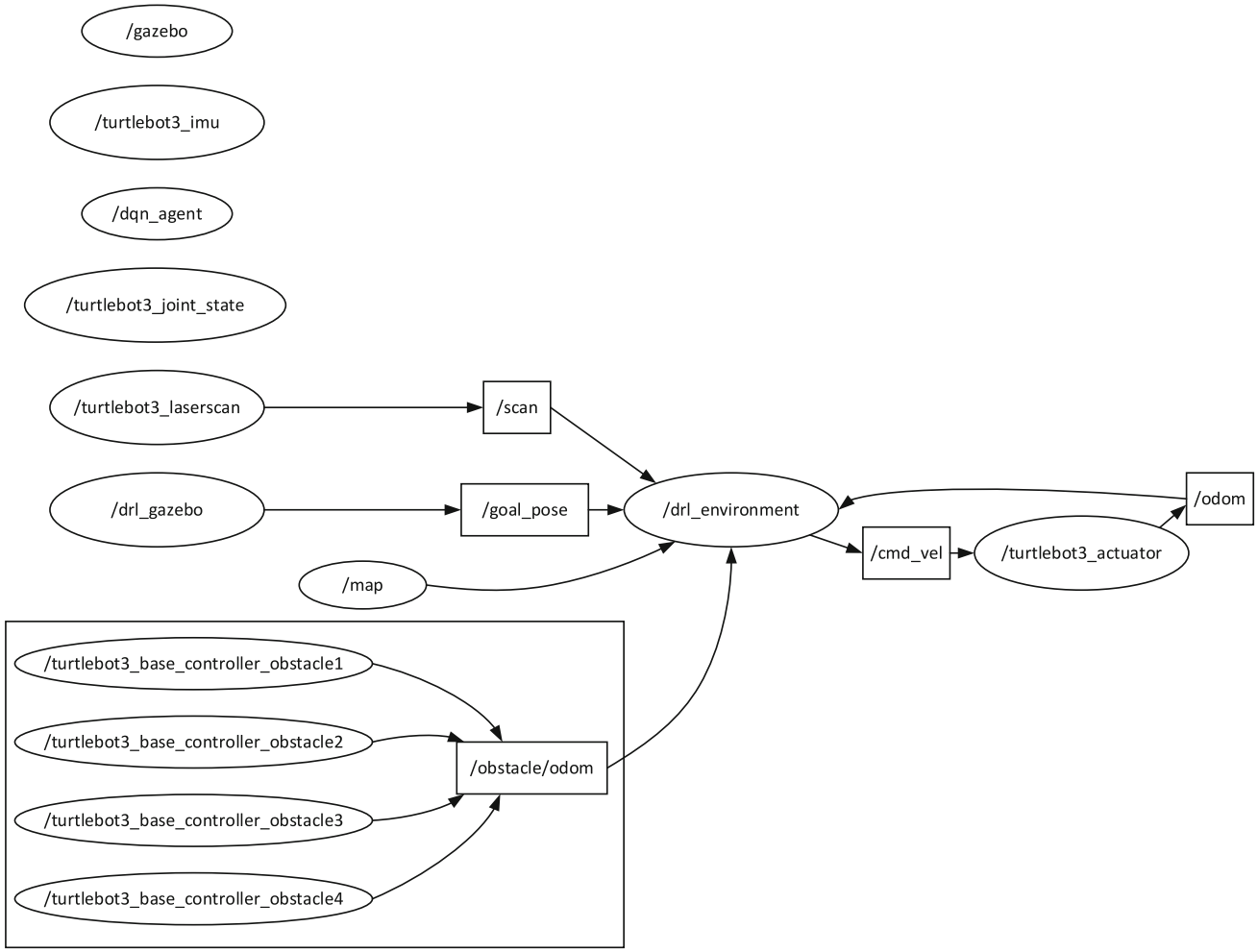

Fig. 12 Rqt_graph for implementation of the DRL algorithm for Dynamic Environment

Q-networks, are a common practice in TD3. Usually, the policy is updated for only half the frequency of updating Q-networks.

5.3.3.1 Algorithm 3: TD3 implementation of motion planning

1. Input
- Obtain data of LiDAR *(/scan)* and odometry *(/odom)*
- Provide stating goal location *(/goal_pose)*
2. Initialization
- Establish networks of actors and critics with arbitrary weights.
- Configure the reward system according to velocity, collision prevention, and objective range.
3. Training Loop

A. The 'train' method shows how this postponed update mechanism is in play using the check that the current iteration is a multiple of '*policy_freq*'. If that is the case, the actor can be altered to improve actions that have the potential of achieving low negative Q-values, as expected. In the context of TurtleBot3, TD3 can be employed to train the robot to navigate autonomously while avoiding obstacles and reaching a goal.

B. The 'TD3' class initializes the actor and critic networks, noise for exploration ('OUNoise'), and optimizers. The '*get_action*' method computes the action given the current state, incorporating exploration noise during training.

C. The 'train' method updates the networks based on the agent's experience, leveraging the TD3 enhancements to improve stability and performance.

D. Parameters in TD3 include the '*policy_noise*', '*noise_clip*', and '*policy_freq*', which control the amount of noise added to target actions,

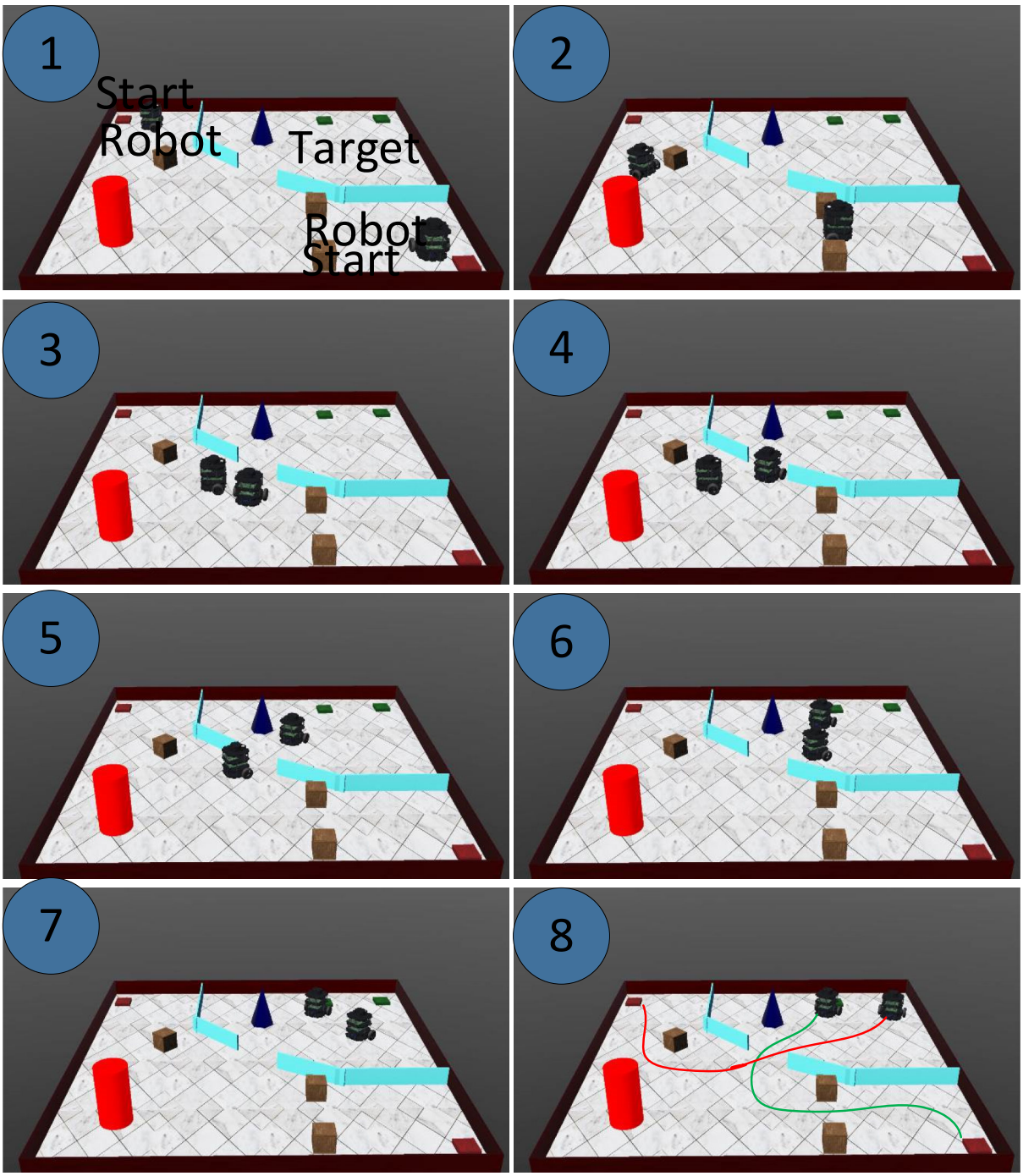

Fig. 13 Navigation of multiple TurtleBot3 robot using TD3 algorithm

the range of this noise, and the frequency of policy updates, respectively portrayed in Tables 4 and 5.

E. The '*discount_factor*' determines the importance of future rewards, and 'tau' controls the rate of soft updates to the target networks. Mathematically, the Q-value updates, presented in Eq. (10), are computed as in:

$$Q_{target} = r + \gamma(1 - d)\min(Q1_{target}(s', a'), Q2_{target}(s', a')) \tag{10}$$

where r is the reward, γ is the discount factor, d is the done signal, s' and a' are the next state and action, and Q1target and Q2target are the target Q-networks.

F. The actor update aims to maximize the expected Q-value by minimizing the loss (presented in Eq. (11)):

Table 9 Navigation parameter of path planning using TD3 in dynamic environment (two Turtlebot3)

Run	Robot 1 path length (cm)	Robot 1 time travel (s)	Robot 2 path length (cm)	Robot 2 time travel (s)	Inter collision count	Success rate (%)
1	280	40	272	34	1	100
2	268	38	262	32	0	100
3	290	45	270	35	2	80
4	275	44	266	36	2	80
5	278	42	270	33	1	100
Avg	278.2	41.8	268	34	1.2	92

$$L_{\text{actor}} = -\sum_{(i=1)}^{N} Q_1(s_i, \pi(s_i)) \tag{11}$$

G. Modify the target networks by utilizing a soft modification (presented in Eq. (12)

$$\theta' = \tau\theta + (1-\tau)\theta' \tag{12}$$

4. Output

TD3 is particularly useful in continuous control tasks where precise and stable policy updates are crucial. Its application to TurtleBot3 demonstrates its capability to train complex robotic behaviors in environments with continuous action spaces, leading to improved navigation and obstacle avoidance skills.

The training process involves updating the DRL agent's policy based on the rewards received from its interactions with the environment. The agent learns to improve its decision-making by iteratively refining its action-value estimates. The trained DRL agent selects actions based on its current policy. These actions are intended to optimize the robot's behaviour for goal tracking and obstacle avoidance. The selected actions are converted into velocity commands (*/cmd_vel*) that control the robot's movement. The environment node sends these commands to the robot, directing it to move towards the goal while avoiding obstacles.

6 Result and discussion

The performance of SLAM using a teleoperation node and Nav2 with the TurtleBot3 in static environment is found to be successful, as shown by the outcomes. This setup included information on what ought to be done as well as why it should be done as well as hardware and software components required for this implementation. As TurtleBot3 is remotely controlled to move around, the SLAM algorithm produces a live map that keeps changing over time to cover the objects and shapes within that area. The more advanced version of Nav2, on the other hand, offers automatic movement paths in relation to pre-fixed static obstructions in the environments. Researchers in robotics could deduce vital information from a successful adoption of both SLAM and Nav2. Figure 9 depicts the TurtleBot3 navigation and path planning in randomly placed static obstacle in the environment. The robot is guided towards the target by applying DDPG, TD3 and DQN algorithm in same environment. The SLAM is used to make the map and then the Nav2 along with the individual algorithm is fed to the robot one by one. DDPG, DQN and TD3 have successfully guided the robot to the target in 89, 75 and 60 iterations, respectively.

DQN used a discrete action space approach. This discretized the action space, in which an action is selected from a finite set into predefined linear and angular velocities. TurtleBot3 illustrated its ability to steer clear of objects and follow goals at the same time effectively. Yet, the robot moved in an un-smooth manner, thus making abrupt switches when moving between actions due to its discrete nature as far as DQN's homogeneity was concerned. Then, implementation of TD3 has been done, which is actually an enhancement of DDPG. The primary goal of TD3 was minimizing overestimation errors while improving policy robustness through two critics system merge and target policy smoothing. These outcomes looked quite optimistic—it seems that the TurtleBot3 showed better performance in terms of navigation and target tracking than in other environments. The algorithm was impressive due to the fact that it did not depend much on the changes in hyperparameters and could adapt well to tough and changing situations. Effective exploration during training was made possible by incorporating noise to the

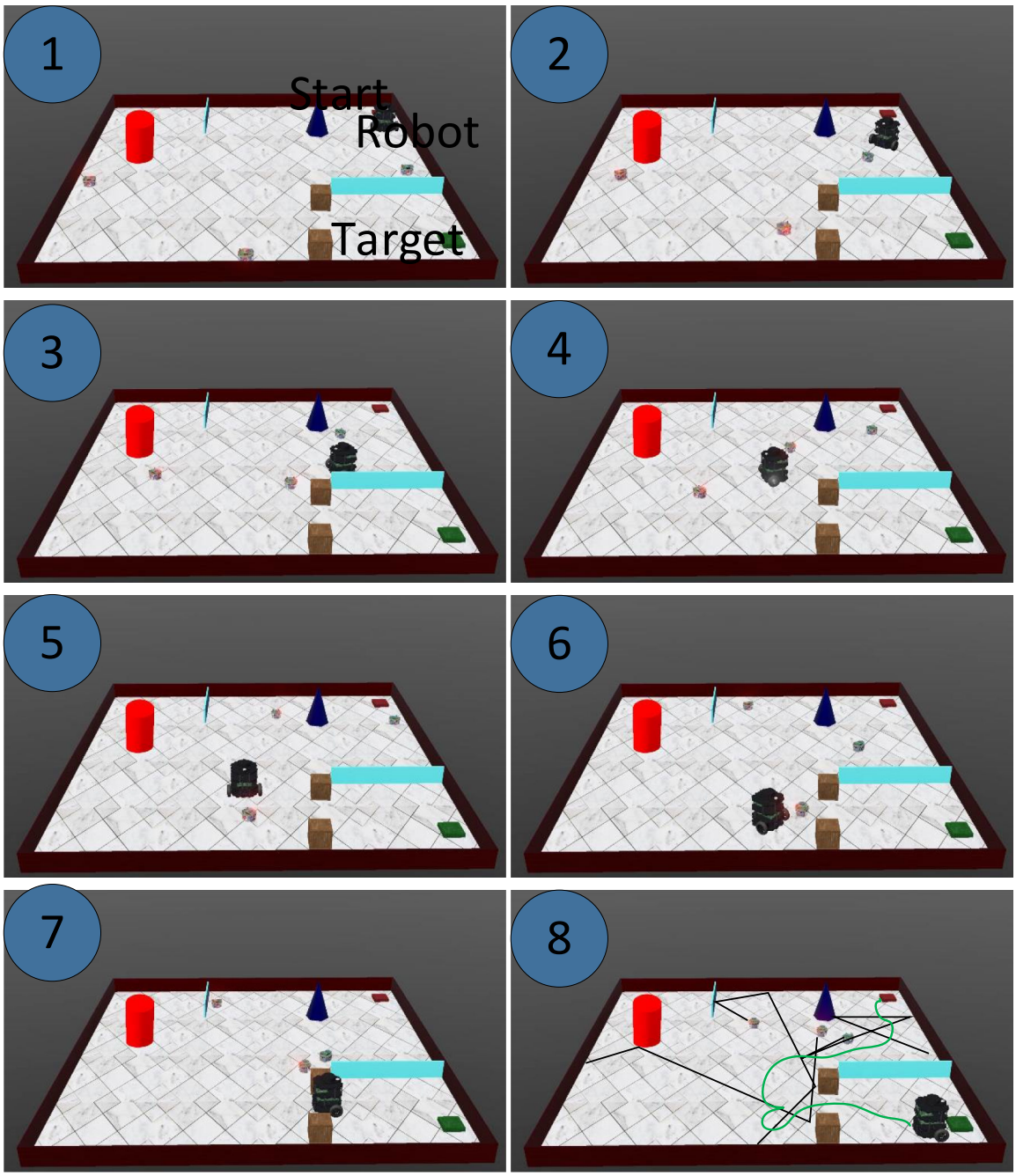

Fig.14 Implementation of the TD3 algorithm for Dynamic Environment having 3 E Puck robots

Table 10 Navigation parameter of path planning using TD3 in dynamic environment (three E Puck)

Run	Path Length (cm)	Time travel (s)	Collision Count	Reroute count	Success Rate (%)
1	278	47	4	5	80
2	285	52	6	6	70
3	265	45	3	4	80
4	253	35	1	2	100
5	259	42	2	1	100
Avg	268	44.2	3.2	3.6	86

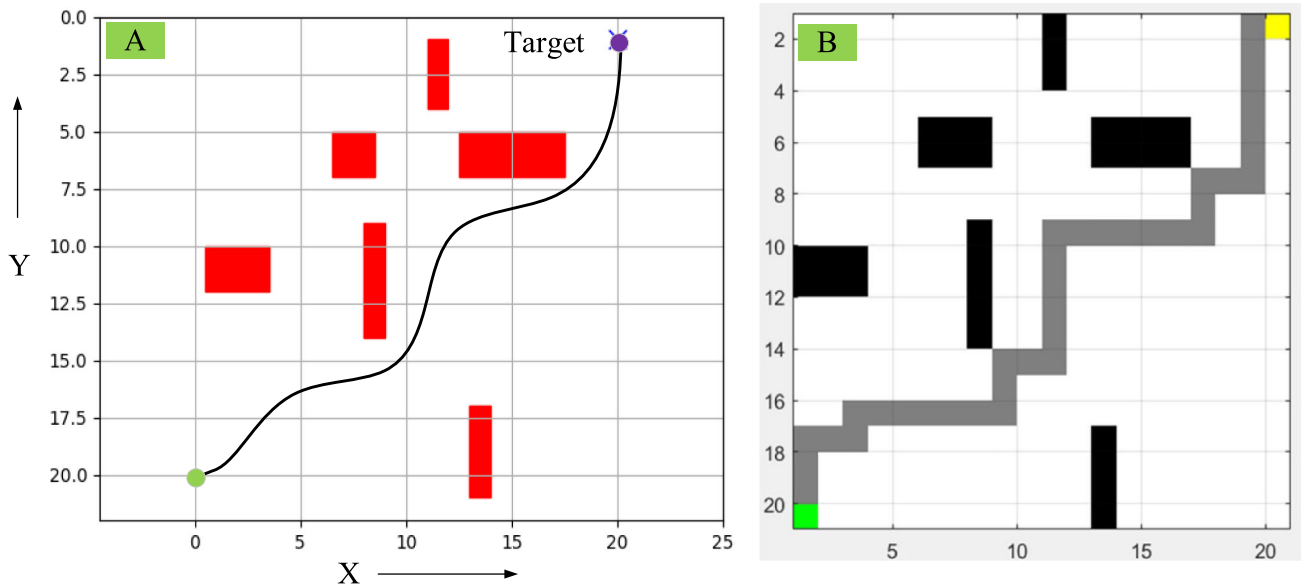

Fig. 15 Comparison of the (A) TD3 algorithm with respect to an (B) existing technique [29]

Table 11 Representation of comparison of TD3 with existing technique [29] with respect to path length (unit)

Dijkstra algorithm	A* algorithm	Hybrid algorithm	TD3	Improvement of TD3 w.r.t other algorithms (%)		
				Dijkstra	A*	Hybrid
40	35	32	29	27.5	17.14	9.37

policy, hence improving learning efficiency. In quantitative terms, the following metrics and presented in Table 6. These metrics point out the comparative advantages of TD3 when compared to DDPG and DQN. In terms of average navigation time, TD3 spends less time, which is evident in the reduced number of collisions per episode taking place within the process of moving around. In this context, TD3's ability to be successful in changing circumstances is also remarkable, as is its durability when put into actual practice. It is established that reinforcement learning methods TD3 improve the autonomous navigation capability of TurtleBot3 by increasing policy stability and facilitating effective discovery. TD3 emerges as an attractive policy for real-time robotic operations. These metrics point out the comparative advantages of TD3 when compared to DDPG and DQN. In terms of average navigation time, TD3 spends less time, which is evident in the reduced number of collisions per episode taking place within the process of moving around. In this context, TD3's ability to be successful in changing circumstances is also remarkable, as is its durability when put into actual practice.

Furthermore, the TD3 is applied in various environment to evaluate the robustness of the proposed algorithm. By using the Robot Operating System 2 (ROS2) and displaying the simulations in the Webots applications, the (TD3) method has been applied in a variety of situations. Utilizing ROS2's real-time connectivity and robotic system management skills enables an effective foundation to assess TD3's effectiveness in various robotic scenarios. The capacity of TD3 to optimize pathways while maintaining stability will be tested as it navigates around a number of fixed obstacles in the complex static environment as shown in Fig. 10. By facilitating smooth interaction between the various parts of the robotic system, ROS2 makes it possible to transfer data and issue control commands effectively. This configuration highlights TD3's capacity to manage challenging navigation tasks in a regulated environment. The quantitative analysis of the algorithm, which shows path length, time travel, average reward per episode, and convergence time, is presented in Table 7.

Further, the proposed algorithm is implemented for maze solving, as shown in Fig. 11. The agent uses a reward mechanism to learn how to navigate a maze in the subsequent environment. To show how well TD3 learns from limited incentives, the objective is to complete the maze in 38–40 iterations. As it adjusts to various maze settings, ROS2's middleware facilitates dynamic modifications to

the agent's state and behaviors, improving the learning procedure. The quantitative analysis of the algorithm, which shows path length, time travel, average reward per episode, and convergence time, is presented in Table 8.

The proposed algorithm is then validated in two different types of dynamic environment. In one environment, two Turtlebot3 is present and another has three E-Puck [28] robots. The state in environments with dynamic obstacles is defined by laser scans alongside position and velocity coordinates. Linear and angular velocities are used to specify what the robot can do in such circumstances. Then, implementation of TD3 has been done, which is actually an enhancement of DDPG. Accordingly, a continuous control policy is developed for the path of TurtleBot3 through the application of the TD3 algorithm. TD3, being an actor-critic method, leverages a deterministic policy with an off-policy learning strategy. It has been noticed that TD3 shows a good performance level in environments that are easy by keeping the control of the robot's movement smooth and continuous. TurtleBot3 moved successfully without any challenges past hurdles and followed goals without any waver.

Figure 12 represents the *rqt_graph* for the implementation of the */drl_environment* node to run the reinforcement algorithms for dynamic environment. As discussed before, the /scan and */odom* output generated from the lidar of TurtleBot3 are fed to the */drl_environment* node along with */goal_pose*. */Goal_pose* is a node that implements the goal in the environment. This will be updated randomly when the TurtleBot3 reaches the goal. The output generated from */drl_environment* node is fed to */cmd_vel,* which moves the TurtleBot3. The*/odom* of the new pose is fed back to the */drl_environment* node, which helps the localization of the TurtleBot3 robot in the environment.

The primary goal of TD3 is minimizing overestimation errors while improving policy robustness through two critics system merge and target policy smoothing. These outcomes looked quite optimistic—it seems that the TurtleBot3 showed better performance in terms of navigation and target tracking than in other environments. The algorithm was impressive due to the fact that it did not depend much on the changes in hyperparameters and could adapt well to tough and changing situations. Effective exploration during training was made possible by incorporating noise into the policy, hence improving learning efficiency.

In the third environment, two TurtleBot3 robots work together in a single area as shown in Fig. 13. In Fig. 13 (3), robots get into conflict as both robots come into each other's sensory range. The first robot cannot see the target; therefore, the first robot comes back (as shown in Fig. 13 (4)) to lead others along the best course. Once the second robot go out of the threshold range of the first robot it starts moving and both reach the target. This situation puts TD3's ability to coordinate and work well with others to the test. The directing robot may effectively share its learned path with other robots thanks to its connection with ROS2, which enhances job efficiency overall. The quantitative analysis for the navigation is presented in Table 9. Finally, a TurtleBot3 must maneuver toward a goal while dodging dynamic obstacles in the fourth scenario, which is an unpredictable environment with 3 E-Puck robots (as shown in Fig. 14). In this situation, all three E Puck robots are left in the environment to move freely. It came in the path of TurtleBot3, due to which Turtlebot3 have to change its path and reach the target. This configuration tests TD3's ability to make decisions in real-time and avoid obstacles under changing circumstances. By offering real-time sensory information processing and command commands—both essential for negotiating unforeseen obstacles ROS2 improves this environment. The quantitative analysis for the navigation is presented in Table 10.

Stable path lengths and suitable travel time quantities are maintained by the algorithm for the static barrier setting, indicating that it effectively avoids needless detours. Strong obstacle recognition and evasion are shown by a low collision count throughout runs, especially in static conditions. Strong effectiveness in predictable circumstances is indicated by the high success rate (almost 100%), which confirms that the algorithm consistently achieves the target without significant disruptions. In spite of the maze's difficulty, the algorithm successfully cuts down on the amount of time needed to go there and comes across a few dead ends, exhibiting effective path planning and decision-making in complicated settings. The algorithm's ability to avoid becoming stuck or experiencing excessive delays in complex areas is demonstrated by its excellent success rate of around 82%, which shows that it manages the maze well without becoming lost. The length of the path to the target and "Run into dear ends" indicators, as well as the "Reward/Run" score, demonstrate that the method is successful in identifying effective routes through the maze while eliminating needless dead ends in a setting with numerous TurtleBot3s. Better reward scores, which indicate more ideal efficiency, are obtained by runs with smaller path lengths and fewer dead ends. High replan counts demonstrate the method's dynamic flexibility in the midst of dynamic barriers since it may modify routes in reaction to shifting impediments. The algorithm's real-time flexibility and obstacle avoidance effectiveness are demonstrated by the travel time metrics remaining appropriate in the face of dynamic barriers and the low collision counts. The outcomes in every setting show how reliable, flexible, and effective the navigation method is. Effective path planning in expected conditions is demonstrated by the algorithm's consistent avoidance of collisions and timely arrival at the destination in the static barrier setting.

High incentive scores in the labyrinth environment indicate consistent travel time data, few dead-end experiences, and ideal path efficiency, confirming the algorithm's efficacy in intricate layouts. The program demonstrates its capacity for multi-agent contacts in the multi-robot situation by achieving coordinated, seamless movement with minimal inter-bot collision counts. Lastly, the algorithm's real-time flexibility is demonstrated in the dynamic obstacle environment, where it adjusts to moving barriers with low collision and replan counts. All of these findings support the algorithm's applicability to a range of complicated real-world navigation problems.

6.1 Comparison with existing technique

The hybrid approach introduced by Mohanraj et al. [29] that integrates the Dijkstra and A* algorithms for planning paths, is contrasted with the methods suggested in this paper. The authors' work used simulations in a pre-established world where the robot used this hybrid method to navigate to approach a target. A comparable setting has been established, and the Twin Delayed Deep Deterministic Policy Gradient (TD3) has been utilized for route selection in order to duplicate and expand their work. The results are shown in Fig. 15, where 15(A) displays the outcome utilizing the suggested method, in which the robot successfully approaches the target, and 15(B) displays the findings from the paper by Mohanraj et al. [29]. An evaluation of performance with respect to path length is also given in Table 11, which shows improvements of 10% over the Dijkstra method, 12% over the A* approach, and 10% over their hybrid technique. These outcomes show how successful TD3 is in this situation and provide significant benefits over conventional path planning techniques.

7 Conclusion

The consistent evolution in the field of robotics, particularly object tracking and navigation, is important for self-governed moving objects. The objective of this project is to create an advanced object tracking system and navigation for TurtleBot3. It is meant to improve real-time object detection and tracking using ROS two and sophisticated machine-learning methods. The SLAM and Nav2 libraries enable the robotic system to move on its own in static environments with the application reinforcement learning algorithms such as TD3, DDPG and DQN. From the comparison, it is observed that TD3 works better and it is 14% and 27% better than DDPG, respectively in reference to time travel. TD3 is 28.57% and 55% better than DDPG and DQN, respectively with in reference to collision/Episode. Further, it is compared with respect to success rate and average reward. Further, TD3 is implemented in different static and dynamic environments to evaluate the robustness. It is implemented to solve mazes and dynamic environments. It is also compared with existing techniques with respect to path length. The proposed approach shows more than 9% improvement in path length. The lidar sensor of TurtleBot3, which they rely on, then comes into play by giving them data so that they can calculate both rewards, which include collision avoidance parameters when about to crash into something but also penalizing penalties that involve distances traveled wrong side up when trying not only avoid hitting objects head-on (downwards) but also maintain safe distances from them in order not stand still halfway through a given route. The project's evaluation criteria are navigation time to reach the target, the mean number of collisions, the rate of success in dynamic environments and the average reward per episode. This development will significantly improve surveillance, inspection and human–robot interaction applications by enabling TurtleBot3 to navigate complex environments more accurately and efficiently, thus improving performance and capabilities in different scenarios. In the future, advancements in the controller can be made by tuning its parameters using a nature-based algorithm. The advanced algorithm can be implemented in different robots for path planning. In addition, an interaction between turtlebots and drones can be developed to solve different social problems.

Funding Open access funding provided by Manipal Academy of Higher Education, Manipal.

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

References

1. Kumar A, Saini T, Pandey PB, Agarwal A, Agrawal A, Agarwal B (2022) Vision-based outdoor navigation of self-driving car

using lane detection. Int J Inf Technol 14(1):215–227. https://doi.org/10.1007/s41870-021-00747-2
2. Qaisar MW, Shakeel MM, Kędzia K, Machado JM, Jan AZ (2024) Localization-based waiter robot for dynamic environment using Internet of Things. Int J Inf Technol. https://doi.org/10.1007/s41870-023-01723-8
3. Thomas A, Antony JK, Isaac AV, Aromal MS, Verghese S (2024) A novel road attribute detection system for autonomous vehicles using sensor fusion. Int J Inf Technol. https://doi.org/10.1007/s41870-024-02255-5
4. Khan H, Khatoon S, Gaur P, Khan SA (2022) Speed control comparison of wheeled mobile robot by ANFIS, Fuzzy and PID controllers. Int J Inf Technol. https://doi.org/10.1007/s41870-022-00862-8
5. Khan H, Khatoon S, Gaur P (2024) Stabilization of wheeled mobile robot by social spider algorithm based PID controller. Int J Inf Technol 16(3):1437–1447. https://doi.org/10.1007/s41870-023-01438-w
6. Gupta A, Adhikari R, Pandey A, Kashyap AK (2021) Orientation angle based online motion control of an Aldebaran NAO humanoid robot in V-REP software environment using novel sunflower optimization (SFO) algorithm. Int J Inf Technol. https://doi.org/10.1007/s41870-021-00796-7
7. Lee MFR, Yusuf SH (2022) Mobile robot navigation using deep reinforcement learning. Processes. https://doi.org/10.3390/pr10122748
8. Algabri R, Choi MT (2020) Deep-learning-based indoor human following of mobile robot using color feature. Sensors (Switzerland). https://doi.org/10.3390/s20092699
9. Kamble A, Mitra AC, Tathe A, Kumbharkar S, Bhandare A (2023) Object following robot based on AI/ML. Mater Today Proc 72:1817–1824. https://doi.org/10.1016/j.matpr.2022.09.577
10. Park S, Shi Z, Zhong Y (2023) Feature-based SLAM and moving object detection and tracking with ego-motion compensation. Chin Control Conf (CCC). https://doi.org/10.23919/CCC58697.2023.10240465
11. Eslami N, Amiadifard R (2019) Moving target tracking and obstacle avoidance for a mobile robot using MPC. ICEE Iran Conf Electr Eng. https://doi.org/10.1109/IranianCEE.2019.8786418
12. Zhang Y, Xia Q, Xie P (2021) Research and implementation of path planning for mobile robot in unknown dynamic environment. IEEE Int Conf Artific Intell Comput Appl (ICAICA). https://doi.org/10.1109/ICAICA52286.2021.9498260
13. Nandkumar C, Shukla P, Varma V (2021) Simulation of indoor localization and navigation of Turtlebot 3 using real time object detection. Proc IEEE Int Conf Disrupt Technol Multi-Discip Res Appl. https://doi.org/10.1109/CENTCON52345.2021.9687937
14. Gulalkari AV, Sheng D, Pratama PS, Kim HK, Byun GS, Kim SB (2015) Kinect camera sensor-based object tracking and following of four wheel independent steering automatic guided vehicle using Kalman filter. Int Conf Control Autom Syst Proc. https://doi.org/10.1109/ICCAS.2015.7364622
15. Molina Santiago JF, Fragoso-Mandujano JA, Gomez-Penate S, Gonzalez VDC, Lopez-Estrada FR (2023) Trajectory tracking and obstacle avoidance with Turtlebot 3 Burger and ROS 2. Proc Robot Mex Congr. https://doi.org/10.1109/COMRob60035.2023.10349744
16. Liu L, Wang X, Yang X, Liu H, Li J, Wang P (2023) Path planning techniques for mobile robots: review and prospect. Expert Syst Appl 227:120254. https://doi.org/10.1016/j.eswa.2023.120254
17. Kavraki LE, Švestka P, Latombe JC, Overmars MH (1996) Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans Robot Autom. https://doi.org/10.1109/70.508439
18. Zhang X, Zhao Y, Deng N, Guo K (2016) Dynamic path planning algorithm for a mobile robot based on visible space and an improved genetic algorithm. Int J Adv Robot Syst 13(3):91. https://doi.org/10.5772/63484
19. Zeng T, Si B (2019) Mobile robot exploration based on rapidly-exploring random trees and dynamic window approach. Int Conf Control Autom Robot (ICCAR). https://doi.org/10.1109/ICCAR.2019.8813489
20. Dewang HS, Mohanty PK, Kundu S (2018) A robust path planning for mobile robot using smart particle swarm optimization. Proced Comput Sci 133(1):290–297. https://doi.org/10.1016/j.procs.2018.07.036
21. Teimoori H, Savkin AV (2010) A biologically inspired method for robot navigation in a cluttered environment. Robotica 28(5):637–648. https://doi.org/10.1017/S0263574709990294
22. Khooban MH, Alfi A, Abadi DNM (2013) Teaching-learning-based optimal interval type-2 fuzzy PID controller design: a nonholonomic wheeled mobile robots. Robotica 31(7):1059–1071. https://doi.org/10.1017/S0263574713000283
23. Ahmed-Abdulsaheb J, Jasim-Kadhim D (2023) Real-time SLAM mobile robot and navigation based on cloud-based implementation. J Robot. https://doi.org/10.1155/2023/9967236
24. Kashyap AK, Parhi DR, Kumar V (2023) Navigation for multi-humanoid using MFO-aided reinforcement learning approach. Robotica. https://doi.org/10.1017/S0263574722001357
25. Jiang L, Huang H, Ding Z (2020) Path planning for intelligent robots based on deep Q-learning with experience replay and heuristic knowledge. IEEE/CAA J Autom Sin 7(4):1179–1189. https://doi.org/10.1109/JAS.2019.1911732
26. Lobos-Tsunekawa K, Leiva F, Ruiz-Del-Solar J (2018) Visual navigation for biped humanoid robots using deep reinforcement learning. IEEE Robot Autom Lett 3(4):3247–3254. https://doi.org/10.1109/LRA.2018.2851148
27. Muktiadji RF, Ramli MAM, Milyani AH (2024) Twin-delayed deep deterministic policy gradient algorithm to control a boost converter in a DC microgrid. Electronics 13(2):433. https://doi.org/10.3390/electronics13020433
28. Nemec D, Janota A, Hruboš M, Gregor M, Pirník R (2017) Mutual acoustic identification in the swarm of e-puck robots. Int J Adv Robot Syst 14(3):172988141771079. https://doi.org/10.1177/1729881417710794
29. Mohanraj T, Dinesh T, Guruchandhramavli B, Sanjai S, Sheshadhri B (2023) Mobile robot path planning and obstacle avoidance using hybrid algorithm. Int J Inf Technol 15(8):4481–4490. https://doi.org/10.1007/s41870-023-01475-5