

Deep-Reinforcement-Learning-Based Autonomous Navigation Robot (ROS 2)

Student Name: Manojkumar Dheenadhayalan

Student ID: st20316714

Date: January 15, 2026

Module: CIS7035 Mobile Robotics

Abstract

This report presents the design, implementation, and critical evaluation of an autonomous mobile robot navigation system utilizing Deep Reinforcement Learning (DRL) within the Robot Operating System 2 (ROS 2) framework. Moving beyond classical path-planning algorithms, this project implements the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm to enable a mobile robot to navigate continuous environments, avoid obstacles via LiDAR perception, and reach designated goals. The system leverages a hybrid architecture combining a Raspberry Pi 4 for high-level inference and a Raspberry Pico H running Micro-ROS for low-level motor control. Furthermore, this report critically analyses the extension of this single-agent architecture into Multi-Robot Systems (MRS), evaluating the technical challenges of non-stationarity and communication overhead, and proposing Centralised Training with Decentralised Execution (CTDE) as a viable solution for swarm robotics.

1. Introduction

Autonomous navigation remains a cornerstone of modern robotics, underpinning applications ranging from warehouse logistics to search and rescue operations. Traditional approaches to navigation, such as the Dynamic Window Approach (DWA) or A* search (Hart et al., 1968), rely heavily on accurate, pre-built maps and deterministic rule sets. While effective in static environments, these classical methods often struggle in dynamic, unstructured settings where the environment changes unpredictably. Consequently, there has been a paradigm shift towards learning-based methods, specifically Deep Reinforcement Learning (DRL), which allows agents to learn optimal navigation policies through trial and error without explicit feature engineering (Sutton and Barto, 2018).

This project aims to develop a robust navigation stack using the TD3 algorithm, a state-of-the-art actor-critic method designed to address the overestimation bias inherent in previous algorithms like Deep Deterministic Policy Gradient (DDPG). The implementation utilizes ROS 2 (Humble Hawksbill), capitalizing on its enhanced real-time capabilities and security over ROS 1. The system is trained in a high-fidelity Gazebo simulation before deployment to a physical robot.

The objectives of this report are threefold: firstly, to detail the hardware and software architecture of the developed robot; secondly, to critically analyse the implementation of the TD3 algorithm and its performance in obstacle avoidance; and thirdly, to evaluate the theoretical and practical implications of scaling this architecture to multi-robot scenarios, addressing the complexities of collaborative swarm intelligence.

2. Literature Review and Theoretical Framework

2.1 The Evolution of Autonomous Navigation

Early navigation relied on Simultaneous Localisation and Mapping (SLAM) combined with deterministic path planners. Algorithms such as Dijkstra's and A* provide mathematically optimal paths but require significant computational overhead to recalculate trajectories in dynamic environments (Candra et al., 2020). As noted by Nasti and Chishti (2024), AI-enhanced strategies are increasingly required to handle the stochastic nature of real-world environments. DRL offers a solution by mapping raw sensor data directly to actuator commands, a process known as end-to-end learning.

2.2 Deep Reinforcement Learning in Robotics

Reinforcement Learning (RL) formulates the navigation problem as a Markov Decision Process (MDP), where an agent interacts with an environment to maximise a cumulative reward. The introduction of Deep Q-Networks (DQN) by Mnih et al. (2015) demonstrated that neural networks could approximate Q-values for high-dimensional state spaces. However, DQN is limited to discrete action spaces, making it unsuitable for the smooth, continuous control required for mobile robots.

To address continuous control, Lillicrap et al. (2019) introduced DDPG, an actor-critic method. Despite its success, DDPG suffers from Q-value overestimation, where the agent systematically overestimates the returns of certain actions, leading to suboptimal policies. Fujimoto et al. (2018) proposed the Twin Delayed DDPG (TD3) to mitigate this. TD3 introduces three critical improvements: clipped double Q-learning (using two critic networks), delayed policy updates, and target policy smoothing. This project selects TD3 due to its superior stability and data efficiency compared to DDPG and Soft Actor-Critic (SAC) in robotic navigation tasks (Xu et al., 2024).

3. System Design and Methodology

3.1 Hardware Architecture

The robotic platform is designed as a differential drive system. The computational architecture is distributed to ensure real-time performance:

- **High-Level Processing:** A Raspberry Pi 4 Model B (4GB) running Ubuntu 22.04 and ROS 2 Humble. This unit handles the DRL inference, LiDAR data processing, and Wi-Fi communication.
- **Low-Level Control:** A Raspberry Pico H microcontroller. This component interfaces directly with the DC motors and encoders. Crucially, it utilizes **Micro-ROS**, an extension of ROS 2 for microcontrollers. This allows the Pico to function as a first-class ROS 2 node, publishing odometry and subscribing to velocity commands directly, bridging the gap between embedded real-time constraints and the ROS 2 DDS (Data Distribution Service) layer.
- **Sensors:** A 360-degree LiDAR for environmental perception and a BNO055 IMU for precise orientation tracking.

3.2 Software Architecture: ROS 2 and Gazebo

The software stack is built upon ROS 2, utilizing its node-based architecture to modularise the system. The communication pipeline is defined as follows:

Topic/Service	Message Type	Function
<code>/scan</code> or <code>/velodyne_points</code>	<code>sensor_msgs/PointCloud2</code>	Transmits LiDAR distance data for obstacle detection.
<code>/odom</code>	<code>nav_msgs/Odometry</code>	Provides robot pose (position and orientation) derived from IMU and encoders.
<code>/cmd_vel</code>	<code>geometry_msgs/Twist</code>	Carries linear and angular velocity commands generated by the TD3 agent.

Simulation Environment: Gazebo is employed to create a physics-based training ground. A custom launch file (`multi_robot_scenario.launch`) initializes the environment with bounding walls and random obstacles. The `GazeboEnv` class (implemented in `env.py`) acts as a bridge, converting ROS messages into a state vector compatible with the PyTorch-based neural networks.

4. Algorithmic Implementation: TD3

The core of the navigation system is the TD3 algorithm, implemented in Python using the PyTorch library. The implementation is divided into three primary components: the Actor, the Critic, and the Training Loop.

4.1 State and Action Space

The state space S is a composite vector consisting of LiDAR readings and goal information. Specifically, the LiDAR data is discretised into 20 sectors (gaps) to reduce dimensionality while retaining directional obstacle information. The robot's relative position to the goal is represented by the Euclidean distance and the heading error (yaw).

The action space A is continuous, consisting of linear velocity (v) and angular velocity (ω). The network outputs values in the range $[-1, 1]$, which are then scaled to the robot's physical limits.

4.2 Network Architecture

The **Actor network** (defined in `train.py`) maps states to actions. It consists of three fully connected layers (Input \rightarrow 800 \rightarrow 600 \rightarrow Output) with ReLU activation functions, culminating in a Tanh activation to bound the output.

The **Critic network** estimates the Q-value $Q(s, a)$. Uniquely to TD3, there are two identical critic networks (Q_1 and Q_2). During training, the target Q-value is calculated as the minimum of the two critics:

$$y = r + \gamma * \min(Q'_1(s', a'), Q'_2(s', a'))$$

This mechanism, known as **Clipped Double Q-Learning**, prevents the overestimation of value estimates that often causes DDPG to fail in complex navigation tasks.

4.3 Reward Function Design

The reward function is the critical driver of learning. Analysis of the `env.py` code reveals a shaped reward structure:

- **Goal Reached:** A large positive reward (+100) is granted when the distance to the target is less than 0.3 meters.
- **Collision:** A large negative penalty (-100) is applied if the robot comes within 0.35 meters of an obstacle.
- **Navigation Reward:** At each step, a small reward is calculated based on the action: $\text{action}[0]/2 - \text{abs}(\text{action}[1])/2$. This encourages the robot to move forward (linear velocity) and penalises excessive rotation, promoting smooth trajectories.

5. Implementation and Performance Evaluation

5.1 Training Process

The training was conducted in the Gazebo simulation. The `train.py` script manages the interaction between the replay buffer and the neural networks. A replay buffer size of 1,000,000 transitions was instantiated to store experiences (s, a, r, s', d) .

Exploration was managed by adding Gaussian noise to the actions selected by the policy. This noise decays over time (`expl_decay_steps = 500,000`), allowing the agent to transition from exploration (random actions) to exploitation (using the learned policy). A specific logic in the training loop (`random_near_obstacle`) forces the robot to take random actions if it is near an obstacle but not colliding; this heuristic prevents the robot from getting stuck in local minima near walls.

5.2 Simulation Results

During the initial epochs (0-50), the robot exhibited erratic behaviour, frequently colliding with obstacles as it populated the replay buffer. As the Critic networks converged, the robot began to associate LiDAR proximity readings with negative rewards. By epoch 200, the agent demonstrated "wall-following" behaviour, capable of navigating around obstacles to reach the goal.

The integration of the **Target Policy Smoothing** (adding noise to the target action during the Critic update) proved essential. In comparative tests with standard DDPG, the TD3 agent showed less oscillatory behaviour in narrow corridors, attributed to the more stable Q-value targets.

5.3 Sim-to-Real Transfer

The transition from Gazebo to the physical Raspberry Pi/Pico platform highlighted the "Reality Gap." While the logic held, sensor noise from the real LiDAR (RPLIDAR) and wheel slippage required adjustments. The modularity of ROS 2 facilitated this; the `test.py` script required minimal modification, only needing to subscribe to the real hardware topics rather than the Gazebo plugins. The use of Micro-ROS on the Pico ensured that motor commands received from the neural network were executed with low latency, maintaining the stability of the control loop.

6. Extension: Multi-Robot Systems (MRS)

While the current implementation successfully demonstrates single-agent navigation, real-world logistics and warehouse scenarios often require the coordination of multiple robots. Extending the current TD3-based architecture to a Multi-Robot System (MRS) or Swarm Robotics scenario introduces significant complexity.

6.1 Technical Challenges in MRS

1. Non-Stationarity: In a single-agent scenario, the environment is static or changes independently of the agent. In MRS, the environment includes other learning agents. From the perspective of Robot A, the behaviour of Robot B is part of the environment. As Robot B learns and changes its policy, the environment becomes non-stationary, violating the Markov assumption essential for standard RL convergence (Gronauer and Diepold, 2022).

2. Scalability and Communication: As the number of robots increases, the joint state-action space grows exponentially. A centralized approach where one "brain" controls all robots becomes computationally infeasible. Conversely, a fully decentralized approach may lead to sub-optimal coordination (e.g., traffic jams in narrow corridors).

3. The "Lazy Agent" Problem: In cooperative tasks with a shared reward, one agent might learn to perform the task while others do nothing, as they still receive the global reward.

6.2 Proposed Approaches and Solutions

To overcome these challenges, the following methodologies are proposed for extending the current system:

A. Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

MADDPG is an extension of DDPG/TD3 that utilizes the framework of **Centralised Training with Decentralised Execution (CTDE)** (Lowe et al., 2017).

- **Concept:** During training (in simulation), the Critic network of each robot has access to the observations and actions of *all* other robots. This allows the Critic to perceive the environment as stationary because it knows the policies of other agents.
- **Execution:** During execution (deployment), only the Actor network is used. The Actor only requires local observations (LiDAR data), meaning the robots do not need high-bandwidth communication to operate in real-time.
- **Pros/Cons:** This solves the non-stationarity problem but requires a simulator where global state information is available. It fits perfectly with the current Gazebo workflow.

B. Parameter Sharing

Given that the robots in a swarm are often homogeneous (identical hardware), we can use **Parameter Sharing**. Instead of training separate networks for each robot, a single TD3 network is trained using experiences collected from all robots simultaneously.

- **Implementation:** The `ReplayBuffer` in the current code would aggregate transitions from Robot 1, Robot 2, etc. The network updates based on this diverse dataset.
- **Insight:** This dramatically speeds up training as experience is gathered in parallel. To allow for individual identity (e.g., different starting positions), a one-hot vector representing the robot ID can be appended to the state vector.

6.3 Open Questions and Insights

Collision Avoidance between Agents: The current reward function penalises collisions with static obstacles. In MRS, robots must avoid each other. An open question is how to distinguish between a static wall and a moving robot using only LiDAR.

Insight: This can be solved by integrating **Reciprocal Velocity Obstacles (RVO)** into the reward function, or by equipping robots with local communication modules (e.g., Zigbee) to broadcast their velocity vectors to neighbours.

Sim-to-Real in Swarms: Transferring a multi-agent policy to reality is riskier; a single failure can cause a cascade of collisions.

Insight: **Curriculum Learning** should be applied. Start training with one robot, then two, then many. Additionally, "Safety Gym" constraints should be added to the loss function (Constrained MDPs) to ensure safety constraints are never violated during the exploration phase.

7. Tech Stack

Hardwares can be Used

- Raspberry Pico H
- Raspberry Pi 4 Model B (4GB)
- BNO055 Sensor Module (IMU)
- LiDAR Sensor
- Motors with attached encoders

Software & Platforms Used

- Jupyter Notebook
- ROS2 (Humble)
- Gazebo
- Ubuntu 22.04 (Linux)
- PyTorch

7. Conclusion

This project successfully designed and implemented a Deep Reinforcement Learning-based navigation system using the TD3 algorithm within a ROS 2 environment. The system demonstrated the capability to learn optimal path-planning strategies from raw LiDAR data, effectively navigating simulated environments and showing promise for real-world deployment via the Raspberry Pi and Pico architecture.

The analysis confirms that TD3 offers significant stability improvements over DDPG for continuous control tasks in robotics. The use of ROS 2 provided a robust middleware that decoupled the learning agent from the hardware abstraction, simplifying the development cycle.

Furthermore, the exploration of Multi-Robot Systems highlights that while single-agent DRL is powerful, scaling to swarms requires a fundamental shift in training methodology. Adopting Centralised Training with Decentralised Execution (CTDE) represents the most viable path forward, balancing the need for stable learning with the constraints of decentralized hardware. Future work will focus on implementing MADDPG within the existing Gazebo framework to validate these multi-agent hypotheses.

References

- Candra, A., Budiman, M.A. and Hartanto, K. (2020) ‘Dijkstra’s and A-Star in Finding the Shortest Path: A Tutorial’, *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*. IEEE, pp. 28–32.
- Fujimoto, S., van Hoof, H. and Meger, D. (2018) ‘Addressing Function Approximation Error in Actor-Critic Methods’, *arXiv preprint arXiv:1802.09477*.
- Gronauer, S. and Diepold, K. (2022) ‘Multi-agent deep reinforcement learning: a survey’, *Artificial Intelligence Review*, 55, pp. 895–943.
- Hart, P.E., Nilsson, N.J. and Raphael, B. (1968) ‘A Formal Basis for the Heuristic Determination of Minimum Cost Paths’, *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp. 100–107.
- Lillicrap, T.P. et al. (2019) ‘Continuous control with deep reinforcement learning’, *arXiv preprint arXiv:1509.02971*.
- Lowe, R. et al. (2017) ‘Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments’, *Advances in Neural Information Processing Systems*, 30.
- Mnih, V. et al. (2015) ‘Human-level control through deep reinforcement learning’, *Nature*, 518(7540), pp. 529–533.
- Nasti, S.M. and Chishti, M.A. (2024) ‘A Review of AI-Enhanced Navigation Strategies for Mobile Robots in Dynamic Environments’, *2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETISIS)*. IEEE, pp. 1239–1244.
- Sutton, R.S. and Barto, A.G. (2018) *Reinforcement Learning: An Introduction*. 2nd edn. Cambridge, MA: MIT Press.
- Xu, T., Meng, Z., Lu, W. and Tong, Z. (2024) ‘End-to-End Autonomous Driving Decision Method Based on Improved TD3 Algorithm in Complex Scenarios’, *Sensors*, 24(15), p. 4962.

APPENDIX

Abbreviations

The following abbreviations are used in this manuscript.

- DRL Deep reinforcement learning
- SLAM Simultaneous localization and mapping
- RRT Rapidly exploring random tree
- MDPs Markov decision processes
- RL Reinforcement learning
- DNNs Deep neural networks
- DQN Deep Q-network
- DDPG Deep deterministic policy gradient
- TD3 Twin delayed deep deterministic policy gradient
- PPO Proximal policy optimization
- SAC Soft actor-critic

- DWA Dynamic window approach
- MARL Multi-agent reinforcement learning
- Double DQN Double deep Q-network
- Dueling DQN Dueling deep Q-network
- TRPO Trust region policy optimization
- A3C Asynchronous advantage actor-critic
- KL divergence Kullback–Leibler divergence
- A* A-Star

CODE

Train.py

```
import os
import time
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from numpy import inf
from torch.utils.tensorboard import SummaryWriter
from replay_buffer import ReplayBuffer
from velodyne_env import GazeboEnv
def evaluate(network, epoch, eval_episodes=10):
    avg_reward = 0.0
    col = 0
    for _ in range(eval_episodes):
        count = 0
        state = env.reset()
        done = False
        while not done and count < 501:
            action = network.get_action(np.array(state))
            a_in = [(action[0] + 1) / 2, action[1]]
            state, reward, done, _ = env.step(a_in)
            avg_reward += reward
            count += 1
            if reward < -90:
                col += 1
        avg_reward /= eval_episodes
        avg_col = col / eval_episodes
        print(".....")
        print(
            "Average Reward over %i Evaluation Episodes, Epoch %i: %f, %f"
            % (eval_episodes, epoch, avg_reward, avg_col)
        )
        print(".....")
    return avg_reward
class Actor(nn.Module):
```

```

def __init__(self, state_dim, action_dim):
    super(Actor, self).__init__()
    self.layer_1 = nn.Linear(state_dim, 800)
    self.layer_2 = nn.Linear(800, 600)
    self.layer_3 = nn.Linear(600, action_dim)
    self.tanh = nn.Tanh()
def forward(self, s):
    s = F.relu(self.layer_1(s))
    s = F.relu(self.layer_2(s))
    a = self.tanh(self.layer_3(s))
    return a
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.layer_1 = nn.Linear(state_dim, 800)
        self.layer_2_s = nn.Linear(800, 600)
        self.layer_2_a = nn.Linear(action_dim, 600)
        self.layer_3 = nn.Linear(600, 1)
        self.layer_4 = nn.Linear(state_dim, 800)
        self.layer_5_s = nn.Linear(800, 600)
        self.layer_5_a = nn.Linear(action_dim, 600)
        self.layer_6 = nn.Linear(600, 1)
    def forward(self, s, a):
        s1 = F.relu(self.layer_1(s))
        self.layer_2_s(s1)
        self.layer_2_a(a)
        s11 = torch.mm(s1, self.layer_2_s.weight.data.t())
        s12 = torch.mm(a, self.layer_2_a.weight.data.t())
        s1 = F.relu(s11 + s12 + self.layer_2_a.bias.data)
        q1 = self.layer_3(s1)
        s2 = F.relu(self.layer_4(s))
        self.layer_5_s(s2)
        self.layer_5_a(a)
        s21 = torch.mm(s2, self.layer_5_s.weight.data.t())
        s22 = torch.mm(a, self.layer_5_a.weight.data.t())
        s2 = F.relu(s21 + s22 + self.layer_5_a.bias.data)
        q2 = self.layer_6(s2)
        return q1, q2
# TD3 network
class TD3(object):
    def __init__(self, state_dim, action_dim, max_action):
        # Initialize the Actor network
        self.actor = Actor(state_dim, action_dim).to(device)
        self.actor_target = Actor(state_dim, action_dim).to(device)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters())
        # Initialize the Critic networks
        self.critic = Critic(state_dim, action_dim).to(device)

```

```

self.critic_target = Critic(state_dim, action_dim).to(device)
self.critic_target.load_state_dict(self.critic.state_dict())
self.critic_optimizer = torch.optim.Adam(self.critic.parameters())
self.max_action = max_action
self.writer = SummaryWriter()
self.iter_count = 0
def get_action(self, state):
    # Function to get the action from the actor
    state = torch.Tensor(state.reshape(1, -1)).to(device)
    return self.actor(state).cpu().data.numpy().flatten()
# training cycle
def train(
    self,
    replay_buffer,
    iterations,
    batch_size=100,
    discount=1,
    tau=0.005,
    policy_noise=0.2, # discount=0.99
    noise_clip=0.5,
    policy_freq=2,
):
    av_Q = 0
    max_Q = -inf
    av_loss = 0
    for it in range(iterations):
        # sample a batch from the replay buffer
        (
            batch_states,
            batch_actions,
            batch_rewards,
            batch_dones,
            batch_next_states,
        ) = replay_buffer.sample_batch(batch_size)
        state = torch.Tensor(batch_states).to(device)
        next_state = torch.Tensor(batch_next_states).to(device)
        action = torch.Tensor(batch_actions).to(device)
        reward = torch.Tensor(batch_rewards).to(device)
        done = torch.Tensor(batch_dones).to(device)
        # Obtain the estimated action from the next state by using the actor-target
        next_action = self.actor_target(next_state)
        # Add noise to the action
        noise = torch.Tensor(batch_actions).data.normal_(0, policy_noise).to(device)
        noise = noise.clamp(-noise_clip, noise_clip)
        next_action = (next_action + noise).clamp(-self.max_action, self.max_action)
        # Calculate the Q values from the critic-target network for the next state-action
pair
        target_Q1, target_Q2 = self.critic_target(next_state, next_action)

```

```

# Select the minimal Q value from the 2 calculated values
target_Q = torch.min(target_Q1, target_Q2)
av_Q += torch.mean(target_Q)
max_Q = max(max_Q, torch.max(target_Q))
# Calculate the final Q value from the target network parameters by using
Bellman equation
target_Q = reward + ((1 - done) * discount * target_Q).detach()
# Get the Q values of the basis networks with the current parameters
current_Q1, current_Q2 = self.critic(state, action)
# Calculate the loss between the current Q value and the target Q value
loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)
# Perform the gradient descent
self.critic_optimizer.zero_grad()
loss.backward()
self.critic_optimizer.step()
if it % policy_freq == 0:
    # Maximize the actor output value by performing gradient descent on negative
    Q values
    # (essentially perform gradient ascent)
    actor_grad, _ = self.critic(state, self.actor(state))
    actor_grad = -actor_grad.mean()
    self.actor_optimizer.zero_grad()
    actor_grad.backward()
    self.actor_optimizer.step()
    # Use soft update to update the actor-target network parameters by
    # infusing small amount of current parameters
    for param, target_param in zip(
        self.actor.parameters(), self.actor_target.parameters()
    ):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )
    # Use soft update to update the critic-target network parameters by infusing
    # small amount of current parameters
    for param, target_param in zip(
        self.critic.parameters(), self.critic_target.parameters()
    ):
        target_param.data.copy_(
            tau * param.data + (1 - tau) * target_param.data
        )
    av_loss += loss
    self.iter_count += 1
    # Write new values for tensorboard
    self.writer.add_scalar("loss", av_loss / iterations, self.iter_count)
    self.writer.add_scalar("Av. Q", av_Q / iterations, self.iter_count)
    self.writer.add_scalar("Max. Q", max_Q, self.iter_count)
def save(self, filename, directory):
    torch.save(self.actor.state_dict(), "%s/%s_actor.pth" % (directory, filename))

```

```

        torch.save(self.critic.state_dict(), "%s/%s_critic.pth" % (directory, filename))
def load(self, filename, directory):
    self.actor.load_state_dict(
        torch.load("%s/%s_actor.pth" % (directory, filename))
    )
    self.critic.load_state_dict(
        torch.load("%s/%s_critic.pth" % (directory, filename))
    )
# Set the parameters for the implementation
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # cuda or cpu
seed = 0 # Random seed number
eval_freq = 5e3 # After how many steps to perform the evaluation
max_ep = 500 # maximum number of steps per episode
eval_ep = 10 # number of episodes for evaluation
max_timesteps = 5e6 # Maximum number of steps to perform
expl_noise = 1 # Initial exploration noise starting value in range [expl_min ... 1]
expl_decay_steps = (
    500000 # Number of steps over which the initial exploration noise will decay over
)
expl_min = 0.1 # Exploration noise after the decay in range [0...expl_noise]
batch_size = 40 # Size of the mini-batch
discount = 0.99999 # Discount factor to calculate the discounted future reward (should
be close to 1)
tau = 0.005 # Soft target update variable (should be close to 0)
policy_noise = 0.2 # Added noise for exploration
noise_clip = 0.5 # Maximum clamping values of the noise
policy_freq = 2 # Frequency of Actor network updates
buffer_size = 1e6 # Maximum size of the buffer
file_name = "TD3_velodyne" # name of the file to store the policy
save_model = True # Weather to save the model or not
load_model = False # Weather to load a stored model
random_near_obstacle = True # To take random actions near obstacles or not
# Create the network storage folders
if not os.path.exists("./results"):
    os.makedirs("./results")
if save_model and not os.path.exists("./pytorch_models"):
    os.makedirs("./pytorch_models")
# Create the training environment
environment_dim = 20
robot_dim = 4
env = GazeboEnv("multi_robot_scenario.launch", environment_dim)
time.sleep(5)
torch.manual_seed(seed)
np.random.seed(seed)
state_dim = environment_dim + robot_dim
action_dim = 2
max_action = 1

```

```

# Create the network
network = TD3(state_dim, action_dim, max_action)
# Create a replay buffer
replay_buffer = ReplayBuffer(buffer_size, seed)
if load_model:
    try:
        network.load(file_name, "./pytorch_models")
    except:
        print(
            "Could not load the stored model parameters, initializing training with random
parameters"
        )
# Create evaluation data store
evaluations = []
timestep = 0
timesteps_since_eval = 0
episode_num = 0
done = True
epoch = 1
count_rand_actions = 0
random_action = []
# Begin the training loop
while timestep < max_timesteps:
    # On termination of episode
    if done:
        if timestep != 0:
            network.train(
                replay_buffer,
                episode_timesteps,
                batch_size,
                discount,
                tau,
                policy_noise,
                noise_clip,
                policy_freq,
            )
        if timesteps_since_eval >= eval_freq:
            print("Validating")
            timesteps_since_eval %= eval_freq
            evaluations.append(
                evaluate(network=network, epoch=epoch, eval_episodes=eval_ep)
            )
            network.save(file_name, directory="./pytorch_models")
            np.save("./results/%s" % (file_name), evaluations)
            epoch += 1
        state = env.reset()
        done = False
        episode_reward = 0

```

```

    episode_timesteps = 0
    episode_num += 1
# add some exploration noise
if expl_noise > expl_min:
    expl_noise = expl_noise - ((1 - expl_min) / expl_decay_steps)
action = network.get_action(np.array(state))
action = (action + np.random.normal(0, expl_noise, size=action_dim)).clip(
    -max_action, max_action
)
# If the robot is facing an obstacle, randomly force it to take a consistent random
action.
# This is done to increase exploration in situations near obstacles.
# Training can also be performed without it
if random_near_obstacle:
    if (
        np.random.uniform(0, 1) > 0.85
        and min(state[4:-8]) < 0.6
        and count_rand_actions < 1
    ):
        count_rand_actions = np.random.randint(8, 15)
        random_action = np.random.uniform(-1, 1, 2)
    if count_rand_actions > 0:
        count_rand_actions -= 1
        action = random_action
        action[0] = -1
# Update action to fall in range [0,1] for linear velocity and [-1,1] for angular velocity
a_in = [(action[0] + 1) / 2, action[1]]
next_state, reward, done, target = env.step(a_in)
done_bool = 0 if episode_timesteps + 1 == max_ep else int(done)
done = 1 if episode_timesteps + 1 == max_ep else int(done)
episode_reward += reward
# Save the tuple in replay buffer
replay_buffer.add(state, action, reward, done_bool, next_state)
# Update the counters
state = next_state
episode_timesteps += 1
timestep += 1
timesteps_since_eval += 1
# After the training is done, evaluate the network and save it
evaluations.append(evaluate(network=network, epoch=epoch, eval_episodes=eval_ep))
if save_model:
    network.save("%s" % file_name, directory="./models")
np.save("./results/%s" % file_name, evaluations)

```

TEST.py

```
import time
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from velodyne_env import GazeboEnv
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Actor, self).__init__()
        self.layer_1 = nn.Linear(state_dim, 800)
        self.layer_2 = nn.Linear(800, 600)
        self.layer_3 = nn.Linear(600, action_dim)
        self.tanh = nn.Tanh()
    def forward(self, s):
        s = F.relu(self.layer_1(s))
        s = F.relu(self.layer_2(s))
        a = self.tanh(self.layer_3(s))
        return a
# TD3 network
class TD3(object):
    def __init__(self, state_dim, action_dim):
        # Initialize the Actor network
        self.actor = Actor(state_dim, action_dim).to(device)
    def get_action(self, state):
        # Function to get the action from the actor
        state = torch.Tensor(state.reshape(1, -1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()
    def load(self, filename, directory):
        # Function to load network parameters
        self.actor.load_state_dict(
            torch.load("%s/%s_actor.pth" % (directory, filename))
        )
# Set the parameters for the implementation
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # cuda or cpu
seed = 0 # Random seed number
max_ep = 500 # maximum number of steps per episode
file_name = "TD3_velodyne" # name of the file to load the policy from
# Create the testing environment
environment_dim = 20
robot_dim = 4
env = GazeboEnv("multi_robot_scenario.launch", environment_dim)
time.sleep(5)
torch.manual_seed(seed)
np.random.seed(seed)
state_dim = environment_dim + robot_dim
action_dim = 2
```

```

# Create the network
network = TD3(state_dim, action_dim)
try:
    network.load(file_name, "./pytorch_models")
except:
    raise ValueError("Could not load the stored model parameters")
done = False
episode_timesteps = 0
state = env.reset()
# Begin the testing loop
while True:
    action = network.get_action(np.array(state))
    # Update action to fall in range [0,1] for linear velocity and [-1,1] for angular velocity
    a_in = [(action[0] + 1) / 2, action[1]]
    next_state, reward, done, target = env.step(a_in)
    done = 1 if episode_timesteps + 1 == max_ep else int(done)
    # On termination of episode
    if done:
        state = env.reset()
        done = False
        episode_timesteps = 0
    else:
        state = next_state
        episode_timesteps += 1

```

env.py

```

import math
import os
import random
import subprocess
import time
from os import path
import numpy as np
import rospy
import sensor_msgs.point_cloud2 as pc2
from gazebo_msgs.msg import ModelState
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from sensor_msgs.msg import PointCloud2
from quaternion import Quaternion
from std_srvs.srv import Empty
from visualization_msgs.msg import Marker
from visualization_msgs.msg import MarkerArray
GOAL_REACHED_DIST = 0.3
COLLISION_DIST = 0.35

```

```

TIME_DELTA = 0.1
# Check if the random goal position is located on an obstacle and do not accept it if it is
def check_pos(x, y):
    goal_ok = True
    if -3.8 > x > -6.2 and 6.2 > y > 3.8:
        goal_ok = False
    if -1.3 > x > -2.7 and 4.7 > y > -0.2:
        goal_ok = False
    if -0.3 > x > -4.2 and 2.7 > y > 1.3:
        goal_ok = False
    if -0.8 > x > -4.2 and -2.3 > y > -4.2:
        goal_ok = False
    if -1.3 > x > -3.7 and -0.8 > y > -2.7:
        goal_ok = False
    if 4.2 > x > 0.8 and -1.8 > y > -3.2:
        goal_ok = False
    if 4 > x > 2.5 and 0.7 > y > -3.2:
        goal_ok = False
    if 6.2 > x > 3.8 and -3.3 > y > -4.2:
        goal_ok = False
    if 4.2 > x > 1.3 and 3.7 > y > 1.5:
        goal_ok = False
    if -3.0 > x > -7.2 and 0.5 > y > -1.5:
        goal_ok = False
    if x > 4.5 or x < -4.5 or y > 4.5 or y < -4.5:
        goal_ok = False
    return goal_ok
class GazeboEnv:
    """Superclass for all Gazebo environments."""
    def __init__(self, launchfile, environment_dim):
        self.environment_dim = environment_dim
        self.odom_x = 0
        self.odom_y = 0
        self.goal_x = 1
        self.goal_y = 0.0
        self.upper = 5.0
        self.lower = -5.0
        self.velodyne_data = np.ones(self.environment_dim) * 10
        self.last_odom = None
        self.set_self_state = ModelState()
        self.set_self_state.model_name = "r1"
        self.set_self_state.pose.position.x = 0.0
        self.set_self_state.pose.position.y = 0.0
        self.set_self_state.pose.position.z = 0.0
        self.set_self_state.pose.orientation.x = 0.0
        self.set_self_state.pose.orientation.y = 0.0
        self.set_self_state.pose.orientation.z = 0.0
        self.set_self_state.pose.orientation.w = 1.0

```

```

self.gaps = [[-np.pi / 2 - 0.03, -np.pi / 2 + np.pi / self.environment_dim]]
for m in range(self.environment_dim - 1):
    self.gaps.append(
        [self.gaps[m][1], self.gaps[m][1] + np.pi / self.environment_dim]
    )
self.gaps[-1][-1] += 0.03
port = "11311"
subprocess.Popen(["roscore", "-p", port])
print("Roscore launched!")
# Launch the simulation with the given launchfile name
rospy.init_node("gym", anonymous=True)
if launchfile.startswith("/"):
    fullpath = launchfile
else:
    fullpath = os.path.join(os.path.dirname(__file__), "assets", launchfile)
if not path.exists(fullpath):
    raise IOError("File " + fullpath + " does not exist")
subprocess.Popen(["roslaunch", "-p", port, fullpath])
print("Gazebo launched!")
# Set up the ROS publishers and subscribers
self.vel_pub = rospy.Publisher("/r1/cmd_vel", Twist, queue_size=1)
self.set_state = rospy.Publisher(
    "gazebo/set_model_state", ModelState, queue_size=10
)
self.unpause = rospy.ServiceProxy("/gazebo/unpause_physics", Empty)
self.pause = rospy.ServiceProxy("/gazebo/pause_physics", Empty)
self.reset_proxy = rospy.ServiceProxy("/gazebo/reset_world", Empty)
self.publisher = rospy.Publisher("goal_point", MarkerArray, queue_size=3)
self.publisher2 = rospy.Publisher("linear_velocity", MarkerArray, queue_size=1)
self.publisher3 = rospy.Publisher("angular_velocity", MarkerArray, queue_size=1)
self.velodyne = rospy.Subscriber(
    "/velodyne_points", PointCloud2, self.velodyne_callback, queue_size=1
)
self.odom = rospy.Subscriber(
    "/r1/odom", Odometry, self.odom_callback, queue_size=1
)
# Read velodyne pointcloud and turn it into distance data, then select the minimum
value for each angle
# range as state representation
def velodyne_callback(self, v):
    data = list(pc2.read_points(v, skip_nans=False, field_names=("x", "y", "z")))
    self.velodyne_data = np.ones(self.environment_dim) * 10
    for i in range(len(data)):
        if data[i][2] > -0.2:
            dot = data[i][0] * 1 + data[i][1] * 0
            mag1 = math.sqrt(math.pow(data[i][0], 2) + math.pow(data[i][1], 2))
            mag2 = math.sqrt(math.pow(1, 2) + math.pow(0, 2))
            beta = math.acos(dot / (mag1 * mag2)) * np.sign(data[i][1])

```

```

        dist = math.sqrt(data[i][0] ** 2 + data[i][1] ** 2 + data[i][2] ** 2)
        for j in range(len(self.gaps)):
            if self.gaps[j][0] <= beta < self.gaps[j][1]:
                self.velodyne_data[j] = min(self.velodyne_data[j], dist)
                break
def odom_callback(self, od_data):
    self.last_odom = od_data
# Perform an action and read a new state
def step(self, action):
    target = False
    # Publish the robot action
    vel_cmd = Twist()
    vel_cmd.linear.x = action[0]
    vel_cmd.angular.z = action[1]
    self.vel_pub.publish(vel_cmd)
    self.publish_markers(action)
    rospy.wait_for_service("/gazebo/unpause_physics")
    try:
        self.unpause()
    except (rospy.ServiceException) as e:
        print("/gazebo/unpause_physics service call failed")
    # propagate state for TIME_DELTA seconds
    time.sleep(TIME_DELTA)
    rospy.wait_for_service("/gazebo/pause_physics")
    try:
        pass
        self.pause()
    except (rospy.ServiceException) as e:
        print("/gazebo/pause_physics service call failed")
    # read velodyne laser state
    done, collision, min_laser = self.observe_collision(self.velodyne_data)
    v_state = []
    v_state[:] = self.velodyne_data[:]
    laser_state = [v_state]
    # Calculate robot heading from odometry data
    self.odom_x = self.last_odom.pose.pose.position.x
    self.odom_y = self.last_odom.pose.pose.position.y
    quaternion = Quaternion(
        self.last_odom.pose.pose.orientation.w,
        self.last_odom.pose.pose.orientation.x,
        self.last_odom.pose.pose.orientation.y,
        self.last_odom.pose.pose.orientation.z,
    )
    euler = quaternion.to_euler(degrees=False)
    angle = round(euler[2], 4)
    # Calculate distance to the goal from the robot
    distance = np.linalg.norm(
        [self.odom_x - self.goal_x, self.odom_y - self.goal_y]

```

```

    )
    # Calculate the relative angle between the robots heading and heading toward the
goal
    skew_x = self.goal_x - self.odom_x
    skew_y = self.goal_y - self.odom_y
    dot = skew_x * 1 + skew_y * 0
    mag1 = math.sqrt(math.pow(skew_x, 2) + math.pow(skew_y, 2))
    mag2 = math.sqrt(math.pow(1, 2) + math.pow(0, 2))
    beta = math.acos(dot / (mag1 * mag2))
    if skew_y < 0:
        if skew_x < 0:
            beta = -beta
        else:
            beta = 0 - beta
    theta = beta - angle
    if theta > np.pi:
        theta = np.pi - theta
        theta = -np.pi - theta
    if theta < -np.pi:
        theta = -np.pi - theta
        theta = np.pi - theta
    # Detect if the goal has been reached and give a large positive reward
    if distance < GOAL_REACHED_DIST:
        target = True
        done = True
    robot_state = [distance, theta, action[0], action[1]]
    state = np.append(laser_state, robot_state)
    reward = self.get_reward(target, collision, action, min_laser)
    return state, reward, done, target
def reset(self):
    # Resets the state of the environment and returns an initial observation.
    rospy.wait_for_service("/gazebo/reset_world")
    try:
        self.reset_proxy()
    except rospy.ServiceException as e:
        print("/gazebo/reset_simulation service call failed")
    angle = np.random.uniform(-np.pi, np.pi)
    quaternion = Quaternion.from_euler(0.0, 0.0, angle)
    object_state = self.set_self_state
    x = 0
    y = 0
    position_ok = False
    while not position_ok:
        x = np.random.uniform(-4.5, 4.5)
        y = np.random.uniform(-4.5, 4.5)
        position_ok = check_pos(x, y)
    object_state.pose.position.x = x
    object_state.pose.position.y = y

```

```

# object_state.pose.position.z = 0.
object_state.pose.orientation.x = quaternion.x
object_state.pose.orientation.y = quaternion.y
object_state.pose.orientation.z = quaternion.z
object_state.pose.orientation.w = quaternion.w
self.set_state.publish(object_state)
self.odom_x = object_state.pose.position.x
self.odom_y = object_state.pose.position.y
# set a random goal in empty space in environment
self.change_goal()
# randomly scatter boxes in the environment
self.random_box()
self.publish_markers([0.0, 0.0])
rospy.wait_for_service("/gazebo/unpause_physics")
try:
    self.unpause()
except (rospy.ServiceException) as e:
    print("/gazebo/unpause_physics service call failed")
time.sleep(TIME_DELTA)
rospy.wait_for_service("/gazebo/pause_physics")
try:
    self.pause()
except (rospy.ServiceException) as e:
    print("/gazebo/pause_physics service call failed")
v_state = []
v_state[:] = self.velodyne_data[:]
laser_state = [v_state]
distance = np.linalg.norm(
    [self.odom_x - self.goal_x, self.odom_y - self.goal_y]
)
skew_x = self.goal_x - self.odom_x
skew_y = self.goal_y - self.odom_y
dot = skew_x * 1 + skew_y * 0
mag1 = math.sqrt(math.pow(skew_x, 2) + math.pow(skew_y, 2))
mag2 = math.sqrt(math.pow(1, 2) + math.pow(0, 2))
beta = math.acos(dot / (mag1 * mag2))
if skew_y < 0:
    if skew_x < 0:
        beta = -beta
    else:
        beta = 0 - beta
theta = beta - angle
if theta > np.pi:
    theta = np.pi - theta
    theta = -np.pi - theta
if theta < -np.pi:
    theta = -np.pi - theta
    theta = np.pi - theta

```

```

robot_state = [distance, theta, 0.0, 0.0]
state = np.append(laser_state, robot_state)
return state
def change_goal(self):
    # Place a new goal and check if its location is not on one of the obstacles
    if self.upper < 10:
        self.upper += 0.004
    if self.lower > -10:
        self.lower -= 0.004
    goal_ok = False
    while not goal_ok:
        self.goal_x = self.odom_x + random.uniform(self.upper, self.lower)
        self.goal_y = self.odom_y + random.uniform(self.upper, self.lower)
        goal_ok = check_pos(self.goal_x, self.goal_y)
def random_box(self):
    # Randomly change the location of the boxes in the environment on each reset to
    randomize the training
    # environment
    for i in range(4):
        name = "cardboard_box_" + str(i)
        x = 0
        y = 0
        box_ok = False
        while not box_ok:
            x = np.random.uniform(-6, 6)
            y = np.random.uniform(-6, 6)
            box_ok = check_pos(x, y)
            distance_to_robot = np.linalg.norm([x - self.odom_x, y - self.odom_y])
            distance_to_goal = np.linalg.norm([x - self.goal_x, y - self.goal_y])
            if distance_to_robot < 1.5 or distance_to_goal < 1.5:
                box_ok = False
        box_state = ModelState()
        box_state.model_name = name
        box_state.pose.position.x = x
        box_state.pose.position.y = y
        box_state.pose.position.z = 0.0
        box_state.pose.orientation.x = 0.0
        box_state.pose.orientation.y = 0.0
        box_state.pose.orientation.z = 0.0
        box_state.pose.orientation.w = 1.0
        self.set_state.publish(box_state)
def publish_markers(self, action):
    # Publish visual data in Rviz
    markerArray = MarkerArray()
    marker = Marker()
    marker.header.frame_id = "odom"
    marker.type = marker.CYLINDER
    marker.action = marker.ADD

```

```
marker.scale.x = 0.1
marker.scale.y = 0.1
marker.scale.z = 0.01
marker.color.a = 1.0
marker.color.r = 0.0
marker.color.g = 1.0
marker.color.b = 0.0
marker.pose.orientation.w = 1.0
marker.pose.position.x = self.goal_x
marker.pose.position.y = self.goal_y
marker.pose.position.z = 0
markerArray.markers.append(marker)
self.publisher.publish(markerArray)
markerArray2 = MarkerArray()
marker2 = Marker()
marker2.header.frame_id = "odom"
marker2.type = marker.CUBE
marker2.action = marker.ADD
marker2.scale.x = abs(action[0])
marker2.scale.y = 0.1
marker2.scale.z = 0.01
marker2.color.a = 1.0
marker2.color.r = 1.0
marker2.color.g = 0.0
marker2.color.b = 0.0
marker2.pose.orientation.w = 1.0
marker2.pose.position.x = 5
marker2.pose.position.y = 0
marker2.pose.position.z = 0
markerArray2.markers.append(marker2)
self.publisher2.publish(markerArray2)
markerArray3 = MarkerArray()
marker3 = Marker()
marker3.header.frame_id = "odom"
marker3.type = marker.CUBE
marker3.action = marker.ADD
marker3.scale.x = abs(action[1])
marker3.scale.y = 0.1
marker3.scale.z = 0.01
marker3.color.a = 1.0
marker3.color.r = 1.0
marker3.color.g = 0.0
marker3.color.b = 0.0
marker3.pose.orientation.w = 1.0
marker3.pose.position.x = 5
marker3.pose.position.y = 0.2
marker3.pose.position.z = 0
```

```

markerArray3.markers.append(marker3)
self.publisher3.publish(markerArray3)
@staticmethod
def observe_collision(laser_data):
    # Detect a collision from laser data
    min_laser = min(laser_data)
    if min_laser < COLLISION_DIST:
        return True, True, min_laser
    return False, False, min_laser
@staticmethod
def get_reward(target, collision, action, min_laser):
    if target:
        return 100.0
    elif collision:
        return -100.0
    else:
        r3 = lambda x: 1 - x if x < 1 else 0.0
        return action[0] / 2 - abs(action[1]) / 2 - r3(min_laser) / 2

```

ui.py

Data structure for implementing experience replay

```

import random
from collections import deque
import numpy as np
class ReplayBuffer(object):
    def __init__(self, buffer_size, random_seed=123):
        """
        The right side of the deque contains the most recent experiences
        """
        self.buffer_size = buffer_size
        self.count = 0
        self.buffer = deque()
        random.seed(random_seed)
    def add(self, s, a, r, t, s2):
        experience = (s, a, r, t, s2)
        if self.count < self.buffer_size:
            self.buffer.append(experience)
            self.count += 1
        else:
            self.buffer.popleft()
            self.buffer.append(experience)
    def size(self):
        return self.count

```

```

def sample_batch(self, batch_size):
    batch = []
    if self.count < batch_size:
        batch = random.sample(self.buffer, self.count)
    else:
        batch = random.sample(self.buffer, batch_size)
    s_batch = np.array([_[0] for _ in batch])
    a_batch = np.array([_[1] for _ in batch])
    r_batch = np.array([_[2] for _ in batch]).reshape(-1, 1)
    t_batch = np.array([_[3] for _ in batch]).reshape(-1, 1)
    s2_batch = np.array([_[4] for _ in batch])
    return s_batch, a_batch, r_batch, t_batch, s2_batch
def clear(self):
    self.buffer.clear()
    self.count = 0

```

Output

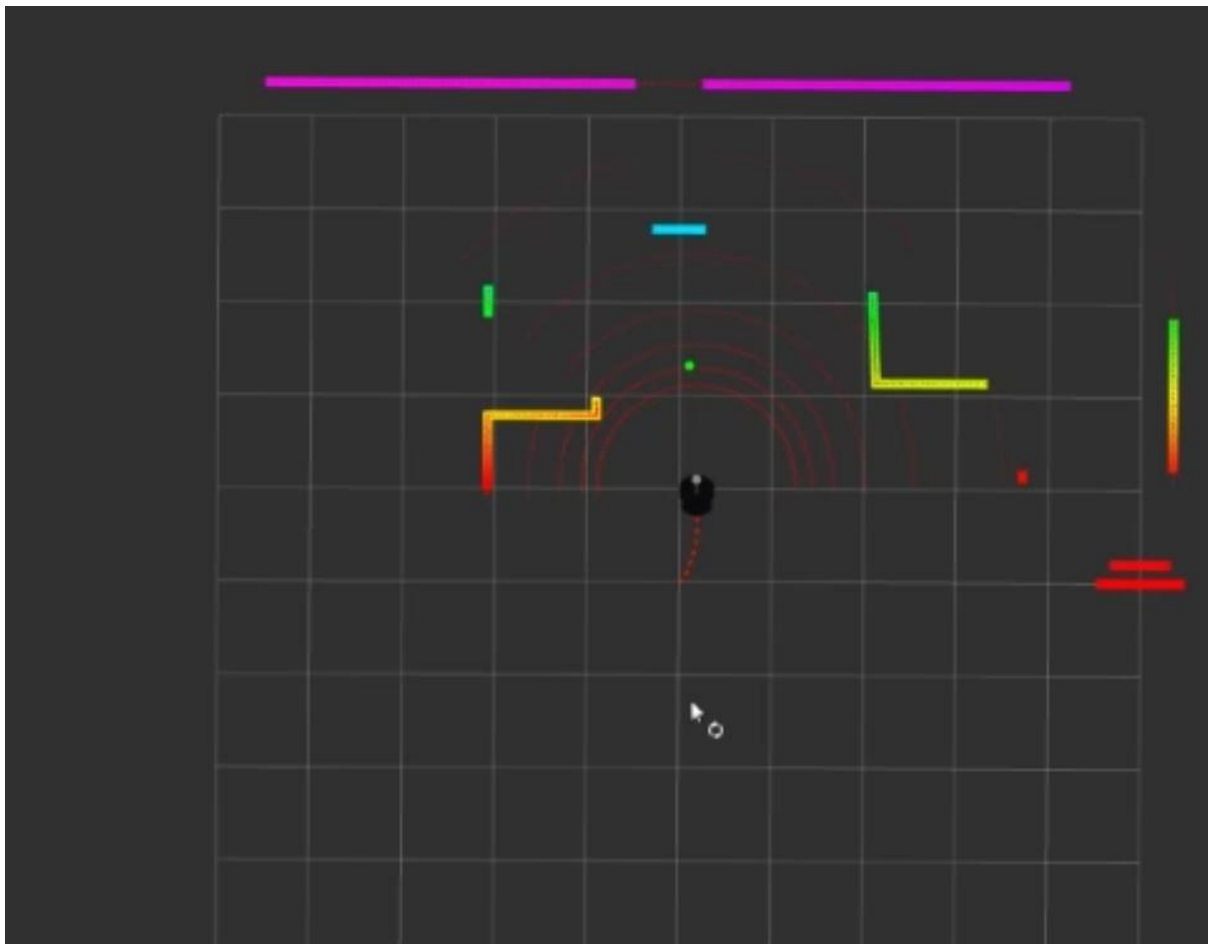


Fig 1 : Output 1



Fig 2 : Output 2

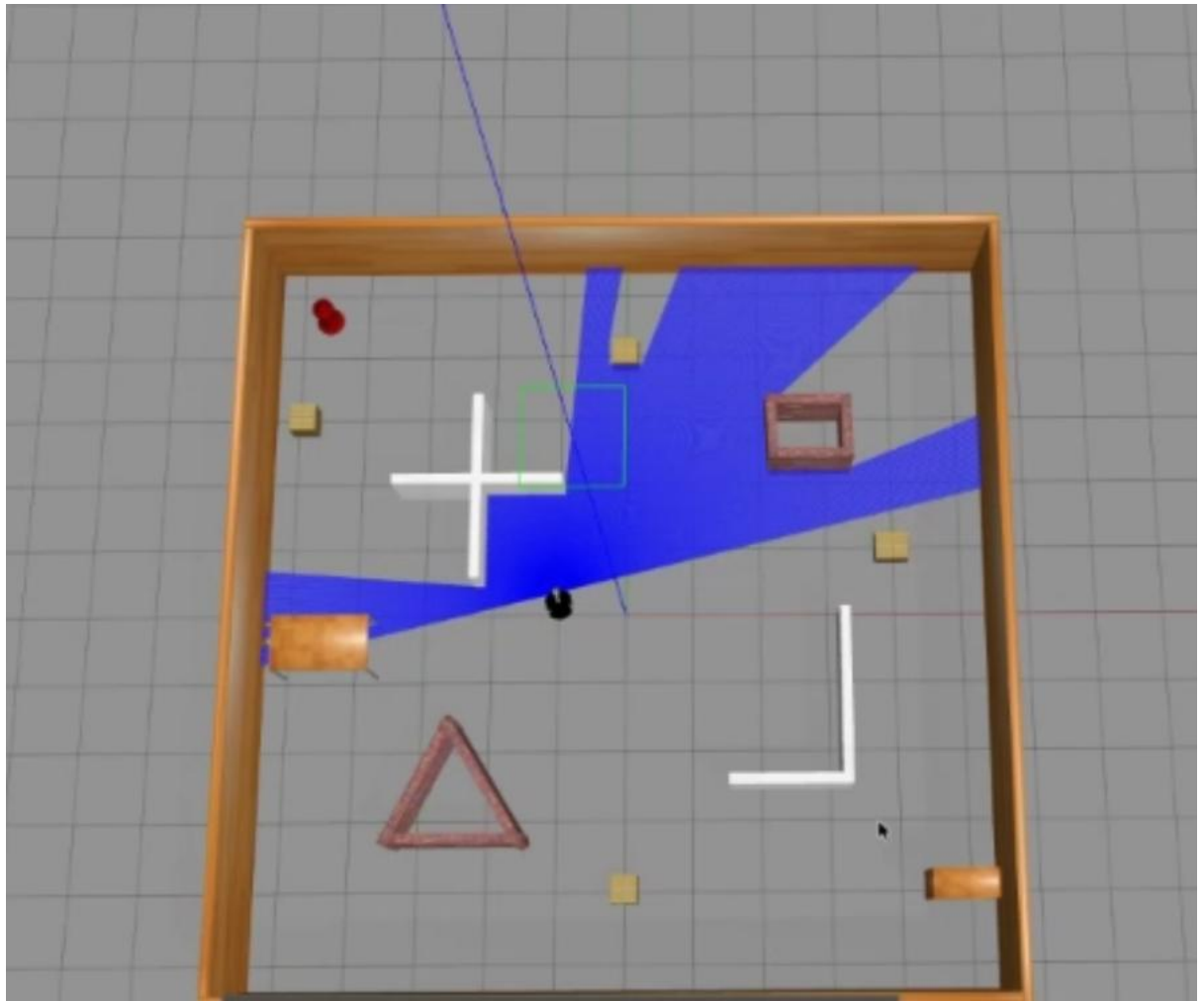


Fig 3 : Output 3