COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MASTER'S DEGREE IN INDUSTRIAL ENGINEERING

MASTER'S THESIS

TRAINING A VIRTUAL REINFORCEMENT LEARNING AGENT FOR OBSTACLE AVOIDANCE AND TRANSFERRING IT TO A REAL MOBILE ROBOT

Author: Carrera Fresneda, Javier

Co-Director: Boal Martín-Larrauri, Jaime

Co-Director: Zamora Macho, Juan Luis

Madrid

August 2023

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

"Training a Virtual Reinforcement Learning Agent for Obstacle Avoidance and Transferring It to a Real Mobile Robot"

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2022-2023 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

[illegible]

Fdo.: Javier Carrera Fresneda Fecha: 21/08/2023

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

[illegible] Firmado digitalmente por BOAL MARTIN LARRAURI JAIME - 05304600H
Fecha: 2023.08.22 20:06:18 +02'00'

Fdo.: …………………………… Fecha: ……/ ……/ ……

EL DIRECTOR DEL PROYECTO

Fdo.: [illegible] Fecha: 22/ 09/ 2023

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MASTER'S DEGREE IN INDUSTRIAL ENGINEERING

MASTER'S THESIS

TRAINING A VIRTUAL REINFORCEMENT LEARNING AGENT FOR OBSTACLE AVOIDANCE AND TRANSFERRING IT TO A REAL MOBILE ROBOT

Author: Carrera Fresneda, Javier

Co-Director: Boal Martín-Larrauri, Jaime

Co-Director: Zamora Macho, Juan Luis

Madrid

August 2023

TRAINING A VIRTUAL REINFORCEMENT LEARNING AGENT FOR OBSTACLE AVOIDANCE AND TRANSFERRING IT TO A REAL MOBILE ROBOT

Author: Carrera Fresneda, Javier

Co-Director: Boal Martín-Larrauri, Jaime

Co-Director: Zamora Macho, Juan Luis

ABSTRACT

In a world with drones becoming a common sight in our skies and roads increasingly shared with autonomous vehicles, the quest for efficiency, safety, and innovation takes center stage.

The increasing demand for autonomous mobile robots (AMRs) across diverse sectors such as industrial automation [1] , healthcare [2] [3] , and military [4] has pushed the development of intelligent algorithms that enable these systems to navigate and execute tasks in varying environments without human intervention. Reinforcement learning can provide the backbone for these systems to traverse complex environments, recognize obstacles, detect anomalies, and adapt in real-time to varying situations.

However, the path to creating these systems is filled with challenges, primarily due to the complexities and risks of experimenting in real-world environments. This is where the value of simulation to reality (sim-to-real) [13] becomes pronounced. Developing, testing, and refining autonomous algorithms in simulated environments presents numerous advantages, including accelerated development, cost efficiency, and safety. This thesis aims to explore the application of reinforcement learning to the development of autonomous navigation behaviors in a differential vehicle and a drone, addressing the simulation-to-real gap and providing insights into the challenges and complexities of real-world navigation.

PROJECT OBJECTIVES

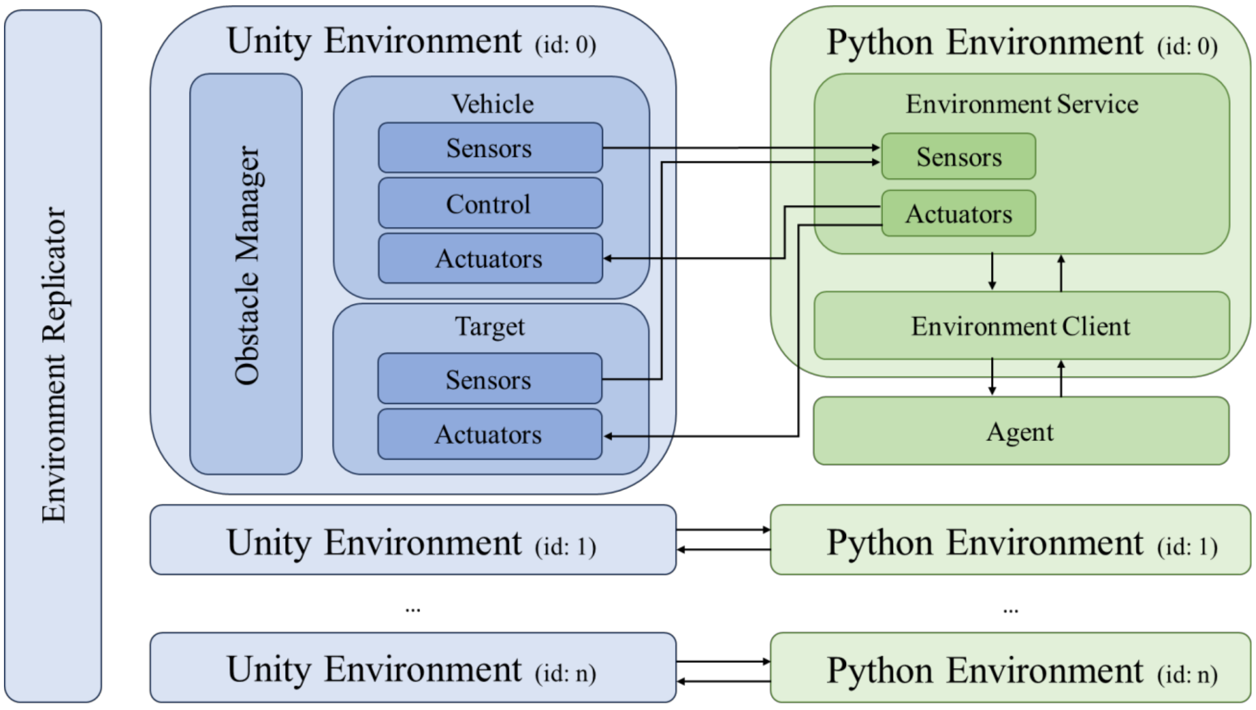
Development of Autonomous Behaviors: To design and develop autonomous navigation behaviors for both a differential vehicle and a drone using reinforcement learning.

Simulation-to-Real Transition: To validate the trained models in a simulated environment and then successfully transition and adapt them for real-world deployment.

Integration of Technologies: Seamless integration of technologies such as Unity, Simulink, ROS2, Docker, and reinforcement learning frameworks to achieve a holistic system that can function in both simulated and real-world settings.

SIMULATION METHODOLOGY

The simulation environment is designed to provide for the requirements of both the differential vehicle and the drone. It incorporates two implementations based on Unity Engine: a simplified environment for rapid testing and validation, and a more realistic environment that integrates with existing systems in MATLAB and Simulink. The latter incorporates more complex dynamics and control systems, thus providing an evaluation platform for the reinforcement learning algorithms. The integration of these environments with the Distributed Distributional Deep Deterministic Policy Gradient (D4PG) algorithm ensures that the autonomous agents are trained under varying conditions, preparing them for the unpredictabilities of the real world.



System Implementation Block Diagram

SIMULATION RESULTS

The simulation results provide a comprehensive analysis of the performance of the autonomous agents, the differential vehicle and the drone, in both simplified and realistic environments.

The differential vehicle exhibited a robust performance, achieving a success rate of 83.33% in the simplified environment and 78.33% in the realistic environment. The agent's ability to navigate with minimal collisions, even in the presence of dynamic obstacles, underscores the stability of the reinforcement learning model applied.

Differential Vehicle's Success Rates in the Simplified Environment

Criteria	Episodes	Percentage (%)
Target Reached	100	83.33
Collision with Static Obstacle	2	1.67
Collision with Moving Obstacle	6	5.00
Maximum Time Reached	12	10.00

Differential Vehicle's Success Rates in the Realistic Environment

Criteria	**Episodes**	**Percentage (%)**
Target Reached	94	78.33
Collision with Static Obstacle	4	3.33
Collision with Moving Obstacle	10	8.33
Maximum Time Reached	12	10.00

The drone, on the other hand, showcased a slightly different learning curve. In the simplified environment, it achieved a remarkable success rate of 90%, demonstrating rapid adaptation and efficient decision-making. However, when transitioned to the realistic environment, its success rate slightly decreased to 76.67%. This drop can be attributed to the more complex dynamics and control challenges inherent to aerial vehicles.

Drone's Success Rates in the Simplified Environment

Criteria	**Episodes**	**Percentage (%)**
Target Reached	108	90.00
Collision with Static Obstacle	2	1.67
Collision with Moving Obstacle	4	3.33
Maximum Time Reached	6	5.00

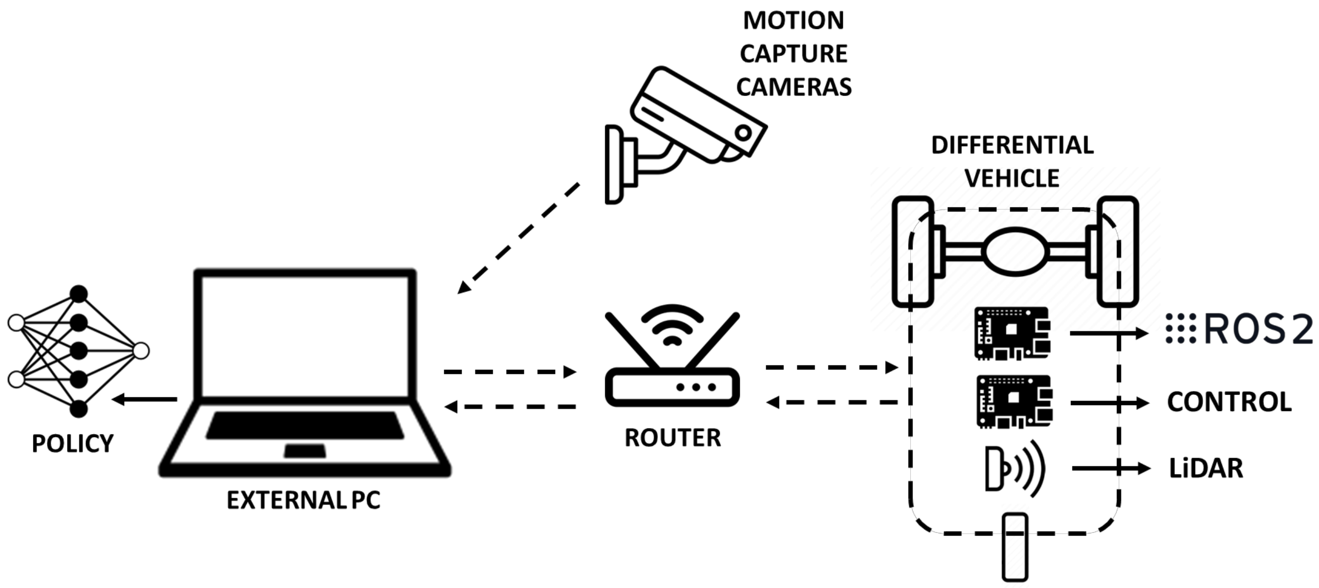
Drone's Success Rates in the Realistic Environment

Criteria	**Episodes**	**Percentage (%)**
Target Reached	92	76.67
Collision with Static Obstacle	6	5.00
Collision with Moving Obstacle	14	11.67
Maximum Time Reached	8	6.67

Despite the differences in their performances, both agents' results validate the effectiveness of the D4PG algorithm in training autonomous agents for navigation tasks.

REAL-WORLD METHODOLOGY

The real-world methodology covers the transition of the autonomous agents from the simulation environment to a physical laboratory setting. The differential vehicle is equipped with two Raspberry Pi units, one dedicated to the control system and the other for communication purposes, interfacing with ROS2. The vehicle's perception of its environment is enabled by the LiDAR sensor, which provides a two-dimensional "slice" of the surrounding environment, essential for obstacle detection and avoidance. Additionally, external motion capture cameras are employed for accurate pose estimation. The control systems manage the vehicle's linear and angular velocities, employing the same control algorithm as in the realistic simulation environment.



Real-World Environment Setup

The real-world testing environment features static obstacles made up of cardboard boxes, which can be rearranged for flexible testing. A moving obstacle was introduced in one of the tests, serving primarily as an exploratory component of the study.

REAL-WORLD EXPERIMENTAL RESULTS

In the real-world experimental phase, the vehicle's performance was evaluated based on its ability to navigate autonomously, avoid obstacles, and reach the target destination within a specified time frame. The results of the real-world experiments revealed a success rate of 73.33% in reaching the target without collisions. The vehicle's trajectories were recorded and analyzed, revealing smooth and direct paths towards the destination in successful episodes. However, in episodes where collisions occurred, the trajectories indicated the vehicle's inability to navigate around obstacles effectively. The observed discrepancies can be attributed to several factors, including differences in obstacle geometry, vehicle geometry and dimensions, sensor noise, and the inherent complexities of the real world.

Differential Vehicle's Success Rates in the Real Environment

Criteria	Episodes	Percentage (%)
Target Reached	11	73.33
Collision with Obstacle	3	20.00
Maximum Time Reached	1	6.67

CONCLUSIONS AND FUTURE WORK

This project successfully demonstrated the feasibility of using reinforcement learning for the development of autonomous navigation behaviors in both a differential vehicle and a drone. The project also addressed the simulation-to-real gap, with the differential vehicle successfully transitioning to the real-world environment. The integration of various technologies, including Unity, Simulink, ROS2, Docker, and reinforcement learning frameworks, facilitated the development and simulation phases and ensured reproducibility and compatibility across platforms.

Future work could focus on several areas to further enhance the performance of the autonomous agents. Incorporating cameras as additional sensors could provide richer sensory data and enable the agents to better perceive their environment. Training the differential vehicle agent with a position control strategy, similar to the drone agent, could provide a more direct comparison between the two agents and potentially improve the differential vehicle's performance. Advanced neural network architectures, such as Long Short-Term Memory (LSTM) or transformer architectures, could enhance the agents' ability to learn and remember temporal dependencies in the environment. Extending the testing scenarios to include dynamic obstacles would provide a more comprehensive evaluation of the agents' performance. Transferring the drone agent to a real-world environment would provide valuable insights into the challenges and complexities of real-world aerial navigation.

ENTRENAMIENTO DE UN AGENTE VIRTUAL MEDIANTE APRENDIZAJE POR REFUERZO PARA LA EVASIÓN DE OBSTÁCULOS Y TRANSFERENCIA A UN ROBOT MÓVIL REAL

Autor: Carrera Fresneda, Javier

Co-Director: Boal Martín-Larrauri, Jaime

Co-Director: Zamora Macho, Juan Luis

<u>RESUMEN</u>

En un mundo en el que los drones son cada vez más habituales en nuestros cielos y las carreteras están cada vez más transitadas por vehículos autónomos, la búsqueda de eficiencia, seguridad e innovación ocupa un lugar central.

La creciente demanda de robots móviles autónomos (AMR) en sectores tan diversos como la automatización industrial [1] , la sanidad [2] [3] y el ejército [4] ha impulsado el desarrollo de algoritmos inteligentes que permitan a estos sistemas navegar y ejecutar tareas en entornos variables sin intervención humana. El aprendizaje por refuerzo puede servir de base para que estos sistemas atraviesen entornos complejos, reconozcan obstáculos, detecten anomalías y se adapten en tiempo real a situaciones variables.

Sin embargo, el camino hacia la creación de estos sistemas está lleno de desafíos, principalmente debido a las complejidades y los riesgos de experimentar en entornos reales. Aquí es donde se hace patente el valor de la simulación a la realidad (sim-to-real) [13] . Desarrollar, probar y perfeccionar algoritmos autónomos en entornos simulados presenta numerosas ventajas, como la aceleración del desarrollo, la rentabilidad y la seguridad. Esta tesis pretende explorar la aplicación del aprendizaje por refuerzo al desarrollo de comportamientos de navegación autónoma en un vehículo diferencial y un dron, abordando la brecha simulación-realidad y proporcionando una visión de los retos y complejidades de la navegación en el mundo real.

OBJETIVOS DEL PROYECTO

Desarrollo de comportamientos autónomos: Diseñar y desarrollar comportamientos de navegación autónoma tanto para un vehículo diferencial como para un dron utilizando el aprendizaje por refuerzo.

Transición de la simulación a la realidad: Validar los modelos entrenados en un entorno simulado y, a continuación, realizar con éxito la transición y adaptarlos para su despliegue en el mundo real.

Integración de tecnologías: Integración perfecta de tecnologías como Unity, Simulink, ROS2, Docker y marcos de aprendizaje por refuerzo para lograr un sistema holístico que pueda funcionar tanto en entornos simulados como en el mundo real.

METODOLOGÍA DE SIMULACIÓN

El entorno de simulación está diseñado para satisfacer los requisitos tanto del vehículo diferencial como del dron. Incorpora dos implementaciones basadas en Unity Engine: un entorno simplificado para pruebas

y validación rápidas, y un entorno más realista que se integra con los sistemas existentes en MATLAB y Simulink. Este último incorpora sistemas dinámicos y de control más complejos, proporcionando así una plataforma de evaluación para los algoritmos de aprendizaje por refuerzo. La integración de estos entornos con el algoritmo Distributed Distributional Deep Deterministic Policy Gradient (D4PG) garantiza el entrenamiento de los agentes autónomos en condiciones variables, preparándolos para las imprevisibilidades del mundo real.

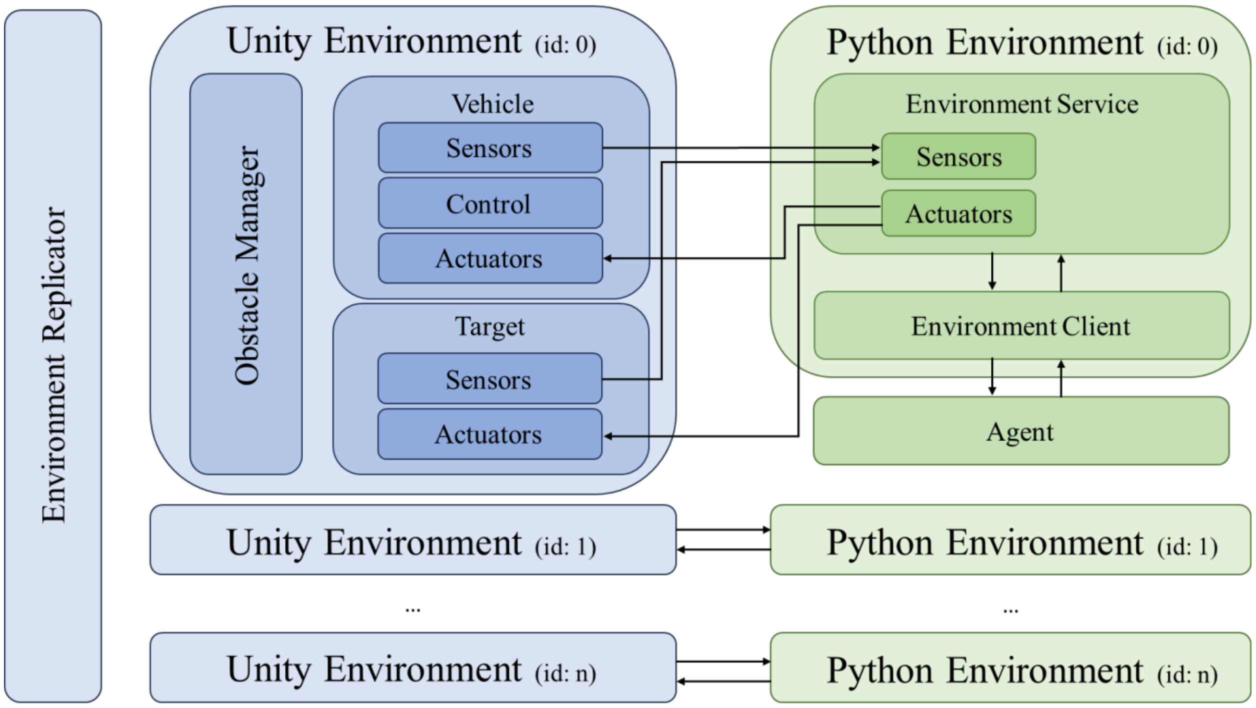


Diagrama de Bloques del Sistema Implementado

RESULTADOS DE LA SIMULACIÓN

Los resultados de la simulación proporcionan un análisis exhaustivo del rendimiento de los agentes autónomos, el vehículo diferencial y el dron, tanto en entornos simplificados como realistas.

El vehículo diferencial mostró un rendimiento robusto, alcanzando una tasa de éxito del 83,33% en el entorno simplificado y del 78,33% en el entorno realista. La capacidad del agente para navegar con colisiones mínimas, incluso en presencia de obstáculos dinámicos, subraya la estabilidad del modelo de aprendizaje por refuerzo aplicado.

Tasas de Éxito del Vehículo Diferencial en el Entorno Simplificado

Criteria	Episodes	Percentage (%)
Target Reached	100	83.33

Collision with Static Obstacle	2	1.67
Collision with Moving Obstacle	6	5.00
Maximum Time Reached	12	10.00

Tasas de Éxito del Vehículo Diferencial en el Entorno Realista

Criteria	**Episodes**	**Percentage (%)**
Target Reached	94	78.33
Collision with Static Obstacle	4	3.33
Collision with Moving Obstacle	10	8.33
Maximum Time Reached	12	10.00

El dron mostró una curva de aprendizaje ligeramente distinta. En el entorno simplificado, alcanzó una notable tasa de éxito del 90%, demostrando una rápida adaptación y una toma de decisiones eficaz. Sin embargo, al pasar al entorno realista, su tasa de éxito disminuyó ligeramente hasta el 76,67%. Este descenso puede atribuirse a la mayor complejidad de la dinámica y los retos de control inherentes a los vehículos aéreos.

Tasas de Éxito del Dron en el Entorno Simplificado

Criteria	**Episodes**	**Percentage (%)**
Target Reached	108	90.00
Collision with Static Obstacle	2	1.67
Collision with Moving Obstacle	4	3.33
Maximum Time Reached	6	5.00

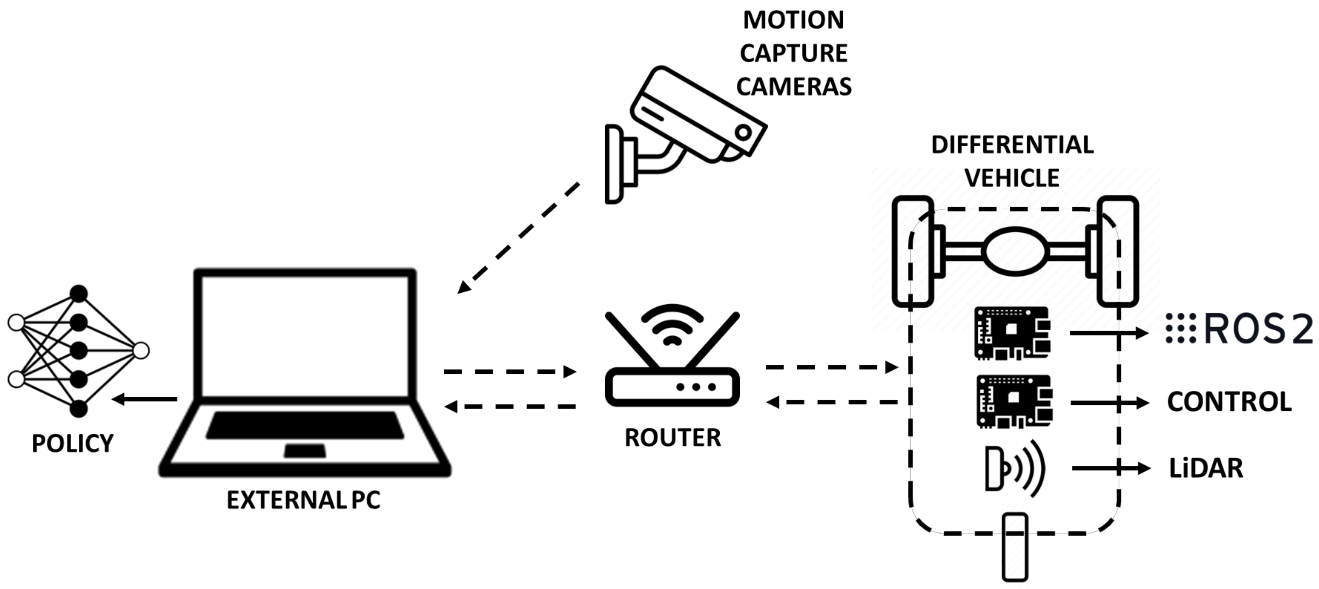
Tasas de Éxito del Dron en el Entorno Realista

Criteria	**Episodes**	**Percentage (%)**
Target Reached	92	76.67
Collision with Static Obstacle	6	5.00
Collision with Moving Obstacle	14	11.67
Maximum Time Reached	8	6.67

A pesar de las diferencias en su rendimiento, los resultados de ambos agentes validan la eficacia del algoritmo D4PG en el entrenamiento de agentes autónomos para tareas de navegación.

METODOLOGÍA DEL MUNDO REAL

La metodología del mundo real abarca la transición de los agentes autónomos desde el entorno de simulación a un entorno físico de laboratorio. El vehículo diferencial está equipado con dos unidades Raspberry Pi, una dedicada al sistema de control y la otra con fines de comunicación mediante ROS2. El sensor LiDAR permite que el vehículo perciba su entorno, ya que proporciona un "corte" bidimensional del entorno, esencial para la detección y evasión de obstáculos. Además, se emplean cámaras externas de captura de movimiento para estimar con precisión la posición y orientación. Los sistemas de control gestionan las velocidades lineal y angular del vehículo, empleando el mismo algoritmo de control que en el entorno de simulación realista.



Configuración del Entorno Real

El entorno de pruebas real presenta obstáculos estáticos formados por cajas de cartón, que pueden reorganizarse para realizar pruebas flexibles. En una de las pruebas se introdujo un obstáculo móvil, que sirvió principalmente como componente exploratorio del estudio.

RESULTADOS EXPERIMENTALES EN CONDICIONES REALES

En la fase de experimentación en condiciones reales, el rendimiento del vehículo se evaluó en función de su capacidad para desplazarse de forma autónoma, evitar obstáculos y alcanzar el destino en un tiempo determinado. Los resultados de los experimentos en condiciones reales revelaron una tasa de éxito del 73,33% en el alcance del objetivo sin colisiones. Se registraron y analizaron las trayectorias del vehículo, que revelaron recorridos suaves y directos hacia el destino en los episodios exitosos. Sin embargo, en los episodios en los que se produjeron colisiones, las trayectorias indicaron la incapacidad del vehículo para sortear obstáculos con eficacia. Las discrepancias observadas pueden atribuirse a varios factores, como las diferencias en la geometría de los obstáculos, la geometría y dimensiones del vehículo, el ruido de los sensores y las complejidades inherentes al mundo real.

Tasas de Éxito del Vehículo Diferencial en el Entorno Real

Criteria	Episodes	Percentage (%)
Target Reached	11	73.33
Collision with Obstacle	3	20.00
Maximum Time Reached	1	6.67

CONCLUSIONES Y TRABAJO FUTURO

Este proyecto ha demostrado con éxito la viabilidad de utilizar el aprendizaje por refuerzo para el desarrollo de comportamientos de navegación autónoma tanto en un vehículo diferencial como en un dron. El proyecto también abordó la brecha simulación-realidad, con el vehículo diferencial realizando con éxito la transición al entorno del mundo real. La integración de varias tecnologías, como Unity, Simulink, ROS2, Docker y marcos de aprendizaje por refuerzo, facilitó las fases de desarrollo y simulación y garantizó la reproducibilidad y compatibilidad entre plataformas.

El trabajo futuro podría centrarse en varias áreas para mejorar aún más el rendimiento de los agentes autónomos. La incorporación de cámaras como sensores adicionales podría proporcionar datos sensoriales más ricos y permitir a los agentes percibir mejor su entorno. Entrenar al agente del vehículo diferencial con una estrategia de control de posición, similar a la del agente del dron, podría proporcionar una comparación más directa entre los dos agentes y mejorar potencialmente el rendimiento del vehículo diferencial. Las arquitecturas avanzadas de redes neuronales, como Long Short-Term Memory (LSTM) o transformers, podrían mejorar la capacidad de los agentes para aprender y recordar las dependencias temporales del entorno. La ampliación de los escenarios de prueba para incluir obstáculos dinámicos proporcionaría una evaluación más completa del rendimiento de los agentes. Trasladar el agente dron a un entorno real proporcionaría información valiosa sobre los retos y complejidades de la navegación aérea en el mundo real.

Contents

Figures

Tables

CHAPTER 1: INTRODUCTION AND CONTEXT

In a world with drones becoming a common sight in our skies and roads increasingly shared with autonomous vehicles, the quest for efficiency, safety, and innovation takes center stage. The growing need for autonomous systems across various sectors such as transport, emergencies and military, has generated significant interest in reinforcement learning. This technology promises not only to revolutionize the field of mobile robotics but also to provide cost-effective and efficient solutions to real-world challenges.

This chapter discusses the significance of reinforcement learning in addressing these challenges, tracing back its history and foundational concepts. The main algorithms, particularly those suitable for continuous action spaces, are detailed. In the realm of mobile robotics, the unique challenges that reinforcement learning can tackle are highlighted, and its position relative to other techniques is assessed. By showcasing how each segment of a robotic system relates to reinforcement learning principles, this chapter sets the groundwork for the detailed discussions on methodologies and results in the following sections.

1.1. Motivation and Relevance of the Project

Autonomous mobile robots (AMRs) have been gaining rapid traction in diverse sectors due to the significant advantages they bring in terms of efficiency, precision, and safety. They leverage sensors, actuators, and intelligent algorithms to navigate and execute tasks in varying environments without human intervention.

Industrial Automation and Manufacturing

AMRs are used for material handling, picking, and sorting, increasing the speed and accuracy of production lines [1] .

Healthcare

Patient Assistance: Robots in hospitals can help in transporting medicines, meals, or lab specimens [2] .

Disinfection: Given the current global health scenario, robots equipped with UV lights or liquid disinfectants have been used in hospitals and public places to prevent the spread of contagious diseases [3] .

Military and Defense

Reconnaissance: Robots can be used for surveillance purposes, scouting areas without putting human lives in danger [4] .

Bomb Disposal: Specialized robots can be utilized to defuse or safely detonate explosive devices [5] .

Warehouse and Logistics

Inventory Management: Robots equipped with scanners can quickly navigate aisles, taking stock of inventory [6] .

Material Handling: AMRs can transport goods from one place to another within warehouses, often using localization and mapping technologies to navigate complex environments [7] .

Search and Rescue

In situations like natural disasters, robots can be deployed to search for survivors in areas deemed too dangerous for human rescuers, such as collapsed buildings or flood zones [8] .

Agriculture

Precision Farming: AMRs can be equipped with sensors and cameras to monitor crop health, soil moisture levels, and pest activity, ensuring precise application of fertilizers and pesticides [9] .

Harvesting: Some robots are designed to pick fruits or vegetables without damaging them, utilizing computer vision and soft grippers [10] .

Maintenance and Inspection

AMRs can be used to inspect infrastructure like bridges, pipelines, powerlines and buildings. They can navigate hard-to-reach areas, providing real-time feedback on structural integrity [11] .

Space Exploration

Planetary rovers, like those used on Mars, are examples of AMRs that can traverse alien terrains, gather samples, and send data back to Earth without direct human intervention [12] .

Reinforcement learning provides the backbone for these systems to navigate complex environments, recognize obstacles, detect anomalies and adapt in real-time to varying situations. However, the path to creating these systems is filled with challenges, primarily due to the complexities and risks of experimenting in real-world environments.

This is where the value of simulation to reality (sim-to-real) [13] becomes pronounced. Developing, testing, and refining autonomous algorithms in simulated environments presents numerous advantages:

Accelerated Development: In a controlled simulation, one can run countless scenarios, speeding up the learning process of the autonomous system. Iterative adjustments can be made quickly, significantly reducing the time from concept to real-world application.

Cost Efficiency: Real-world experimentation can be expensive. The potential for equipment damage or loss, coupled with the time and resources required for each test, can skyrocket costs. Simulated environments bypass these financial implications, allowing for vast amounts of testing without the associated high costs.

Safety: Running preliminary tests in simulations minimizes the risk associated with early-stage failures, ensuring that by the time the system is deployed in the real world, most major flaws have been addressed. This is particularly relevant in sectors like transport and military, where mistakes can have critical consequences.

1.2. History of Reinforcement Learning

Reinforcement Learning (RL) is a subset of machine learning where agents learn how to behave in an environment by performing actions and receiving rewards. The history of RL is rich, spanning several decades with pivotal moments and significant contributions.

Origins in Psychology and Control Theory (1950s-1970s):

Trial and Error Learning: The roots of RL can be traced back to the psychological concept of trial-and-error learning, where organisms learn to perform actions that yield the most rewards Available at: https://arxiv.org/abs/2009.13303.

- [14] .
- **Optimal Control:** On the engineering side, the theory of optimal control, particularly the Bellman Equation [15] , laid the foundation for value-based RL algorithms.

Temporal Difference Learning (1980s):

- **TD-Learning:** Richard Sutton's introduction of temporal difference (TD) learning [16] bridged the gap between dynamic programming [17] and Monte Carlo methods [18]. It allowed for learning directly from raw experience without a model of the environment.
- **Q-Learning:** Watkins and Dayan introduced Q-learning, an off-policy TD control algorithm [19] . It demonstrated the potential of learning optimal policies directly from interaction with an environment.

Neural Networks and RL (1990s):

- **Function Approximation:** Using neural networks as function approximators in RL became popular, allowing RL to handle more complex, high-dimensional environments.
- **Backgammon:** Gerald Tesauro's TD-Gammon [20] , a backgammon-playing program, was one of the first successes of RL combined with neural networks.

Deep Reinforcement Learning (2010s):

- **Deep Q-Networks (DQN):** Combining deep neural networks with Q-learning, DeepMind's DQN was able to achieve human-level performance on many Atari 2600 games [21] , marking a significant advancement in the field.
- **Policy Gradient Methods:** Algorithms like TRPO (Trust Region Policy Optimization) and A3C (Asynchronous Advantage Actor-Critic) provided new ways to train agents, focusing on directly optimizing the policy.

Safety and Multi-agent RL (2020s):

- **Exploration vs. Safety:** With RL being explored for real-world applications like self-driving cars, research on safe exploration, where the agent actively avoids catastrophic actions, became essential.
- **Multi-agent Settings:** Environments with multiple interacting agents introduced new challenges and dynamics, leading to the development of algorithms that can handle cooperative and competitive scenarios [22] .

Reinforcement Learning with Human Feedback (Late 2010s - 2020s):

- **Transformer Architectures Training:** As transformer models like BERT and GPT emerged in NLP, integrating reinforcement learning with human feedback became notable. Researchers fine-tuned these models using reward signals from human evaluations, promoting safer and more desired outputs. OpenAI's projects highlighted this approach's potential, using rankings of model responses to guide reinforcement learning and better align AI outputs with human preferences [23] .

1.3. General Scheme of Reinforcement Learning

1.3.1. Basic Components of the Process

Agent: This refers to the learner or decision-maker that interacts with the environment. In the context of this project, the intelligence behind the behavior of the autonomous differential vehicle and drone are examples of agents. The agent makes decisions based on the information it gathers from the environment.

Environment: The environment is everything that the agent interacts with and learns from. It provides feedback to the agent based on the agent's actions.

Actions: Actions represent the set of all possible moves that the agent can make. These can be discrete, continuous, or a combination of both.

States: The state of an environment at any given time refers to the current configuration or situation of that environment.

Rewards: Rewards are feedback from the environment based on the agent's actions. A positive reward indicates a favorable action, whereas a negative reward indicates an unfavorable action. The goal of the agent is to maximize its cumulative reward over time.

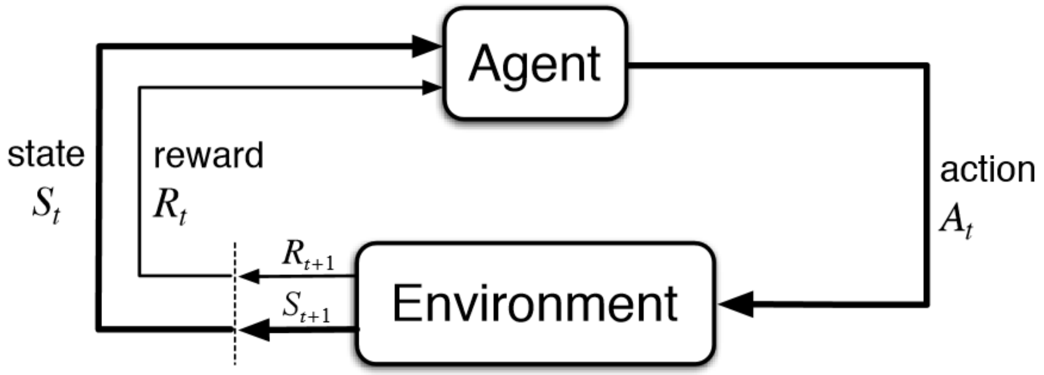


Figure 1: Reinforcement Learning General Scheme
Image Source: [2103.15781.pdf (arxiv.org)]

1.4. The Decision-Making Cycle in Reinforcement Learning

Reinforcement learning is built upon the iterative interaction between an agent and its environment. This iterative process is called the decision-making cycle:

1. The agent observes the current state of the environment.
2. Based on the observed state and its policy, the agent decides on an action to take.
3. The agent performs the chosen action in the environment.
4. After taking the action, the agent receives a reward from the environment. This reward serves as feedback, indicating how good or bad the action was.
5. Using the reward as feedback, the agent updates its knowledge to make better decisions in the future.
6. The process repeats until a termination condition is met, such as the end of an episode.

By continuously going through this decision-making cycle, the agent learns to improve its policy over time, aiming to maximize its cumulative rewards.

1.4.1. Fundamental Concepts

Policy (π): A policy defines the agent's behavior. It is a mapping from states to actions, determining what action the agent should take in each state. The policy can be deterministic or stochastic. A deterministic policy provides a specific action for each state, while a stochastic policy provides a probability distribution over actions for each state. Neural networks can be used to approximate these deterministic policies.

Value Function: The value function approximates the expectation of cumulative future rewards a particular agent can expect to receive starting from a given state or state-action pair. In deep reinforcement learning deep neural networks are employed to represent and approximate this function due to their capability to generalize and handle large or continuous state spaces. Two types of value functions can be defined:

- **State Value Function (V(s)):** estimates the expected reward when starting from state s and following a particular policy 'π'.
- **Action Value Function (Q(s,a)):** estimates the expected reward when taking action 'a' in state 's' and then following policy 'π'.

Episode: An episode refers to a sequence of states, actions, and rewards that ends in a terminal state or after a given number of steps.

Markov Decision Processes (MDPs): To apply reinforcement learning algorithms to an environment, it must satisfy the Markov property. According to this property, all subsequent states following a given state are dependent solely on that state and the action taken. Therefore, knowing the current state eliminates the need to consider the previous history of states to make an optimal decision. This property also enables predicting the next state based on the current state and a chosen action.

Exploration vs. Exploitation: Exploration refers to the act of an agent trying out different actions, especially ones it is not certain about, in order to gather more information about its environment. Exploitation refers to the act of an agent selecting the action it believes will yield the highest reward based on its current knowledge. In continuous spaces, the balance between exploration and exploitation becomes crucial. Strategies like Epsilon-greedy , Softmax action selection or adding noise to the actions can help in efficient exploration.

- **Epsilon-greedy:** The agent exploits its current knowledge with probability $1-\epsilon$ and explores with probability ϵ, where ϵ often starts high and decays over time.
- **Softmax action selection:** Actions are chosen probabilistically based on their expected rewards, introducing stochasticity that encourages exploration.
- **Adding noise to actions:** his method involves adding a random noise to the chosen action. This ensures that the action taken by the agent varies slightly from its original intended action, inducing a level of exploration.

1.5. Main Algorithms

This section focuses on the exploration of various reinforcement learning algorithms. The section begins with a discussion of the distinction between on-policy and off-policy algorithms, highlighting the differences in their learning approaches. The section then delves into the comparison between model-based and model-free learning, emphasizing the trade-offs between sample efficiency and model accuracy. The last part of the section is dedicated to algorithms specifically designed for continuous action spaces, presenting a comprehensive overview of several algorithms, including Actor-Critic (AC), Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C), Deep Deterministic Policy Gradients (DDPG), Distributed Distributional Deterministic Policy Gradients (D4PG), and Proximal Policy Optimization (PPO). Each algorithm is discussed in terms of its underlying principles, advantages, and disadvantages, providing a complete understanding of the available options for reinforcement learning in continuous action spaces.

1.5.1. On-policy vs Off-policy

On-policy: Algorithms that learn the value of the policy currently being used for decision-making. They are tightly coupled with the policy that's being optimized.

Off-policy: Algorithms that can learn the value of a policy different from the one currently being used. They decouple the exploration from the policy being learned, which allows for more flexibility.

1.5.2. Model-Based Learning vs Model-Free Learning

Model-based learning: Algorithms that construct a model of the environment (transition and reward functions) and then use this model to decide on the best action. They can be more sample-efficient but may suffer from model inaccuracies.

Model-free learning: Algorithms that directly learn the policy or value function without requiring a model of the environment. These methods can be less efficient but don't suffer from model approximation errors.

1.5.3. Continuous Action Spaces Specific Algorithms

Actor-Critic (AC)

Utilizes both the policy parameterization (actor) and the value function approximation (critic). The critic evaluates the policy's performance while the actor updates the policy in the direction suggested by the critic.

- **Advantages**: Offers a balance between value-based and policy-based methods, potentially reducing variance in learning.
- **Disadvantages**: Requires maintaining and updating two neural networks, which can complicate implementation.

Advantage Actor-Critic (A2C)

Extends AC by subtracting the estimated value of the current state from the Q-value (calculated as the sum of reward and discounted estimated future value), leading to the advantage function: $A(s,a)=Q(s,a)-V(s)$.

- **Advantages**: Reduces variance compared to vanilla AC, provides a clearer signal for policy updates.
- **Disadvantages**: As with AC, it requires maintaining two networks, potentially complicating the implementation and training process.

Asynchronous Advantage Actor-Critic (A3C)

Parallelizes A2C by having multiple agents in different environments, which update a global model asynchronously.

- **Advantages**: Speeds up training, reduces correlation between experiences from different agents, leading to more stable learning.
- **Disadvantages**: Requires careful synchronization and efficient communication between agents and the main model.

Deep Deterministic Policy Gradients (DDPG)

DDPG is an off-policy algorithm specifically designed for continuous action spaces. It combines the advantages of deep Q-learning and policy gradient methods. It uses an actor-critic architecture, where the actor network outputs a deterministic policy. It employs target networks for both the actor and the critic to stabilize training. These target networks are updated slowly to track the learned networks, preventing rapid changes in the Q-values that could destabilize learning. DDPG also incorporates a replay buffer to store and sample past experiences. This mechanism breaks the correlation between consecutive experiences, allowing the algorithm to learn from a diverse set of experiences and ensuring more stable training.

- **Advantages**: Efficiently handles high-dimensional continuous action spaces.
- **Disadvantages**: Sensitive to hyperparameters and may require normalization techniques for stable training.

Distributed Distributional Deterministic Policy Gradients (D4PG)

Enhances DDPG by incorporating distributional value estimation, which maintains a distribution over value estimates instead of a single mean value. Additionally, parallelizes multiple agents in different distributed environments, which update a global model asynchronously.

- **Advantages**: Offers richer value estimates, potentially improving the robustness and stability of training.
- **Disadvantages**: Adds complexity to value estimation, requires careful management of value distributions.

Proximal Policy Optimization (PPO)

Modifies the policy gradient objective to prevent large policy updates, ensuring changes are only made if the new policy is "close" to the old one. This is done using a clipped objective function.

- **Advantages**: Stable and robust training due to conservative policy updates.
- **Disadvantages**: May progress slower in some scenarios due to limited policy updates.

1.6. Reinforcement Learning Applied to the Field of Mobile Robotics

In the realm of mobile robotics, RL provides a framework where robots can learn optimal behaviors through interaction with their environment, adapting to uncertainties and complexities that might be challenging for traditional methods. This section delves into how core components of RL, such as agents, environments, actions, states, and rewards, are mapped in the context of mobile robotics. Subsequently, the section contrasts RL with other established techniques in robotics, like traditional control algorithms and genetic algorithms.

1.6.1. Reinforcement Learning Components Mapped to Mobile Robotics

Agent: The agent in mobile robotics is the intelligence or algorithm governing the robot's behavior. It is not the robot itself, but rather the computational part that decides actions based on observations and learned knowledge.

Environment: This is the space in which the robot operates. It includes physical obstacles, terrain, and other entities the robot might interact with.

Actions: For mobile robots, these can be decisions like move forward, turn, stop, or more complex ones like path planning. Depending on the robot and task, action spaces can be discrete, continuous, or a combination of both. In the case of the autonomous vehicle, this could be the desired linear and angular velocities. For the drone, it might involve the vector describing the three-dimensional trajectory in the next time step.

States: States represent the current situation or configuration of the robot within the environment. However, in many cases, robots may not have a complete observation of the entire environment, leading to a distinction between the complete state of the environment and the robot's observation. In such partially observable environments, the robot's perception is referred to as an "observation", which might be a subset of the complete state.

Rewards: The reward system in mobile robotics is vital. Properly designed rewards can guide a robot towards desired behaviors while avoiding undesired ones. In RL for mobile robotics, reward design often takes into account objectives like reaching a destination, avoiding collisions, and conserving energy.

1.6.2. Comparison and Synergy with Other Techniques

Traditional Control Algorithms: Traditional control algorithms like PID controllers [24] , LQRs (Linear Quadratic Regulators) [25] , and MPC (Model Predictive Control) [26] have been standard tools in mobile robotics for years. These methods generally rely on mathematical models of the system and the environment. While they are robust and well-understood, they might not handle complex environments or unexpected scenarios as effectively as RL. However, the synergistic combination of these methods with RL can provide the reliability of traditional controllers with the adaptability of reinforcement learning. For example, a PID controller might maintain the stability of a drone, while the RL agent learns to navigate in complex environments.

Genetic Algorithms (GAs): GAs are optimization techniques inspired by the process of natural selection. In the context of mobile robotics, GAs might be employed to optimize certain parameters of an agent's operation. While GAs are excellent for optimization, RL focuses on learning through interaction. When combined, GAs might determine optimal hyperparameters for an RL agent, while the agent learns the best policy through interaction with the environment.

1.7. Project Objectives

This section outlines the primary goals and motivations behind this thesis. At the core of this project is the ambition to design and implement autonomous navigation and obstacle avoidance behaviors for two distinct types of vehicles: a differential vehicle and a drone, using the principles of reinforcement learning. The objectives are not just limited to the design and simulation phases but extend to the real-world application, emphasizing the crucial simulation-to-reality knowledge transfer.

1.7.1. Justification for the Selection of the Differential Vehicle and the Drone

Versatility and Complexity: The differential vehicle and drone represent two distinct types of mobility - ground and air respectively. Studying both provides a comprehensive understanding of autonomous navigation challenges in various terrains and dimensions.

Real-world Relevance: Both drones and differential vehicles are gaining prominence in multiple industries, from logistics and transport to surveillance and agriculture. This makes them pertinent subjects for autonomous research.

Diverse Set of Challenges: The differential vehicle, with its constraints on ground navigation, offers challenges related to terrain navigation, obstacle avoidance, etc. In contrast, drones present challenges in three-dimensional space navigation, collision avoidance with both ground and airborne objects, and stability amidst varying wind conditions.

1.7.2. Specific Objectives of the Project

Development of Autonomous Behaviors: To design and develop autonomous navigation behaviors for both a differential vehicle and a drone using reinforcement learning.

Simulation-to-Real Transition: To validate the trained models in a simulated environment and then successfully transition and adapt them for real-world deployment.

Integration of Technologies: Seamless integration of technologies such as Unity, Simulink, ROS2, Docker, and reinforcement learning frameworks to achieve a holistic system that can function in both simulated and real-world settings.

1.7.3. Expected Contributions to the Field

Methodological Contributions: By seamlessly integrating multiple technologies and platforms, this project aims to provide a replicable methodology for others in the field wishing to undertake similar interdisciplinary projects.

Addressing the Simulation-to-Real Gap: A successful transition from simulation to the real-world would validate the methods used and provide insights to others facing challenges in this area, contributing a valuable case study to the field.

CHAPTER 2: SIMULATION METHODOLOGY

In the realm of autonomous systems, simulation serves as the foundation upon which real-world implementations are built. This chapter delves into the technologies and tools employed to simulate and train the autonomous differential vehicle and drone.

The following sections offer a breakdown of the system as a whole, starting with the definition of the differential vehicle and the drone as distinct systems. The deep reinforcement learning algorithm used, the Distributed Distributional Deep Deterministic Policy Gradient (D4PG) is then discussed.

The chapter then transitions into the modeling and implementation of the environments, distinguishing between the Unity and Python sides, and contrasting the simplified environment with the more realistic one implemented in MATLAB and Simulink.

The chapter concludes by analyzing the evaluation criteria for the experiments, which serve as a crucial component of the simulation methodology, providing a systematic approach to assess the performance of the autonomous differential vehicle and drone.

2.1. Description of the Technologies

Simulation Software

Unity [27] : Chosen for its versatility in 3D simulations, Unity provided the platform for sensor and actuator implementations. It offers an intuitive interface, advanced physics simulations, and real-time visualization.

Simulink [28] : While Unity serves as a primary interface for visualization and high-level dynamics, Simulink complements it by offering realistic physics simulations, especially for the control aspect. Simulink provides a robust platform for implementing low-level controls seamlessly.

Technologies and Tools

ROS2 [29] : The Robot Operating System, version 2, facilitates the communication between different application modules, acting as a middleware. Given the complexity of autonomous systems where various modules and components need to communicate in real-time, ROS2 becomes an indispensable tool.

Docker [30] : To ensure the developed application's reproducibility and compatibility across various platforms, Docker has been used for containerization. Docker encapsulates the application and its dependencies into a 'container' that can run uniformly on any machine, which is critical for collaborative projects and real-world deployment.

Reinforcement Learning Frameworks

PyTorch [31] with low-level D4PG implementation [32] : PyTorch was the framework of choice due to its dynamic computational capacities and intuitive design which makes custom implementations, like D4PG, more straightforward and customizable. The D4PG (Distributed Distributional Deterministic Policy

Gradients) algorithm, given its nature, fits well with complex tasks like autonomous navigation which requires both continuous action spaces and distributional value estimates.

Programming Languages

Python [33] : Given its simplicity and wide array of support libraries, Python was used primarily for reinforcement learning, data analysis, and general scripting.

C#: As Unity's primary scripting language, C# was employed for implementing the logic within the 3D simulation environment.

MATLAB and Simulink: Primarily for control system design and low-level dynamics simulations.

Version Control Systems

Git [24] : Git was the choice for the version control software. It not only allows for tracking changes and collaboration but also ensures that multiple versions of the project can be managed seamlessly.

2.2. System Description

In this section, the system description of both the differential vehicle and the drone is presented, including all the components that constitute their respective architectures: from the mathematical models that replicate their physical designs to the control mechanisms that govern their movements, as well as their sensors and actuators.

Also, the deep reinforcement learning algorithm used in this project is introduced, highlighting its suitability for environments with continuous action spaces. The Deep Deterministic Policy Gradient (DDPG) and its advanced counterpart, the Distributed Distributional Deep Deterministic Policy Gradient (D4PG), are discussed in detail.

2.2.1. Definition of the Differential Vehicle and the Drone as Systems

Differential Vehicle System

The differential vehicle system follows a conventional architecture. As seen from the simulation perspective, it consists of two primary elements:

Mathematical Model: The mathematical model replicates the physical design of the vehicle, which features two motorized wheels at the back that deliver the required torque for movement. Additionally, a stabilizing wheel at the front ensures three contact points, and therefore stability.

Control Mechanism: The vehicle's motion is governed by two conventional PID controllers, one for linear velocity (with a nominal velocity of 0.4 m/s) and the other for angular velocity (with a nominal value of π rad/s). This dual PID control structure facilitates precise control over both speed and direction.

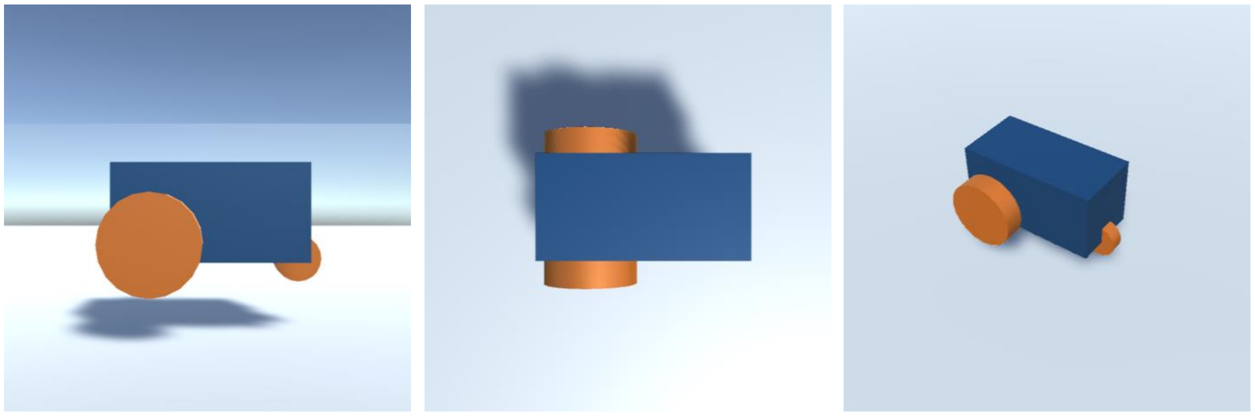


Figure 2: Differential Vehicle Simulation Model

Drone System

The simulation of the drone follows the design of a quadrotor in a '+' structure. The primary components of the drone are:

Mathematical Model: The mathematical model of the drone is carefully designed to emulate a quadrotor's dynamics, with smooth movements ensuring that pitch and roll angles remain close to zero. This simplification facilitates ease of control by considering the system as holonomic, allowing for more intuitive control over the vehicle.

Control Mechanism: The control system for the drone is more intricate, cascading three distinct control loops:

- **Angle Control Loop:** A low-level control loop responsible for maintaining the drone's orientation.
- **Linear Velocity Control Loop:** This loop controls the linear velocities of the drone, allowing for controlled movements in various directions.
- **Position Control Loop:** The top-level control loop utilizes an LQR (Linear Quadratic Regulator) architecture to manage the three-dimensional positioning of the drone.

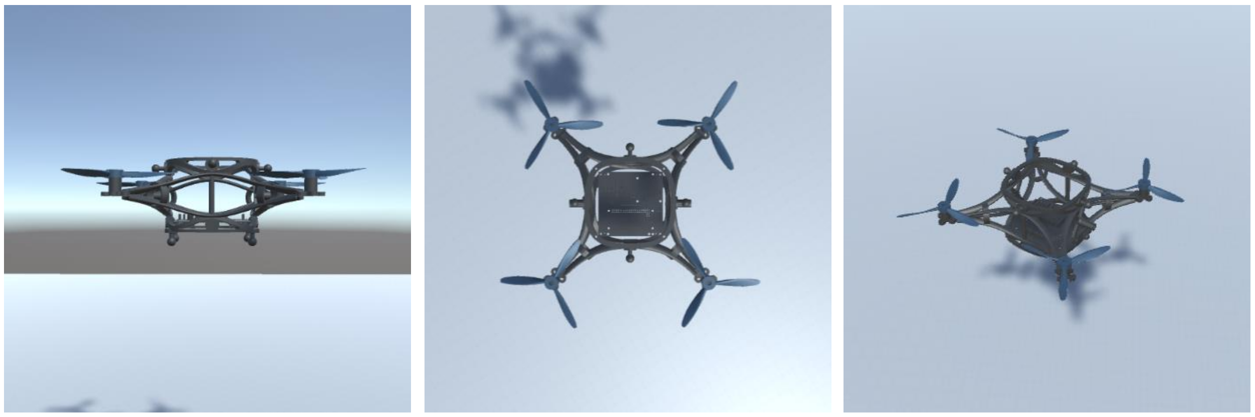


Figure 3: Drone Simulation Model

<u>Sensors</u>

Motion Capture Cameras: These cameras are utilized to obtain accurate positioning data, including both position and orientation. The cameras track specific markers or features within the environment, and this information is translated into a spatial context.

Inertial Measurement Unit (IMU): The IMU integrates accelerometers and gyroscopes to measure the vehicle's specific force and angular rate. This aids in stability and navigation, providing data that is crucial for controlling the movement and equilibrium of both the differential vehicle and the drone.

LiDAR: LiDAR sensors emit laser beams to measure distances to objects within their range. In the context of this project, a 2D rotatory LiDAR is used for both the vehicle and drone, providing a two-dimensional "slice" of the surrounding environment. This technology is vital for obstacle detection and avoidance.

<u>Actuators</u>

Differential Vehicle System: The actuators for the differential vehicle system are the two motorized wheels located at the back. By individually controlling the speed and direction of each motorized wheel, the differential vehicle can achieve both forward motion and turning actions.

Drone System: For the quadrotor drone system, the actuators are its four rotors. Each rotor is powered by a dedicated motor that provides thrust. Two diagonally opposite rotors spin in one direction and the other two in the opposite direction. The collective thrust from all rotors allows the drone to ascend or descend, while the differential thrust between them enables controlled movements in pitch, roll, and yaw. Specifically:

- To move vertically, all rotors increase or decrease their speed uniformly.
- To pitch, the front and back rotors vary their speeds relative to each other.
- To roll, the left and right rotors vary their speeds relative to each other.
- For yaw, the rotors on opposite ends adjust their speeds in opposite directions.

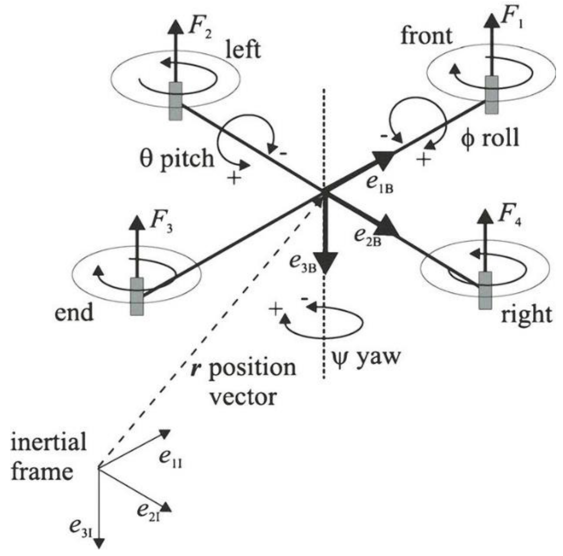


Figure 4: Drone Reference Frame and Rotation Angles
Image Source: [Conceptual Diagram of a Quadcopter | Download Scientific Diagram (researchgate.net)]

2.2.2. Definition of the Deep Reinforcement Learning Algorithm

The complexity of continuous action spaces and the need for stable convergence require the use of advanced reinforcement learning algorithms. The Deep Deterministic Policy Gradient (DDPG) is designed explicitly for environments with continuous action spaces, making it suitable for real-world problems like these autonomous systems. However, while DDPG offers a robust solution, advancements in the field have led to the development of an even more refined algorithm: the Distributed Distributional Deep Deterministic Policy Gradient (D4PG). D4PG not only inherits the strengths of DDPG but also introduces several enhancements that make it particularly well-suited for this project's requirements. The following sections will present the details of DDPG and subsequently explore the improvements introduced by D4PG.

Deep Deterministic Policy Gradient (DDPG)

DDPG [35] is an algorithm specifically designed for environments with continuous action spaces, making it particularly suitable for many real-world problems, such as robotics and autonomous systems. It is an off-policy algorithm and an adaptation of the standard Deep Q-Learning (DQN) to work with continuous actions.

Actor-Critic Architecture

- **Actor:** The actor defines the policy of the agent and therefore is responsible for determining the best action given a particular state. The actor takes the current state as input and outputs a continuous action or a set of continuous actions in multi-dimensional action spaces.
- **Critic:** The critic evaluates the action taken by the actor based on the current state and provides a value estimate (Q-value). This value estimate is used to update both the actor and the critic.

The combination of these two networks allows for stable learning. The critic helps in directing the actor to choose better actions by providing feedback.

Off-Policy Learning

DDPG is an off-policy algorithm, meaning it learns from past experiences stored in a replay buffer. This buffer stores tuples of experiences (state, action, reward, next state). During training, random samples are drawn from this buffer to update the networks, decoupling the correlation between consecutive experiences and stabilizing the learning process.

Deterministic Policy Gradient

Unlike traditional policy gradient methods that work with stochastic policies, DDPG uses a deterministic policy gradient. This means that for any given state, the actor outputs a specific action without any randomness. The deterministic policy gradient theorem ensures convergence and stability in learning.

Exploration vs. Exploitation

Since DDPG is designed for continuous action spaces, traditional exploration methods like epsilon-greedy can't be applied. Instead, DDPG adds noise to the policy, typically Ornstein-Uhlenbeck noise is used due to its temporally correlated nature.

Target Networks and Soft Updates

DDPG employs the concept of target networks, borrowed from DQN, to stabilize learning. There are target versions of both the actor and critic networks.

Instead of copying the weights directly from the main networks to the target networks, DDPG uses "soft updates." This means the target network weights are a blend of the main network weights and their own, ensuring smooth transitions and further stabilizing the learning.

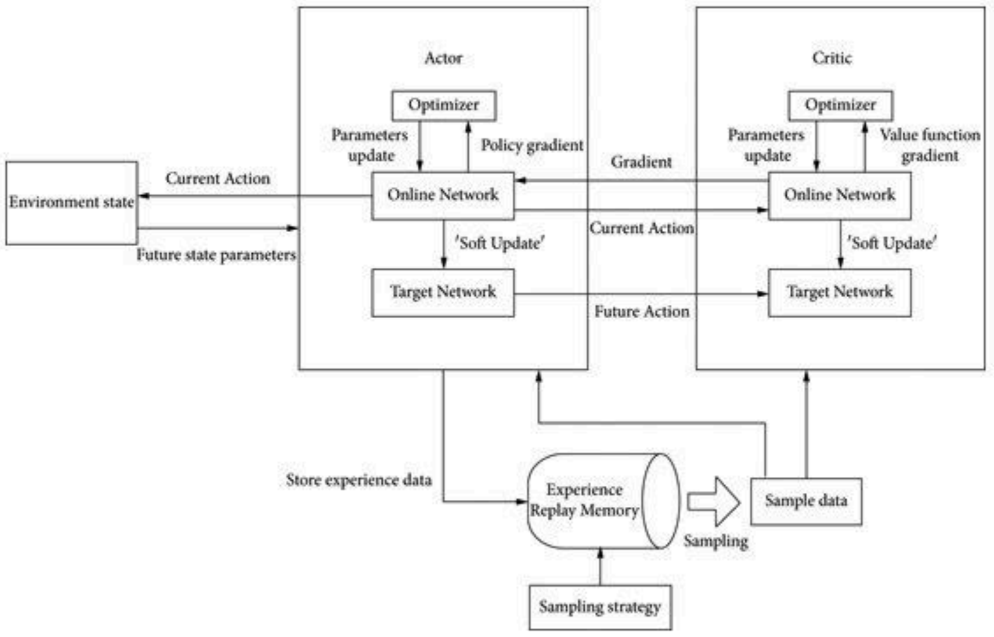


Figure 5: DDPG Block Diagram

Image Source: The structure of DDPG algorithm. | Download Scientific Diagram (researchgate.net)

Distributed Distributional Deep Deterministic Policy Gradient (D4PG)

D4PG [36] introduces several improvements over the standard DDPG algorithm. These include distributional updates to the DDPG algorithm, combined with the use of multiple distributed workers all writing into the same replay table. The algorithm also considers other smaller changes, all of which contribute to its overall performance.

Distributed Experience Gathering

D4PG modifies the standard training procedure to distribute the process of gathering experience. Multiple actors operate in parallel, all contributing to a centralized replay table. A learner process then samples from this replay table, ensuring efficient and diverse data for network updates.

Distributional Perspective

D4PG adopts the distributional perspective on reinforcement learning, a paradigm shift from the traditional approach of estimating the expected value of future rewards to modeling the entire distribution of these rewards. This means that for a given state-action pair, instead of a single value estimate, a distribution over all possible returns is maintained. By modeling the full distribution, this technique captures the inherent randomness and uncertainty in the environment's dynamics and the agent's interactions with it. This can produce better gradients, leading to better and more stable learning signals, and resulting in improved learning performance.

N-step Returns

In traditional Temporal Difference (TD) learning an agent updates its value estimate based on the immediate reward and the estimated value of the next state. This is known as a 1-step return because it considers only the immediate reward. N-step returns extend this idea by considering a sequence of N rewards into the future, rather than just the immediate reward. This approach allows for a more accurate and efficient estimation of the TD error, contributing significantly to the overall performance of the algorithm.

Prioritized Experience Replay

Prioritized experience replay introduces the idea of prioritizing certain experiences (state, action, reward, next state) based on their learning potential, often measured by the magnitude of their TD error. Experiences with higher TD errors are given higher priority when sampling from the replay buffer.

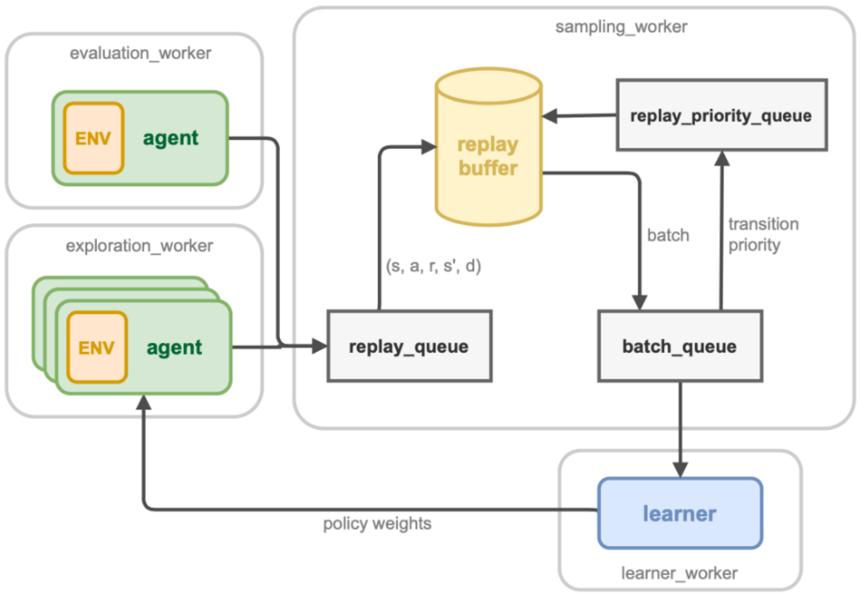


Figure 6: D4PG Block Diagram

Image Source: [d4pg-pytorch/README.md at master · schatty/d4pg-pytorch (github.com)]

2.3. Modeling of the Environments

This section delves into the process of modeling the environments, ensuring they are both representative of real-world scenarios and computationally efficient for training purposes.

Initially, the section provides a comprehensive breakdown of the Unity side of the environment, detailing the foundational classes and components that constitute the simulation. This includes the environment class, various sensor and actuator classes, controls, and the procedural generation of obstacles. Subsequently, the Python side of the environment is explored, emphasizing the communication and interaction mechanisms with the Unity environment.

The modeling of the environments is further divided into two implementations: a simplified environment and a realistic environment. The simplified environment serves as a proof of concept, allowing for quick testing and validation of the fundamental ideas. It is designed with lower computational demands, enabling parallelization. The realistic environment builds upon the simplified environment, incorporating more complex dynamics and control systems to simulate a more realistic scenario. It integrates with an existing system implemented in Matlab and Simulink, providing a more accurate representation of the vehicle's dynamics and control systems.

Through this detailed modeling, the environments aim to provide a robust platform for training autonomous differential vehicles and drones using reinforcement learning.

2.3.1. Components of the Environment

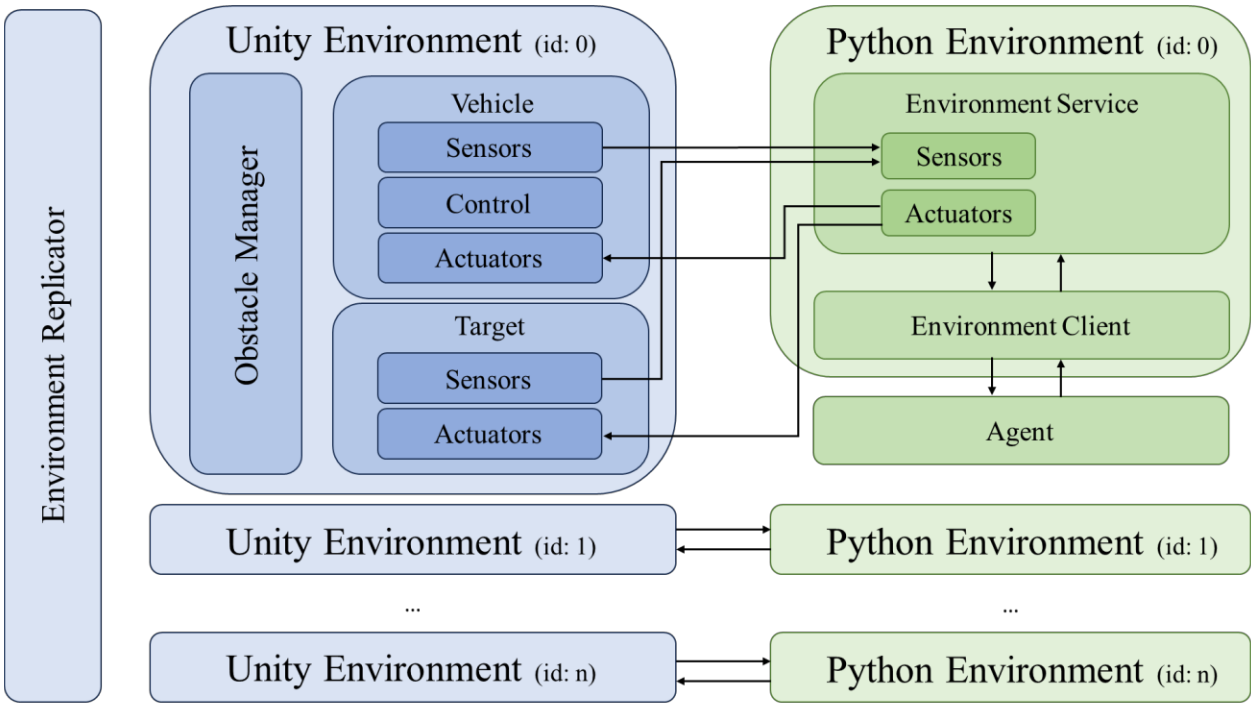


Figure 7: System Implementation Block Diagram

2.3.1.1. *Unity Side of the Environment*

2.3.1.1.1. Environment and Environment Replicator

Environment Class

The Environment class in Unity represents a specific scenario in which the differential vehicle or drone operates. It serves as a foundational element upon which other components, such as sensors, actuators, and controls, are built.

Attributes:

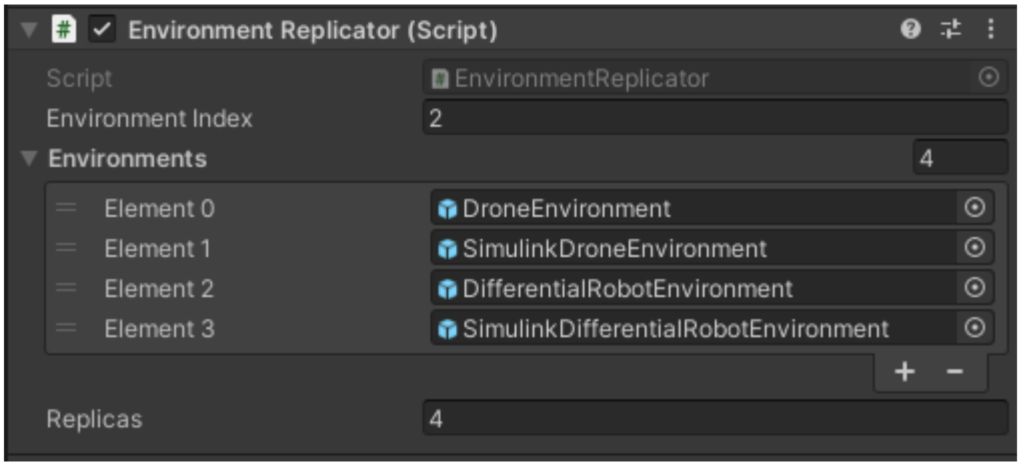
- **environmentID:** A unique identifier for each instance of the environment. This ensures that each environment can be distinctly recognized when multiple instances are created.
- **ROS:** A connection to the Robot Operating System (ROS), facilitating communication between the Unity simulation and other components including the reinforcement learning agent.
- **distance, width, height:** These attributes define the physical dimensions of the environment. They ensure that when multiple environments are instantiated side by side, they don't overlap or interfere with each other.

Methods:

- **SetupROSConnection():** Initializes the ROS connection for the environment. This connection is vital for real-time communication and data exchange between the simulation and external components.

Environment Replicator Class

The EnvironmentReplicator class is responsible for creating multiple parallelized instances of a given environment. The environment acts as a container GameObject. Within this container, other GameObjects reside, which allocate the scripts responsible for sensors, actuators, and controls. When the EnvironmentReplicator creates replicas of the environment, it inherently replicates all the contained logic, ensuring that each environment operates independently and in parallel.



Environment Replicator (Script)

Script	EnvironmentReplicator
Environment Index	2
Environments	4
Element 0	DroneEnvironment
Element 1	SimulinkDroneEnvironment
Element 2	DifferentialRobotEnvironment
Element 3	SimulinkDifferentialRobotEnvironment
Replicas	4

Figure 8: Environment Replicator Class Settings

Attributes:

- **environmentIndex:** Determines which specific environment from the available list will be replicated.
- **environments:** A list of potential environments (as GameObjects) that can be instantiated. These include simplified and realistic environment variations for both the differential vehicle and drone.
- **replicas:** Specifies the number of instances to be created for the chosen environment.

Methods:

- **OnEnable():** Activated when the script is initialized, this method handles the replication process. It creates the specified number of environment replicas, ensuring each is positioned adjacently based on the distance attribute to prevent overlap. Each replica is also assigned a unique environmentID and is named accordingly. This ID plays a pivotal role in ensuring that nodes and topics within ROS are uniquely identified, preventing any communication conflicts. The ROS connection for each environment is also established using a default IP address and port, as per the Unity ROS documentation.

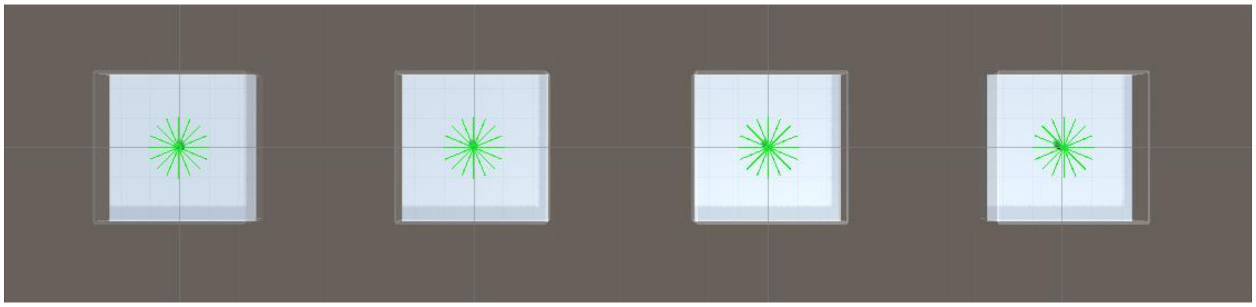


Figure 9: Environment Replicator Simulation Visualization

2.3.1.1.2. Sensors

<u>Sensor Class</u>

The Sensor class represents a generic sensor component within the Unity environment. It is designed to be versatile, allowing for the creation of various sensor types by extending this base class. The sensor communicates with the Robot Operating System (ROS) to publish its data and can be reset based on messages received from ROS.

Attributes:

- **ROS:** An instance of the ROSConnection, which facilitates communication with the Robot Operating System.
- **environment:** Reference to the parent Environment object.
- **tag:** A string tag to identify the type of sensor.
- **environmentID:** A string identifier for the environment in which the sensor resides.
- **sensorID:** A unique identifier for the sensor.
- **sensorName:** A combination of the tag, environmentID, and sensorID to create a unique name for the sensor.

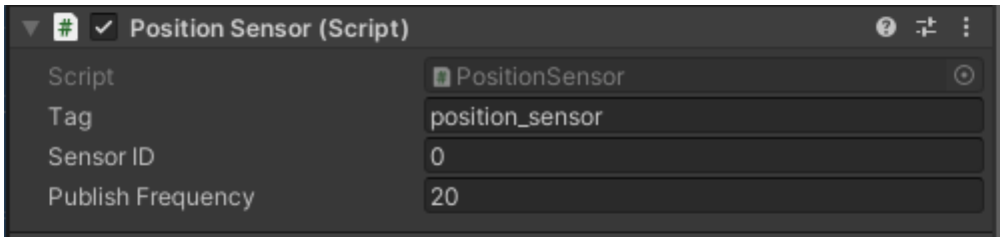
- **topicName:** The ROS topic on which the sensor data will be published.
- **resetTopicName:** The ROS topic on which reset messages will be received.
- **publishFrequency:** The frequency at which the sensor data will be published to ROS.
- **publishPeriod:** The time interval between successive data publications.
- **timeSinceLastPublished:** A timer to keep track of the time since the last data was published.

Methods:

- **Start():** Initializes the sensor by setting up its attributes, registering it as a publisher in ROS, and subscribing to the reset topic.
- **Update():** Checks if it is time to publish new sensor data and does so if the condition is met.
- **GetSensorData():** A virtual method meant to be overridden by child classes to provide the actual sensor data.
- **ResetSensor(BoolMsg reset):** A virtual method meant to be overridden by child classes to reset the sensor based on a received message.

<u>Position Sensor Class</u>

The PositionSensor class is a derived class from the generic Sensor class. It specifically captures and publishes the position data of an object within the Unity environment. This data is then converted into a ROS-compatible message. The position is adjusted to account for the difference between Unity's coordinate system (where Y is up) and the typical ROS coordinate system (where Z is up).



Position Sensor (Script)

Script	PositionSensor
Tag	position_sensor
Sensor ID	0
Publish Frequency	20

Figure 10: Position Sensor Class Settings

Methods:

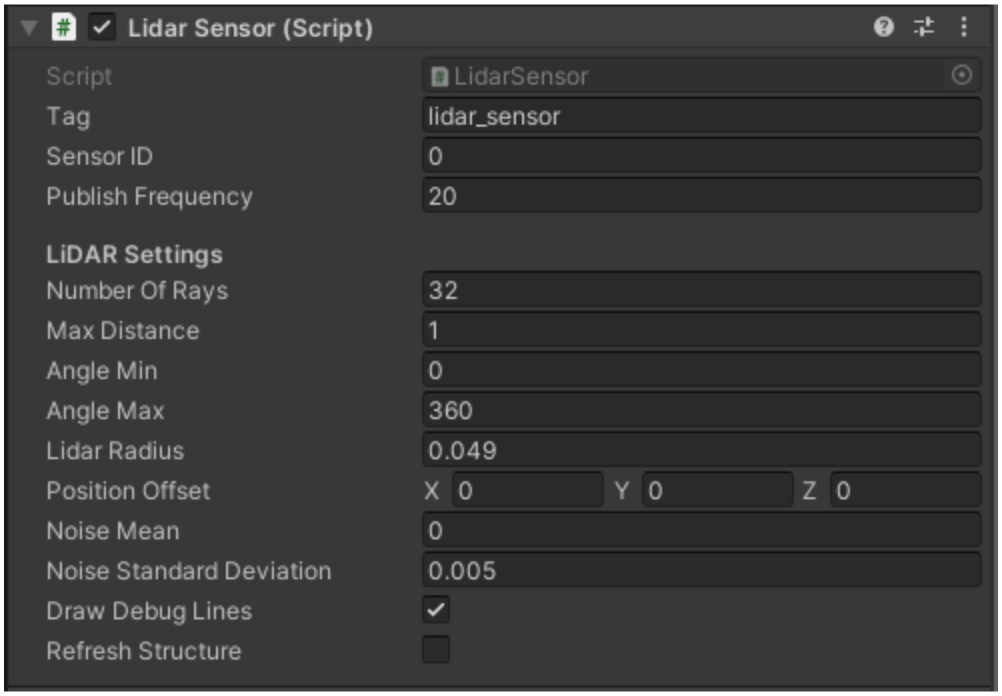
- **Start():** Calls the base class's Start method to initialize the sensor.
- **GetSensorData():** Overrides the base class's method to provide the position data. The method captures the current position of the object (relative to its environment) and returns it in the form of a PointStampedMsg, which is a ROS message type for stamped position data.

LiDAR Sensor Class

The LidarSensor class represents a LiDAR (Light Detection and Ranging) sensor in the Unity environment. This sensor emits multiple rays and measures the distance to the nearest object for each ray. The class is designed to simulate the behavior of a real-world LiDAR sensor.

The LiDAR sensor emits a specified number of rays (numberOfRays) within a defined angular range (angleMin to angleMax). For each ray, the sensor checks if it collides with any object in the environment. If a collision is detected, the distance to the object is measured and stored. This distance can also have Gaussian noise added to it, simulating real-world inaccuracies. The results of these measurements are then packaged into a LaserScanMsg object, which can be communicated to other systems, such as ROS.

In the Unity editor, the rays can be visualized as lines. Rays that hit an object are colored red (hitColor), while rays that don't hit anything are colored green (noHitColor).



Script	LidarSensor
Tag	lidar_sensor
Sensor ID	0
Publish Frequency	20
LiDAR Settings	
Number Of Rays	32
Max Distance	1
Angle Min	0
Angle Max	360
Lidar Radius	0.049
Position Offset	X 0 Y 0 Z 0
Noise Mean	0
Noise Standard Deviation	0.005
Draw Debug Lines	✓
Refresh Structure	

Figure 11: LiDAR Sensor Class Settings

Attributes:

- **numberOfRays:** The total number of rays emitted by the LiDAR.
- **maxDistance:** The maximum distance a ray can travel before it is considered to not have hit anything.
- **angleMin** and **angleMax:** Define the minimum and maximum angles for the emitted rays.
- **lidarRadius:** The radius of the LiDAR, used to determine the starting point of each ray.
- **positionOffset:** The offset of the LiDAR sensor from the object's origin.
- **noiseMean** and **noiseStandardDeviation:** Parameters to introduce Gaussian noise to the LiDAR measurements.
- **drawDebugLines:** A boolean to determine if the rays should be visualized in the Unity editor.

- **refreshStructure:** A boolean to determine if the LiDAR's structure (origins, directions, etc.) should be recalculated.
- **origins, directions, distances,** and **hits:** Arrays that store the starting point, direction, measured distance, and whether each ray hit an object, respectively.
- **hitColor** and **noHitColor:** Colors used to visualize rays that hit an object and rays that didn't, respectively.

Methods:

- **Start():** Initializes the LiDAR sensor, setting up its structure.
- **Update():** Updates the LiDAR sensor each frame. It redraws the debug lines if required.
- **GetSensorData():** Generates and returns a LaserScanMsg object which contains the data from the LiDAR sensor. This method emits rays, checks for collisions, introduces noise to the measurements, and populates the LaserScanMsg object with the measured distances and other relevant data.
- **RefreshStructure():** Recalculates the origins, directions, and default distances for all rays based on the current settings. This method is useful when the LiDAR's settings are changed during runtime.

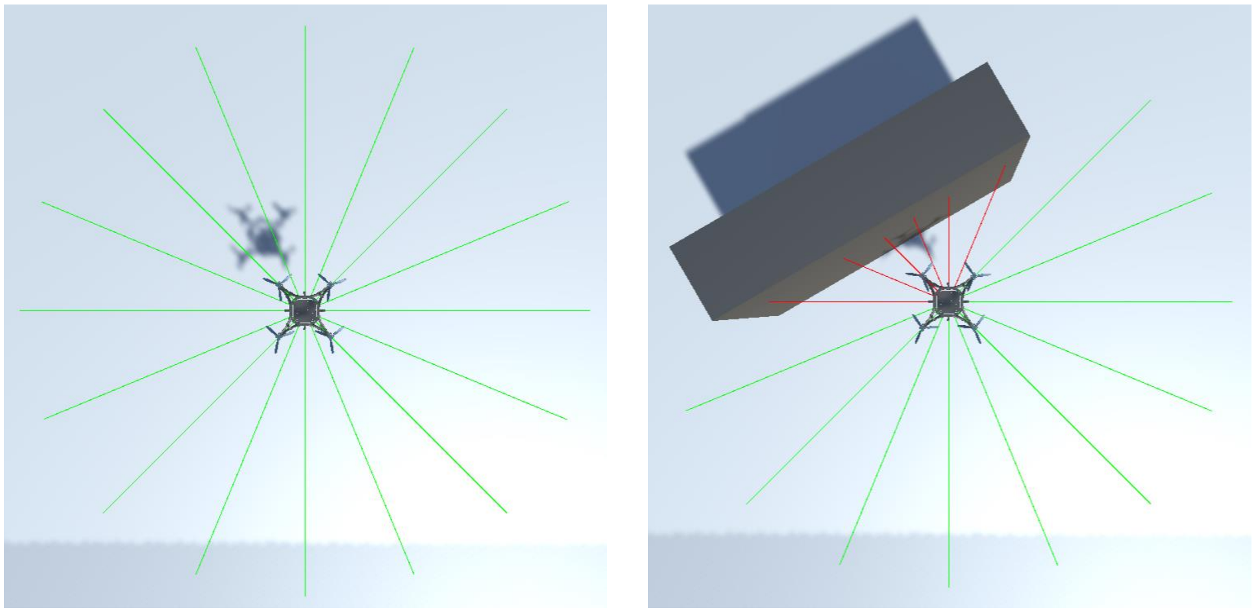
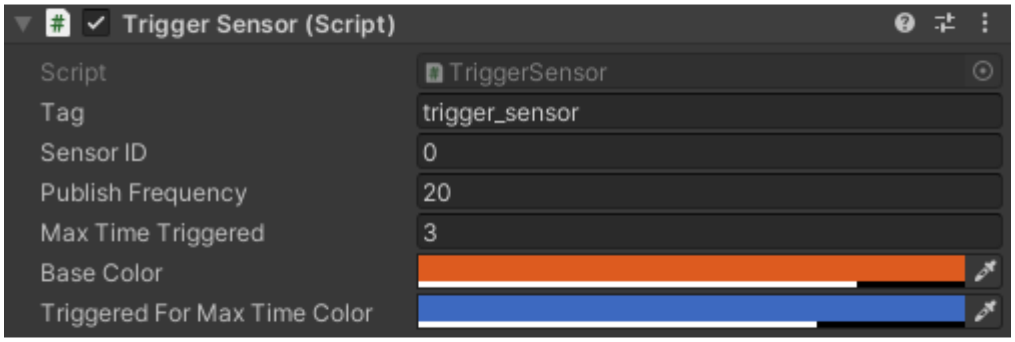


Figure 12: LiDAR Sensor Simulation Visualization

Trigger Sensor Class

The TriggerSensor class represents a sensor in the Unity environment that detects when an object enters or exits its trigger zone. It extends the base Sensor class, which is parameterized with a BoolMsg type. The sensor changes its color based on the time it has been triggered and sends a message when it has been triggered for a specified maximum time.

Note: The TriggerSensor class requires a Collider component (with isTrigger enabled) to be attached to the same GameObject. This ensures that the sensor can detect when objects enter or exit its trigger zone.



Trigger Sensor (Script)

Script	TriggerSensor
Tag	trigger_sensor
Sensor ID	0
Publish Frequency	20
Max Time Triggered	3
Base Color	🟧
Triggered For Max Time Color	🟦

Figure 13: Trigger Sensor Class Settings

Attributes:

- **hasTriggered:** A boolean that indicates whether an object has entered the sensor's trigger zone.
- **hasTriggeredForMaxTime:** A boolean that indicates if the sensor has been triggered for a duration exceeding maxTimeTriggered.
- **timeTriggered:** A float that keeps track of the total time the sensor has been triggered.
- **maxTimeTriggered:** A public float that specifies the maximum time for which the sensor can be triggered before hasTriggeredForMaxTime is set to true.
- **material:** A reference to the material of the GameObject this script is attached to. This is used to change the color of the sensor.
- **baseColor:** The original color of the sensor.
- **triggeredForMaxTimeColor:** The color the sensor will change to when it has been triggered for the maximum time.

Methods:

- **Start():** Initializes the sensor's attributes. It sets the initial values for hasTriggered, hasTriggeredForMaxTime, and timeTriggered. It also gets the material of the GameObject to which this script is attached.
- **Update():** This method updates every frame. It increments the timeTriggered if the sensor is currently triggered. It also checks if the sensor has been triggered for more than the maxTimeTriggered and updates the hasTriggeredForMaxTime accordingly. The color of the sensor is also interpolated between the baseColor and triggeredForMaxTimeColor based on the ratio of timeTriggered to maxTimeTriggered.
- **GetSensorData():** Overrides the base class method. It creates a new BoolMsg object and sets its data attribute to the value of hasTriggeredForMaxTime. This message is then returned.
- **OnTriggerEnter(Collider other):** A Unity callback method that is called when another object enters the trigger zone of this sensor. It sets hasTriggered to true.
- **OnTriggerExit(Collider other):** A Unity callback method that is called when another object exits the trigger zone of this sensor. It sets hasTriggered to false.
- **ResetSensor(BoolMsg reset):** Overrides the base class method. It resets the values of hasTriggered, hasTriggeredForMaxTime, and timeTriggered to their initial states.

2.3.1.1.3. Actuators

Actuator Class

The Actuator class serves as a generic base class for actuators within the Unity environment. It is designed to be versatile, allowing for the creation of various actuator types by extending this base class. The actuator communicates with the Robot Operating System (ROS) to receive actuation commands and can be reset based on messages received from ROS.

Attributes:

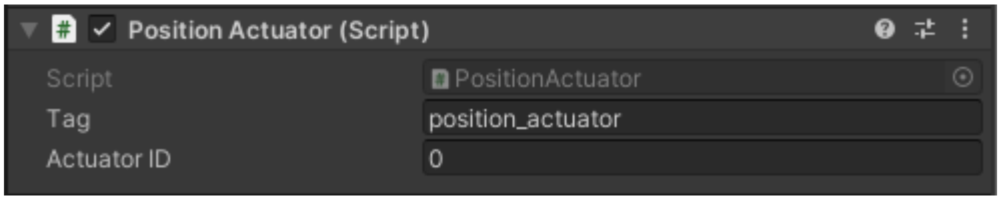
- **ROS:** An instance of the ROSConnection, which facilitates communication with the Robot Operating System.
- **environment:** Reference to the parent Environment object.
- **tag:** A string tag to identify the type of actuator.
- **environmentID:** A string identifier for the environment in which the actuator resides.
- **actuatorID:** A unique identifier for the actuator.
- **actuatorName:** A combination of the tag, environmentID, and actuatorID to create a unique name for the actuator.
- **topicName:** The ROS topic on which the actuator will receive commands.
- **resetTopicName:** The ROS topic on which reset messages will be received.

Methods:

- **Start():** Initializes the actuator by setting up its attributes, registering it as a subscriber in ROS for both actuation commands and reset messages.
- **ProcessActuatorData(T data):** A virtual method meant to be overridden by child classes to process the actuation command received from ROS.
- **ResetActuator(BoolMsg reset):** A virtual method meant to be overridden by child classes to reset the actuator based on a received message.

Position Actuator Class

The PositionActuator class is a specific type of actuator that controls the position of an object within the Unity environment. It extends the generic Actuator class and is designed to receive position commands from ROS and adjust the object's position accordingly.


Position Actuator (Script)

Script	PositionActuator
Tag	position_actuator
Actuator ID	0

Figure 14: Position Actuator Class Settings

Attributes:

- **position:** A Vector3 variable that stores the desired position for the object.

Methods:

- **Start():** Initializes the PositionActuator by calling the base Start() method from the parent Actuator class.
- **ProcessActuatorData(PointStampedMsg data):** Overrides the base method to process the position command received from ROS. It converts the received PointStampedMsg data into a Vector3 position and adjusts the object's position within the Unity environment accordingly.

2.3.1.1.4. Controls

<u>Control Class</u>

The Control class serves as a base class for all controls within the Unity environment. It is responsible for managing the communication with the Robot Operating System (ROS) to receive control references and reset commands. The control references are stored in a Float32MultiArrayMsg format, which is a ROS message type that contains an array of float values.

Attributes:

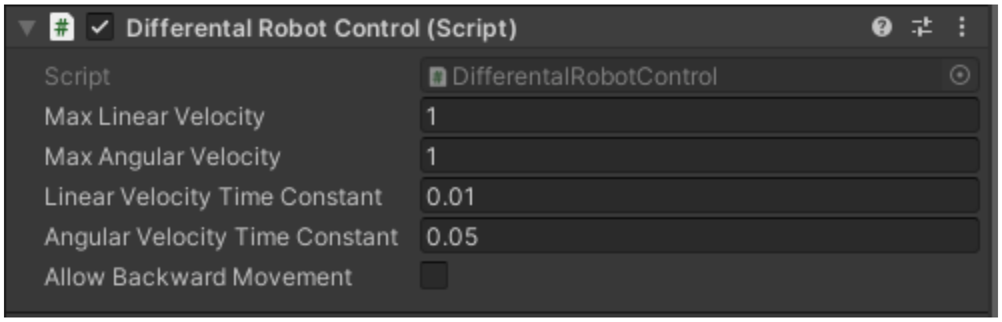
- **ROS:** An instance of the ROSConnection, which facilitates communication with the Robot Operating System.
- **environment:** Reference to the parent Environment object.
- **environmentID:** A string identifier for the environment in which the control resides.
- **controlName:** A unique name for the control, derived from the environmentID.
- **referenceTopicName:** The ROS topic on which control references will be received.
- **resetTopicName:** The ROS topic on which reset messages will be received.
- **reference:** A ROS message type that stores the received control references.

Methods:

- **Start():** Initializes the control by setting up its attributes, subscribing to the reference and reset topics in ROS, and initializing the reference object.
- **FixedUpdate():** Calls the ProcessControlReference method in each fixed frame update.
- **StoreControlReference(Float32MultiArrayMsg data):** Stores the received control reference data.
- **ProcessControlReference():** A virtual method meant to be overridden by child classes to process the received control references.
- **ResetControl(BoolMsg reset):** A virtual method meant to be overridden by child classes to reset the control based on a received message.

Differential Robot Control Class

The DifferentialRobotControl class is a derived class from the Control class, specifically tailored for controlling a differential robot. It includes attributes and methods for managing the robot's linear and angular velocities, with support for exponential smoothing and optional backward movement.


Differental Robot Control (Script)

Script	DifferentalRobotControl
Max Linear Velocity	1
Max Angular Velocity	1
Linear Velocity Time Constant	0.01
Angular Velocity Time Constant	0.05
Allow Backward Movement	☐

Figure 15: Differential Robot Control Class Settings

Attributes:

- **maxLinearVelocity:** The maximum allowable linear velocity.
- **maxAngularVelocity:** The maximum allowable angular velocity.
- **linearVelocityTimeConstant:** A time constant used for exponential smoothing of linear velocity.
- **angularVelocityTimeConstant:** A time constant used for exponential smoothing of angular velocity.
- **allowBackwardMovement:** A boolean flag to determine if backward movement is allowed.
- **robotRigidbody:** A reference to the robot's Rigidbody component.
- **targetLinearVelocity:** The target linear velocity calculated from the reference data.
- **targetAngularVelocity:** The target angular velocity calculated from the reference data.

Methods:

- **Start():** Initializes the differential robot control by calling the base class's Start method and getting the Rigidbody component.
- **ProcessControlReference():** Overrides the base class method to process the control reference data, applying exponential smoothing and considering the max velocities and the allowBackwardMovement flag. It updates the robot's linear and angular velocities accordingly.
- **ResetControl(BoolMsg reset):** Overrides the base class method to reset the robot's linear and angular velocities to zero.

Position Control Class

The PositionControl class is derived from the base Control class and is specifically designed to control the position of an object within the Unity environment. It processes control references from ROS to determine a target position and then smoothly transitions the object to this target position over time.

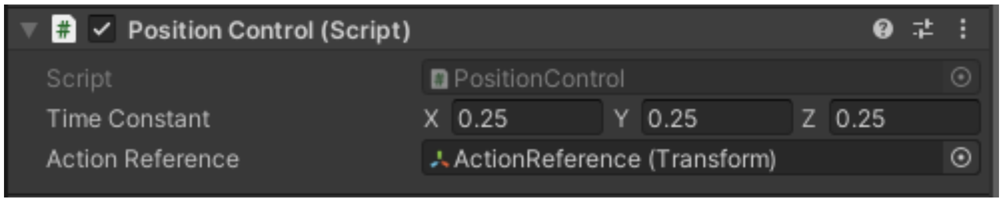

Position Control (Script)
Script: PositionControl
Time Constant: X 0.25 Y 0.25 Z 0.25
Action Reference: ActionReference (Transform)

Figure 16: Position Control Class Settings

Attributes:

- **timeConstant:** A Vector3 representing the time constants for exponential smoothing in the x, y, and z directions.
- **actionReference:** A Transform object that serves as a reference for the target position.
- **currentPosition:** A Vector3 representing the current position of the object.
- **targetPosition:** A Vector3 representing the desired target position based on the control reference.

Methods:

- **Start():** Initializes the position control by calling the base class's Start method and setting the currentPosition to the object's initial position.
- **ProcessControlReference():** Overrides the base class method to process the control reference data. It calculates the target position based on the current position and the received reference data. Exponential smoothing is applied to transition the object smoothly to the target position in each of the x, y, and z directions. The actionReference's position is then updated to the calculated target position.

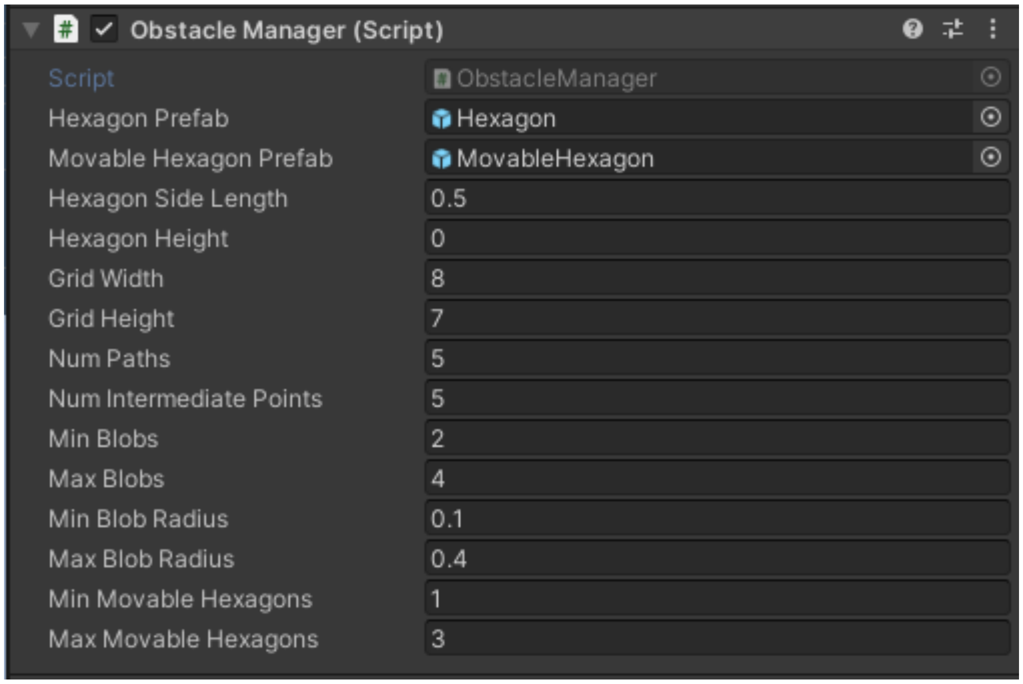
2.3.1.1.5. Procedural Obstacle Generation

Obstacle Manager Class

The ObstacleManager class is designed to manage and generate obstacles within a simulated environment. The primary objective of this class is to ensure that the agent does not overfit to a specific environment configuration. By introducing variability and randomness in the environment, especially in the placement and type of obstacles, the agent is encouraged to learn more generalized strategies that are robust to changes.

This class is responsible for:

1. Generating a grid of hexagonal obstacles (static obstacles) in the environment.
2. Ensuring there are a minimum number of paths through the obstacles, promoting a convex problem where the agent can confidently find a solution.
3. Introducing movable obstacles, adding an extra layer of complexity and unpredictability to the environment.



Obstacle Manager (Script)

Script	ObstacleManager
Hexagon Prefab	Hexagon
Movable Hexagon Prefab	MovableHexagon
Hexagon Side Length	0.5
Hexagon Height	0
Grid Width	8
Grid Height	7
Num Paths	5
Num Intermediate Points	5
Min Blobs	2
Max Blobs	4
Min Blob Radius	0.1
Max Blob Radius	0.4
Min Movable Hexagons	1
Max Movable Hexagons	3

Figure 17: Obstacle Manager Class Settings

Attributes:

- **hexagonPrefab** and **movableHexagonPrefab:** Prefabs for the static and movable hexagonal obstacles.
- **hexagonSideLength** and **hexagonHeight:** Dimensions of the hexagons.
- **gridWidth** and **gridHeight:** Dimensions of the grid on which hexagons are placed.
- **numPaths** and **numIntermediatePoints:** Parameters to ensure a minimum number of paths through the obstacles.
- **minBlobs**, **maxBlobs**, **minBlobRadius** and **maxBlobRadius:** Parameters for random blob-shaped areas where obstacles are disabled.
- **minMovableHexagons** and **maxMovableHexagons:** Range for the number of movable hexagons to be introduced in the environment.
- **hexagons** and **movableHexagons:** Lists to store the instantiated static and movable hexagons.

Methods:

- **InstantiateObstacles():** This method is responsible for generating the hexagonal grid map. It calculates the positions of each hexagon based on the grid dimensions and places them in the environment.
- **CheckAndBuildPath():** Ensures that there are a minimum number of paths through the obstacles. It activates the hexagons, clears paths from the start to the target position, disables random blobs of obstacles, destroys any existing movable hexagons, and spawns new movable hexagons.
- **ClearPath():** Clears a path through the hexagons. It ensures that hexagons near the start and target positions are disabled and then clears multiple paths from the start to the target, ensuring the problem remains convex.
- **DisableRandomBlobs():** Disables random blob-shaped areas of hexagons in the environment. The number of blobs and their radii are determined randomly within the specified ranges.

- **SpawnMovableHexagons():** Introduces movable hexagons in the environment. It first identifies hexagons that are currently disabled, then randomly selects a subset of them and replaces them with movable hexagons.
- **DestroyMovableHexagons():** Destroys all movable hexagons in the environment, clearing the way for new ones to be spawned in the next iteration.

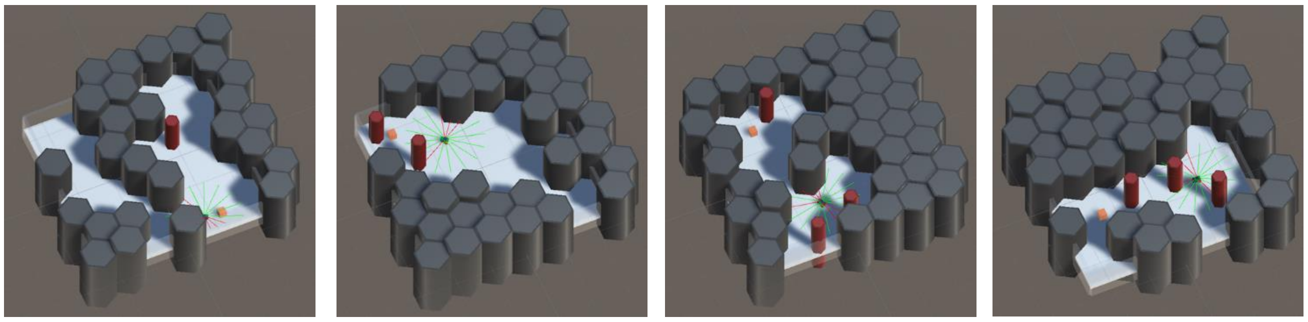


Figure 18: Obstacle Manager Simulation Visualization

<u>Movable Hexagon Class</u>

The MovableHexagon class represents a movable obstacle in the simulated environment. It is designed to move randomly within the environment. It calculates possible directions based on hexagonal geometry and checks if a move in a chosen direction is valid. The hexagon continues to move until it reaches its target position, after which it chooses a new random direction. The movement is constrained by the environment's bounds and proximity to other active hexagons.

Attributes:

- **obstacleManager:** A reference to the ObstacleManager class, which manages and generates obstacles in the environment.
- **targetPosition:** The position the hexagon is moving towards.
- **isMoving:** A boolean flag indicating if the hexagon is currently moving.
- **moveSpeed:** The speed at which the hexagon moves.

Methods:

- **Initialize(ObstacleManager obstacleManager):** Initializes the MovableHexagon with a reference to the ObstacleManager.
- **Update():** Unity's built-in method that gets called every frame. It checks if the hexagon is moving and updates its position. If it is not moving, it triggers the hexagon to move in a random direction.
- **GetRandomDirection():** Calculates a random direction for the hexagon to move in. The direction is based on the hexagonal geometry and the side length of the hexagons.
- **MoveToRandomDirection():** Determines a random direction and checks if the move in that direction is valid. If valid, it initiates the move.
- **IsValidMove(Vector3 newPosition):** Checks if the proposed move is valid. It ensures the move doesn't take the hexagon outside the environment bounds and that the new position isn't too close to any active hexagon.
- **MoveTo(Vector3 newPosition):** Initiates the move of the hexagon to the specified position.

2.3.1.2. Python Side of the Environment

Sensor Class

The Sensor class in the Python side represents a generic sensor component that communicates with the Unity environment through ROS2. This class subscribes to a ROS2 topic to receive data from the Unity environment and provides methods to handle this data.

Attributes:

- **tag:** A string tag to identify the type of sensor.
- **environment_id:** A string identifier for the environment in which the sensor resides.
- **sensor_id:** A unique identifier for the sensor.
- **name:** A combination of the tag, environment_id, and sensor_id to create a unique name for the sensor.
- **topic_name:** The ROS topic on which the sensor data will be received.
- **msg_type:** The type of ROS message that the sensor expects to receive.
- **data:** The latest data received from the Unity environment.
- **subscription:** A ROS subscription object that listens to the specified topic for incoming data.

Methods:

- **__init__(self, tag: str, environment_id: int, sensor_id: int, msg_type):** The constructor initializes the sensor with its unique identifiers, sets up the ROS subscription, and prepares the sensor to receive data.
- **subscriber_callback(self, msg):** This method is called whenever new data is received on the subscribed topic. It updates the data attribute with the latest received data.
- **reset(self):** Resets the data attribute to None, effectively clearing the last received data.
- **log(self, msg: str, log_type: int = 0):** A utility method to print log messages with different log levels (Info, Debug, Warning, Error). The method prints the log message prefixed with the log type and the sensor's name.

Actuator Class

The Actuator class in the Python side represents a generic actuator component that communicates with the Unity environment through ROS2. This class publishes actuation commands to a ROS2 topic, which are then processed by the Unity environment to control the actuators.

Attributes:

- **tag:** A string tag to identify the type of actuator.
- **environment_id:** A string identifier for the environment in which the actuator resides.
- **actuator_id:** A unique identifier for the actuator.
- **name:** A combination of the tag, environment_id, and actuator_id to create a unique name for the actuator.
- **topic_name:** The ROS topic on which the actuator will send commands.
- **data:** The latest actuation command to be sent to the Unity environment.
- **publisher:** A ROS publisher object that sends data to the specified topic.

Methods:

- **__init__(self, tag: str, environment_id: int, actuator_id: int, msg_type):** The constructor initializes the actuator with its unique identifiers, sets up the ROS publisher, and prepares the actuator to send commands.
- **publish_data(self, data):** This method sends the provided actuation command to the Unity environment through the ROS topic. It updates the data attribute with the latest command and then publishes it.
- **reset(self):** Resets the data attribute to None, effectively clearing the last actuation command.
- **log(self, msg: str, log_type: int = 0):** A utility method to print log messages with different log levels (Info, Debug, Warning, Error). The method prints the log message prefixed with the log type and the actuator's name.

EnvironmentService Class

The EnvironmentService class represents the environment's service side, which interacts with both sensors and actuators. It provides services to step through the environment, reset it, and publish references.

Attributes:

- **environment_id:** A string identifier for the environment.
- **name:** A unique name for the environment service.
- **reference_topic_name:** The ROS topic on which the environment will publish references.
- **step_service_name:** The ROS service through which the environment will receive step requests.
- **reset_topic_name:** The ROS topic on which the environment will receive reset messages.
- **sample_time:** The time interval between successive steps.
- **sensors_list:** A list of sensors associated with the environment.
- **actuators_list:** A list of actuators associated with the environment.
- **node:** A ROS node associated with the environment service.

Methods:

- **start(self):** Initializes the environment service, adds the associated node, sensors, and actuators, and starts spinning the service.
- **publish_reference(self, reference):** Publishes a given reference to the reference topic.
- **step_callback(self, request, response):** Handles a step request, publishes the action as a reference, waits for the sample time, and then returns the state, reward, and done status.
- **reset_callback(self, msg):** Resets all sensors and actuators associated with the environment and then calls the reset method.
- **state(self), reward(self, action), done(self), and reset(self):** Abstract methods that need to be implemented by subclasses to define the environment's behavior.
- **log(self, msg: str, log_type: int = 0):** A utility method to print log messages with different log levels.

EnvironmentClient Class

The EnvironmentClient class represents the environment's client side, which allows the agent to interact with the environment service to step through and reset the environment.

Attributes:

- **environment_id:** A string identifier for the environment.
- **name:** A unique name for the environment client.
- **step_service_name:** The ROS service through which the client will send step requests to the environment.
- **reset_topic_name:** The ROS topic on which the client will send reset messages.
- **step_client:** A ROS client for the step service.
- **reset_publisher:** A ROS publisher for the reset topic.
- **reset_wait_time:** The time to wait after sending a reset message.

Methods:

- **step(self, action):** Sends a step request with a given action to the environment service and returns the resulting state, reward, and done status.
- **reset(self):** Sends a reset message to the environment service, waits for the reset_wait_time, and then sends a step request with an empty action to get the initial state.
- **log(self, msg: str, log_type: int = 0):** A utility method to print log messages with different log levels.

2.3.2. Simplified Environment Implementation

Starting with a simplified environment is beneficial for several reasons:

Proof of Concept: Before diving into a complex system, it is essential to ensure that the basic concept works. A simplified environment allows for quick testing and validation of the fundamental ideas.

Parallelization: Simplified environments can be parallelized with lower computational demands, meaning multiple instances can run simultaneously. This is especially beneficial in reinforcement learning where multiple agents can explore the environment concurrently, specifically for the algorithm used (D4PG). Complex environments, especially those designed in tools like Matlab and Simulink, have higher computational costs.

Ease of Debugging: In a simplified environment, it is easier to identify and rectify issues. As the complexity grows, so does the difficulty in identifying problems.

Smooth Transition: Once the basic system is understood and validated, complexities can be introduced incrementally. This step-by-step approach ensures a smooth transition from a simple to a complex model.

EnvService Class

The EnvService class implementations represent the specific environments for a differential vehicle and drone in a reinforcement learning context. They provide a structured environment for an agent to learn how to navigate a differential vehicle or drone towards a target while avoiding obstacles.

Methods:

- **state(self):** This method returns the current state of the environment, which is an observation that the RL agent uses to decide its next action. The state includes the relative position of the target from the vehicle's coordinate system, the readings from the LiDAR sensor, and the vehicle's orientation. This way, the state provides the agent with information about its surroundings and its relative position to the target.
- **reward(self, action):** This method computes the reward the agent receives after taking an action in the current state. The reward is computed based on several factors and then clipped to be within a predefined range to ensure stability in learning:
    - **Distance Reduction**: The agent is rewarded for reducing the distance to the target.
    - **Collision Detection**: If the LiDAR sensor detects that the vehicle is too close to an obstacle, a collision penalty is applied.
    - **Reaching the Target**: A maximum reward is given if the vehicle reaches the target.
- **done(self):** This method determines if the current episode is done. An episode can end for several reasons:
    - The vehicle collides with an obstacle.
    - The vehicle reaches the target (with the dimensional tolerance and a minimum time spent in the target zone, given by the Trigger Sensor).
    - The maximum allowed time for an episode is exceeded.
- **reset(self):** This method resets the environment to its initial state to start a new episode. The positions of the vehicle and the target are randomized, ensuring that the agent faces different scenarios in each episode and learns to generalize its policy.

Differences Between the Differential Vehicle and Drone Environments

The environments for a differential vehicle and a drone, while sharing most elements, exhibit distinct characteristics that reflect the unique dynamics and constraints of each system. These differences manifest in various aspects of the implementation:

State Dimensions:

- **Differential Vehicle:** The state is defined by the relative position (2D) and LiDAR readings.
- **Drone:** The state is defined by the relative position (3D, including height), the drone's height, LiDAR readings, and the previous action.

Unrotate Vector Function: This function is used for transforming vectors from a global reference frame to the vehicle-specific reference frame.

- **Differential Vehicle:** The function unrotate_vector only considers the yaw angle.
- **Drone:** The function unrotate_vector considers roll, pitch, and yaw angles. It uses rotation matrices for all three angles to compute the transpose of the total rotation matrix.

Reward Calculation:

- **Differential Vehicle:** The reward calculation considers distance reduction, collision detection, reaching the target, and yaw rate penalty.
- **Drone:** The reward calculation considers distance reduction, collision detection (including height-based collisions), and reaching the target. The yaw rate penalty is not present in the drone environment.

Collision Detection:

- **Differential Vehicle:** Collision is detected based on the LiDAR readings.
- **Drone:** Collision is detected based on LiDAR readings and the drone's height (both minimum and maximum height thresholds).

<u>Environment Simplifications</u>

In order to focus on the core dynamics of the system the following simplifications were adopted:

Simplified first-order transients: This means that the system's transient responses are modeled as first-order systems. In essence, the system's dynamics are represented with a single time constant, ignoring potential complexities like overshoots or oscillations that might be present in higher-order systems.

No noise in the positioning sensor measurements: This implies that the position sensors provide perfect measurements without any noise. In real-world scenarios, sensors have some level of noise, which can affect the accuracy of readings and the performance of control algorithms.

Low noise in the LiDAR readings: While the position sensors are considered noise-free, the LiDAR readings have low noise. This means that while the LiDAR readings are mostly accurate, there's a small amount of uncertainty or error in the measurements, which is closer to real-world scenarios but still idealized.

From the simplified environment, one can expect to learn:

- The basic dynamics of the system.
- How the agent (vehicle or drone) behaves in response to various actions.
- Initial validation of the algorithms and logic that will be used in the more complex environment.

2.3.3. Realistic Environment Implementation

The realistic environment implementation builds upon the simplified environment, incorporating more complex dynamics and control systems to simulate a more realistic scenario for the differential vehicle and drone. This section describes the implementation details of the realistic environment, highlighting the modifications made to communicate with an already existing system [37] and the incorporation of more realistic features.

Integration with the Existing System

The realistic environment implementation required modifications to communicate with an already existing system, which was implemented in Matlab and Simulink. The original system covers the dynamics of the vehicles and the control systems, providing a more accurate representation of the real-world behaviors. However, the whole implementation of the system was out of the scope of this project, and the focus was on integrating the existing system with the reinforcement learning environment.

The communication between the previously implemented simplified environment in Python and the existing system in Matlab and Simulink needed to be bidirectional. To achieve this integration, the Simulink ROS library was employed, leading to the development of three primary subsystems:

Action Subscriber: This subsystem receives the reference values from the reinforcement learning agent, processes them, and introduces them into the control logic of the existing system. This allows the agent to influence the behavior of the differential vehicle and drone by providing reference values for the control systems.

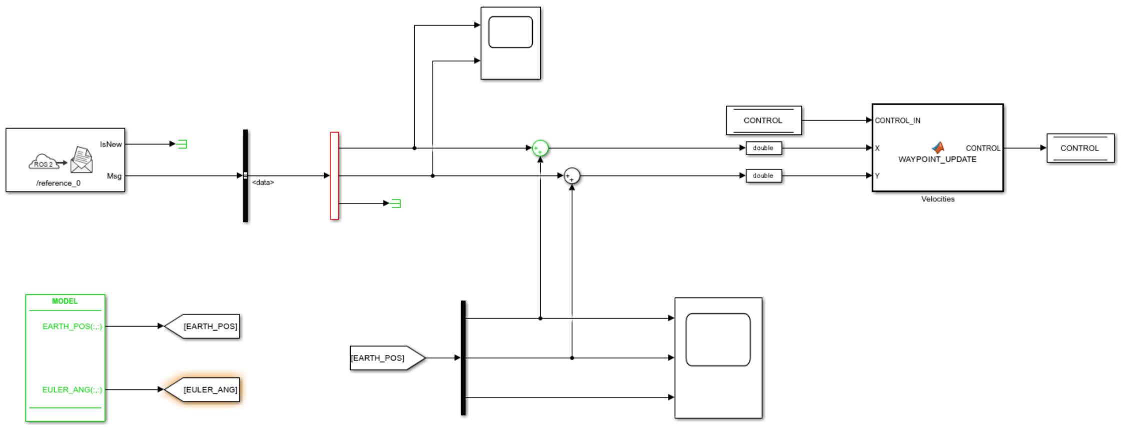


Figure 19: Action Subscriber Simulink Block Diagram

Dynamics Publisher: This subsystem feeds from the vehicle model's buses in the existing system and publishes the information about the vehicle's dynamics to the Python environment. The Python environment processes this information and includes it in the observations provided to the reinforcement learning agent. This allows the agent to receive feedback on the effects of its actions on the vehicle's dynamics.

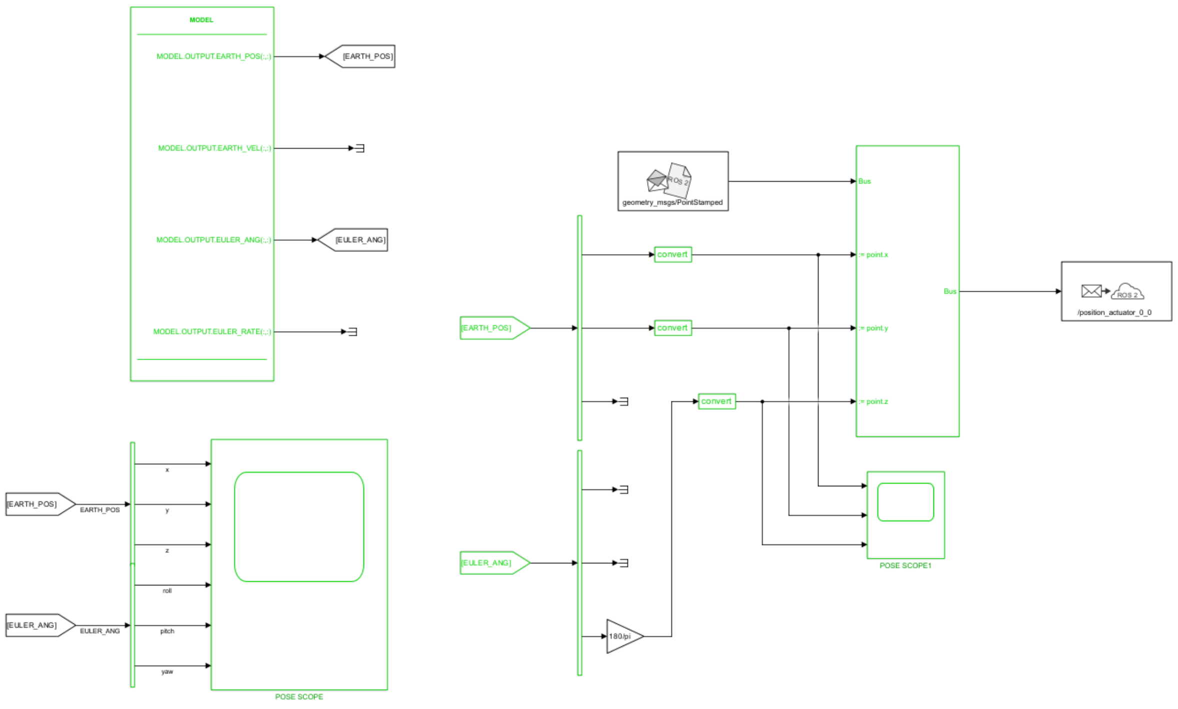


Figure 20: Dynamics Publisher Simulink Block Diagram

Done Subscriber: This subsystem receives the reset signal from the Python environment and resets the entire Simulink system. This allows the Python environment to start a new episode by resetting the Simulink system to its initial state.

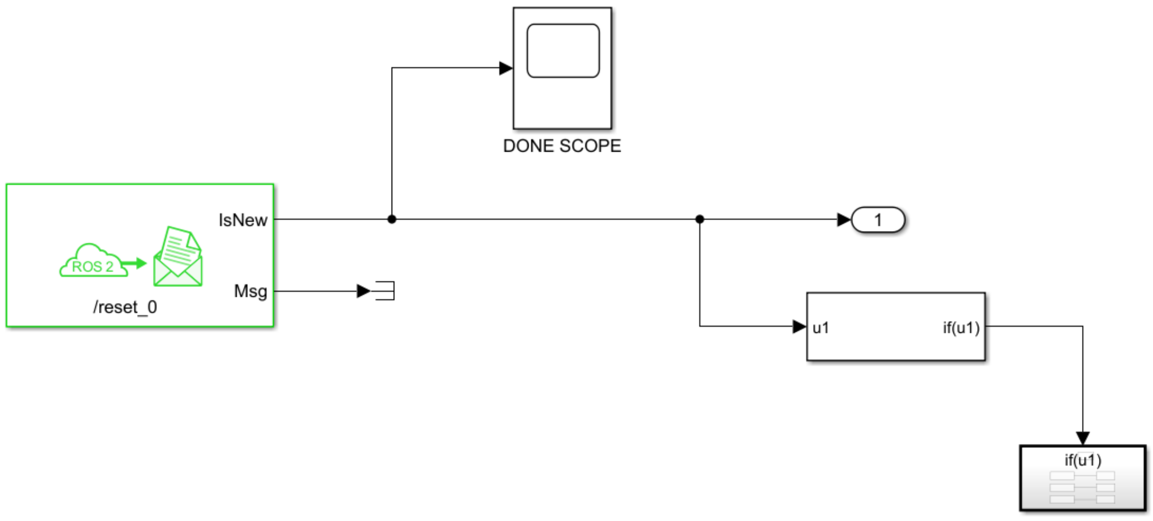


Figure 21: Done Subscriber Simulink Block Diagram

Enhancements and Features in the Realistic Environment

This realistic implementation incorporates several features that make it more representative of real-world scenarios:

Model and Control Accuracy: The existing system in Matlab and Simulink provides a more accurate representation of the vehicle's dynamics and control systems. This results in more realistic transients, allowing the reinforcement learning agent to learn in an environment that closely resembles real-world conditions.

Noise in Positioning Sensor Measurements: Unlike the simplified environment, the realistic environment adds noise to the positioning sensor measurements. This introduces uncertainty in the position readings, which is a common challenge in real-world scenarios. The reinforcement learning agent must learn to handle this uncertainty and make decisions based on noisy observations.

High Noise in LiDAR Measurements: The realistic environment also introduces high noise in the LiDAR measurements. This adds another layer of complexity to the environment, as the agent must learn to navigate and avoid obstacles based on noisy sensor readings.

2.4. Programming of the Agents

This section delves into the core components of the D4PG implementation, including the central engine file that orchestrates the training process, the agent and learner files that define the behavior and learning routines, the neural network architectures essential for policy and value estimation, and a utilities folder housing vital data structures and tools.

2.4.1. Engine File

The engine file is the central hub for the training process of the reinforcement learning agent. It manages the coordination between various components, such as the agent, the learner, and the replay buffer.

Functions

- **sampler_worker(config, replay_queue, batch_queue, replay_priorities_queue, training_on)**:This function is responsible for managing the replay buffer. It transfers experiences (replays) from a queue to the global replay buffer and samples batches of experiences from the replay buffer and places them in a batch queue for the learner to use.
- **learner_worker(config, training_on, policy, target_policy_net, learner_w_queue, replay_priority_queue, batch_queue, update_step):** This function initializes and runs the learner. The learner is responsible for updating the neural network policy based on experiences from the replay buffer. It receives batches of experiences from the batch queue and uses them to update the policy.
- **agent_worker(config, policy, learner_w_queue, global_episode, i, agent_type, training_on, replay_queue):** This function initializes and runs an agent. The agent interacts with the environment, collects experiences, and places them in the replay queue. Depending on its type ("exploitation" or "exploration"), the agent might use the current policy or explore new actions.

Engine Class

The main class that orchestrates the training process. It initializes various data structures and processes, such as the replay queue, batch queue, and multiprocessing values.

Methods:

- **train():**
  1. Initializes the replay buffer and sets up a separate process (sampler_worker) to manage it.
  2. Sets up the learner process (learner_worker) responsible for training the neural network.
  3. Initializes and runs multiple agent processes (agent_worker). One of these agents is for exploitation, while the rest are for exploration.
  4. All processes are started and then joined, ensuring they complete their tasks.

2.4.2. Agent File

The Agent file is responsible for defining the behavior of the agent, its interaction with the environment, and the methods for updating its policy. It also manages the agent's exploration and exploitation strategies.

Agent Class

The Agent class initializes the agent with a given configuration, policy, and type (exploration or exploitation). It also sets up the environment for the agent and defines methods for the agent's operation.

Methods:

- **init(self, config, policy, global_episode, n_agent=0, agent_type='exploration')**: Initializes the agent with the given configuration, policy, and type (either "exploration" or "exploitation"). Sets up Ornstein-Uhlenbeck noise for action exploration. Finally, sets up a directory for saving experiment results.
- **update_actor_learner(self, learner_w_queue, training_on):** Updates the local actor's parameters with those from the learner. If training is ongoing, it gets the latest parameters from the learner's queue and updates the local actor's parameters accordingly.
- **run(self, training_on, replay_queue, learner_w_queue):** This method is the main loop where the agent interacts with the environment. The agent initializes an experience buffer and then starts its interaction loop. In each episode:
  1. Resets the environment and the noise process.
  2. Selects an action based on its current state. If the agent is of type "exploration", it adds noise to the action for exploration purposes.
  3. Takes a step in the environment using the selected action and observes the next state, reward, and whether the episode is done.
  4. Stores the experience in its buffer. Once the buffer has enough experiences for N-step returns, it computes the discounted reward over N steps and adds the N-step experience to the global replay queue (only if the agent is of type "exploration").
  5. If the episode ends or reaches the maximum number of steps, the agent processes and adds any remaining experiences in its buffer to the replay queue.
  6. If the agent is of type "exploitation", it saves its policy if there is a significant improvement in reward.

7. If the agent is of type "exploration" and has completed a specified number of episodes, it updates its actor using the update_actor_learner method.

- **save_models(self)**: Saves the current state of the actor (policy) to a file. The saved model can be used later for policy evaluation or further training.

2.4.3. Learner File

The learner file is responsible for updating the policy and value networks based on experiences from the replay buffer. It uses the Adam optimizer for both the policy and value networks and employs the Binary Cross Entropy Loss for the value network. The learner also handles the prioritized replay mechanism and updates the target networks using soft updates.

Learner Class

The Learner Class handles the policy and value network update routines.

Methods:

- **__init__(self, config, policy_net, target_policy_net, learner_w_queue):** Initializes the learner with the given configuration, policy network, target policy network, and a queue for the learner worker.
- **_update_step(self, batch, replay_priority_queue, update_step):** Handles the update steps for both the critic (value network) and the actor (policy network). It computes the loss for both networks, updates the priorities in the replay buffer if prioritized replay is enabled, and performs the soft updates on the target networks.
- **run(self, training_on, batch_queue, replay_priority_queue, update_step):** The main loop for the learner. It fetches batches of experiences from the batch queue and calls the _update_step method to update the networks. It also sends the updated policy network parameters to the learner worker queue every 100 steps and prints training progress.

2.4.4. Networks File

The networks file is responsible for defining the neural network architectures that will be used by the reinforcement learning agent. It contains the definitions for both the Value Network (Critic) and the Policy Network (Actor). The Value Network is designed to estimate the Q-values given states and actions, while the Policy Network determines the action to be taken given a state.

ValueNetwork Class

The ValueNetwork class, also referred to as the Critic, is designed to estimate the Q-value of given states and actions. The architecture consists of three linear layers. The input to this network is a concatenation of the state and action, which then passes through two hidden layers with ELU (Exponential Linear Unit) activation functions. The ELU activation function was chosen due to its ability to handle the vanishing gradient problem, allowing for faster learning and convergence. The final layer outputs a distribution over a set of discrete values, representing the possible Q-values. These discrete values, termed as atoms, are uniformly spaced between a minimum and maximum value (v_min and v_max).

Methods:

- **__init__(self, num_states, num_actions, hidden_size, v_min, v_max, num_atoms, device='cuda'):** Initializes the ValueNetwork with the given parameters. It sets up the linear layers and the distribution of Q-values (atoms).
- **forward(self, state, action):** Computes the forward pass of the network, taking in a state and action, and returning the Q-value distribution.
- **get_probs(self, state, action):** Returns the softmax probabilities of the Q-value distribution for a given state and action.

PolicyNetwork Class

The PolicyNetwork class, also known as the Actor, is responsible for determining the action to take given a state. This network consists of three linear layers. The input is the state, which passes through two hidden layers with ELU activation functions. The output layer has a size equal to the number of actions and uses a tanh activation function, since the actions are continuous and bounded between -1 and 1.

Methods:

- **__init__(self, num_states, num_actions, hidden_size, device='cuda'):** Initializes the PolicyNetwork with the given parameters. It sets up the linear layers.
- **forward(self, state):** Computes the forward pass of the network, taking in a state and returning the action value.
- **to(self, device):** Overrides the default to method to ensure that the device attribute is also updated when moving the network to a different device.
- **get_action(self, state):** Converts a numpy state to a PyTorch tensor, computes the forward pass to get the action value, and returns the action.

2.4.5. Utils Folder

The Utils folder serves as a repository for essential utilities and data structures that facilitate the reinforcement learning process:

- **ReplayBuffer Class:** This class provides a basic structure to store and sample experiences.
- **PrioritizedReplayBuffer Class:** This class is an extension of the basic ReplayBuffer but includes prioritization of experiences.
- **OUNoise Class:** This class implements the Ornstein-Uhlenbeck process, which is used to add noise to the actions produced by the agent. This helps in exploration during the training process.
- **SegmentTree Class:**_The Segment Tree file provides data structures that allow for efficient computation of sum and minimum over segments of arrays. It is particularly useful in the context of the Prioritized Replay Buffer.
- **MinSegmentTree Class:** This class extends the SegmentTree to specifically handle minimum operations.

2.5. Evaluation Criteria

2.5.1. Evaluation Metrics

In order to assess the performance of the D4PG agent, in both the simplified and realistic simulation environments, it is crucial to establish clear evaluation criteria. These criteria will not only provide insights into the agents' learning progress but also ensure that the agents' behaviors align with the project's objectives. Below are the defined criteria and metrics:

Simplified Environment (Differential Vehicle and Drone):

Mean Step Reward: This metric measures the average reward obtained by the agent at each time step during the training process. A higher mean step reward indicates that the agent is making decisions that align well with the desired outcomes in the environment. Monitoring this metric helps understanding the agent's learning progress and its ability to maximize rewards over time.

Loss Function: The loss function measures the difference between the predicted Q-values and the target Q-values. Monitoring the loss function will provide insights into the convergence of the D4PG algorithm. A decreasing loss trend suggests that the agent's predictions are becoming more accurate over time.

Simplified and Realistic Environments (Differential Vehicle and Drone):

Success Rate: This metric measures the percentage of episodes where the agent successfully navigates the environment without colliding with obstacles. A high success rate in the realistic environment would be a strong indicator of the agent's readiness for real-world deployment.

2.5.2. Alignment with Project Objectives

Development of Autonomous Behaviors: The mean step reward and success rate directly correlate with the agent's ability to develop autonomous behaviors. A consistent increase in the mean step reward and a high success rate indicate that the agent has effectively learned to navigate and make decisions autonomously in the given environment.

Simulation-to-Real Transition: The success rate in the realistic environment serves as a primary indicator of the agent's sim-to-real transition capability. If the agent achieves a high success rate in the realistic environment, it suggests that the behaviors learned in simulation can be effectively transferred to real-world scenarios.

Integration of Technologies: While the evaluation criteria primarily focuses on the agent's performance, the seamless integration of technologies like Unity, Simulink, ROS2, Docker, and reinforcement learning frameworks is validated by the agent's ability to function effectively in both simulation environments. The loss function, in particular, can help diagnose potential integration issues, as inconsistencies or anomalies in the loss trend might indicate problems in the interaction between the agent and the environment.

CHAPTER 3: SIMULATION RESULTS ANALYSIS

3.1. Rewards Evaluation and Policy Validation

This chapter offers a comprehensive analysis of the simulation results for two agents, by examining the evolution of rewards, the convergence of loss functions, and the overall success rates in different environments. The chapter also includes a comparison of the agents' performance during the learning process and an analysis of the success rates in agent deployment in both simplified and realistic environments.

Regarding the following presented graphs, it is essential to clarify the distinction between the steps in the reward function and the steps in the loss function. The steps in the reward function represent episode steps for the exploitation worker, while the steps in the loss function represent learning steps. These two metrics are not directly comparable and have no established conversion ratio. The episode steps in the reward function reflect the agent's interactions with the environment during exploitation, while the learning steps in the loss function represent the agent's updates to its policy based on the collected experiences.

3.1.1. Differential Vehicle

3.1.1.1. Mean Reward per Episode Step

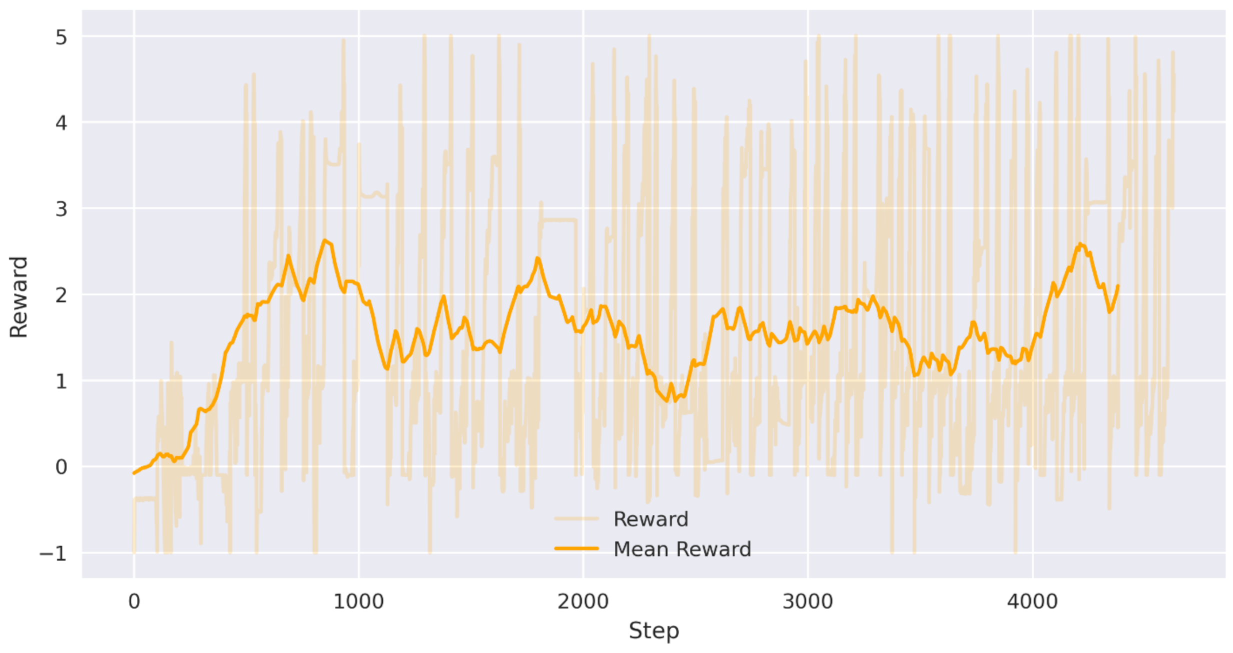


Figure 22: Differential Vehicle's Mean Reward per Episode Step

The reward evolution graph for the differential vehicle provides a visual representation of the agent's learning progress over time. The initial value starts at 0.0, indicating that the agent begins with no prior knowledge of the environment. As the agent interacts with the environment and learns from its experiences, the reward value increases.

By 500 steps, the reward reaches a value of 2.0, suggesting that the agent has quickly adapted to the environment and is making decisions that align with the desired outcomes. However, past this point, the graph shows oscillations, indicating that the agent's performance is not consistently optimal. The reward graph is quite noisy, which can be attributed to the segmentation of the reward function. This means that the agent is presented with different reward functions depending on specific conditions, primarily the distance to the target. The oscillations could be a result of the agent navigating through different segments of the episode, each with its own reward structure.

3.1.1.2. Loss per Training Step

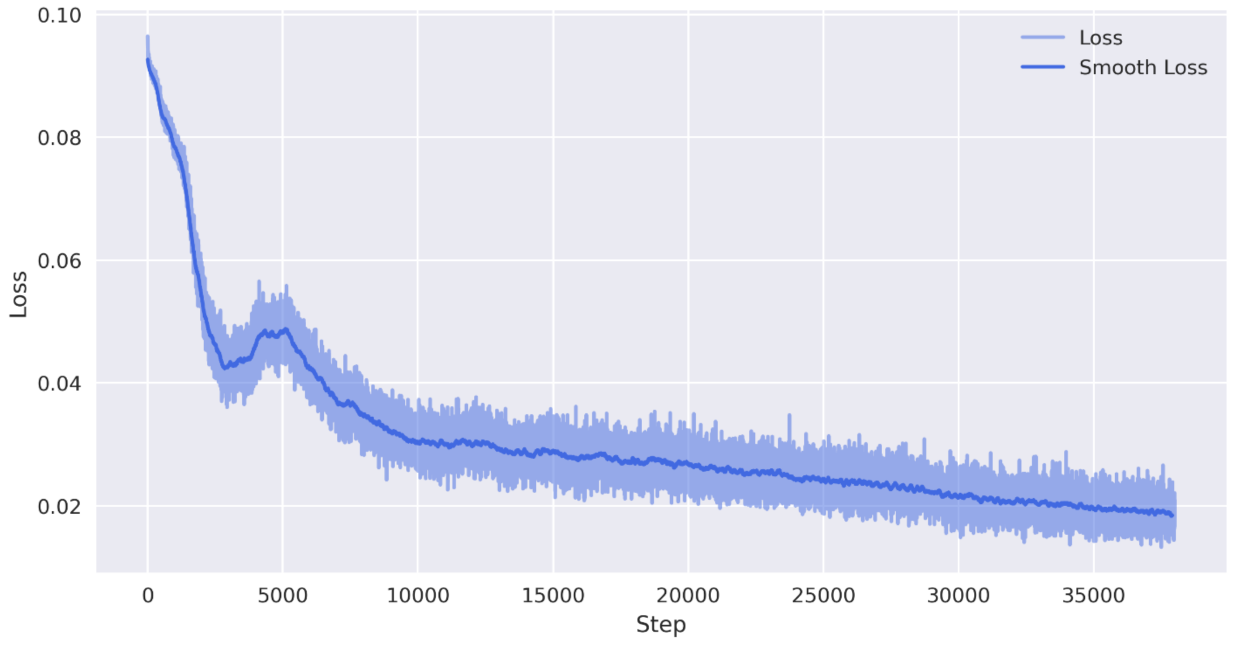


Figure 23: Differential Vehicle's Loss per Training Step

The loss function graph for the differential vehicle provides insights into the convergence of the D4PG algorithm. The initial value of the loss function is 0.095, which is relatively high, indicating that the agent's initial predictions of Q-values are not very accurate. However, as the agent continues to learn and update its policy, the loss value decreases, showing a downward trend. This suggests that the agent's predictions are becoming more aligned with the target Q-values over time.

A notable observation in the loss function graph is the local maximum at 5000 steps. This spike might indicate a temporary divergence in the agent's learning, where its predictions became less accurate for a brief period. This could be a point where the agent might have explored a new strategy, leading to a temporary increase in prediction error.

The final value of the loss function is 0.019, which is significantly lower than the initial value. This indicates that by the end of the training, the agent's predictions of Q-values have become much more accurate, and the D4PG algorithm has converged to a stable policy.

3.1.2. Drone

3.1.2.1. *Mean Reward per Episode Step*

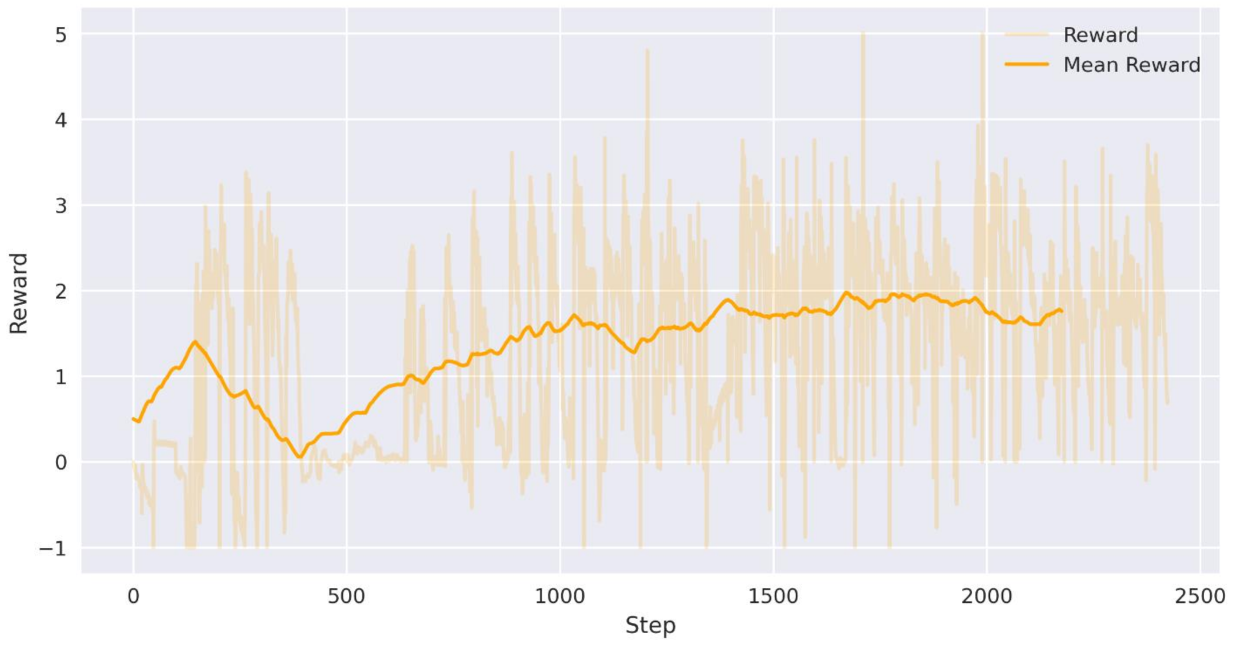


Figure 24: Drone's Mean Reward per Episode Step

The reward evolution graph for the drone, similar to the differential vehicle, starts at 0.0, indicating that the agent begins with no prior knowledge of the environment. By 200 steps, the reward reaches a value of 1.4, suggesting that the agent has quickly adapted to the environment and is making decisions that align with the desired outcomes. However, at around 400 steps, the reward value plummets back to 0. This sudden drop could be attributed to the agent exploring a new strategy that did not yield favorable results.

After the steep reduction at 400 steps, the reward graph shows a steady upward trend, indicating that the agent is gradually improving its performance and learning to make better decisions. The final value of the reward is around 1.8, which is higher than the initial value, suggesting that the agent has made progress in learning the environment and optimizing its actions.

3.1.2.2. Loss per Training Step

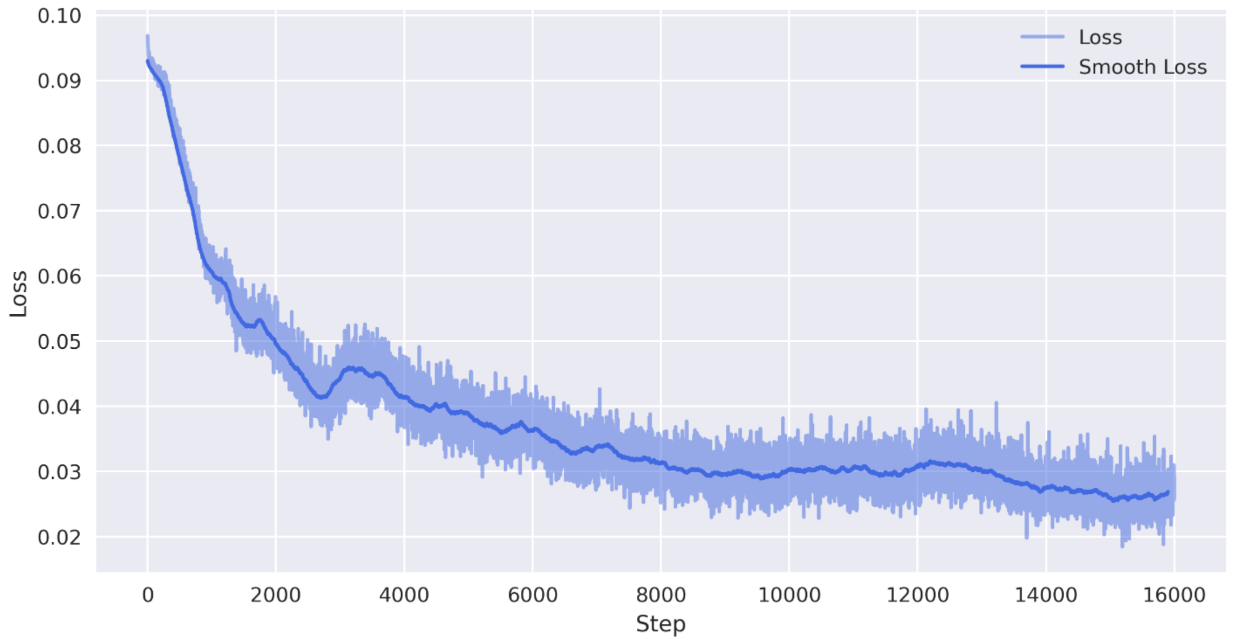


Figure 25: Drone's Loss per Training Step

Starting with an initial loss value of 0.093, the drone's loss function graph shows a general downward trend, indicating the convergence of the D4PG algorithm. This trend suggests that as the drone continues its interactions with the environment, its predictions of Q-values become increasingly accurate, aligning more closely with the target Q-values.

A subtle point of interest is the slight local maxima around 3600 steps. Even though it is barely noticeable, such a blip might indicate a momentary divergence or a brief period where the drone explored a different strategy, leading to a minor increase in prediction error.

However, the overall trend remains downward, with the loss function settling at a final value of 0.025 by 16000 steps. This significant reduction from the initial value underscores the drone's effective learning and the convergence of the D4PG algorithm to a stable and optimized policy.

3.1.3. Agents' Comparison During the Learning Process

The reward evolution graphs for both the differential vehicle and the drone show that both agents start with no prior knowledge of the environment, as indicated by the initial reward value of 0.0. However, the drone adapts more quickly to the environment, reaching a reward value of 1.4 within 200 steps, while the differential vehicle takes 500 steps to reach a reward value of 2.0.

The loss function graphs for both agents show a general downward trend, indicating the convergence of the D4PG algorithm. However, the drone's loss function decreases more steadily, settling at a final value of

0.025 by 16000 steps, while the differential vehicle's loss function shows more oscillations and reaches a final value of 0.019.

The differential vehicle, being a non-holonomic system, has constraints on its motion, which means it cannot move in any arbitrary direction from its current position. This inherent nature of the differential vehicle, combined with its velocity control strategy, means that it has to learn a more complex mapping between its actions and the resulting state transitions. This complexity can lead to a longer time convergence in its learning process.

In contrast, the drone, at low velocities, can be considered a holonomic system. This means it has the ability to move in any direction irrespective of its current orientation. Given this nature and its position control strategy, the drone's learning process is more straightforward, allowing for faster convergence. The drone's control mechanism, which cascades three distinct control loops, provides a more granular control over its movement, further aiding its learning process.

3.2. Success Rates in Agent Deployment

In this section, the success rates of the differential vehicle and the drone in both the simplified and realistic environments for a total of 120 episodes in each environment for each vehicle are analyzed. The success rates are evaluated based on four criteria:

- **Target Reached:** The agent successfully navigated to the target without any collisions.
- **Collision with Static Obstacle:** The agent collided with a static obstacle.
- **Collision with Moving Obstacle:** The agent collided with a moving obstacle.
- **Maximum Time Reached:** The agent took the maximum allowed time for an episode without reaching the target.

3.2.1. Differential Vehicle

3.2.1.1. Simplified Environment

Table 1: Differential Vehicle's Success Rates in the Simplified Environment

Criteria	Episodes	Percentage (%)
Target Reached	100	83.33
Collision with Static Obstacle	2	1.67
Collision with Moving Obstacle	6	5.00
Maximum Time Reached	12	10.00

In the simplified environment, the differential vehicle successfully reached the target in 83.33% of the episodes. Collisions with static and moving obstacles were relatively low, at 1.67% and 5%, respectively. The vehicle reached the maximum allowed time without reaching the target in 10% of the episodes.

3.2.1.2. Realistic Environment

Table 2: Differential Vehicle's Success Rates in the Realistic Environment

Criteria	**Episodes**	**Percentage (%)**
Target Reached	94	78.33
Collision with Static Obstacle	4	3.33
Collision with Moving Obstacle	10	8.33
Maximum Time Reached	12	10.00

In the realistic environment, the differential vehicle's success rate decreased slightly to 78.33%. Collisions with static obstacles increased to 3.33%, and collisions with moving obstacles increased to 8.33%. The percentage of episodes where the maximum time was reached remained the same at 10%.

3.2.2. Drone

3.2.2.1. Simplified Environment

Table 3: Drone's Success Rates in the Simplified Environment

Criteria	**Episodes**	**Percentage (%)**
Target Reached	108	90.00
Collision with Static Obstacle	2	1.67
Collision with Moving Obstacle	4	3.33
Maximum Time Reached	6	5.00

In the simplified environment, the drone had a higher success rate than the differential vehicle, reaching the target in 90% of the episodes. Collisions with static and moving obstacles were low, at 1.67% and 3.33%, respectively. The drone reached the maximum allowed time without reaching the target in 5% of the episodes.

3.2.2.2. Realistic Environment

Table 4: Drone's Success Rates in the Realistic Environment

Criteria	**Episodes**	**Percentage (%)**
Target Reached	92	76.67
Collision with Static Obstacle	6	5.00
Collision with Moving Obstacle	14	11.67
Maximum Time Reached	8	6.67

In the realistic environment, the drone's success rate decreased to 76.67%. Collisions with static obstacles increased to 5%, and collisions with moving obstacles increased to 11.67%. The percentage of episodes where the maximum time was reached increased slightly to 6.67%.

3.3. Agents' Comparison

In the simplified environment, the differential vehicle achieved a success rate of 83.33%, which slightly decreased to 78.33% in the realistic environment. Collisions with static and moving obstacles were relatively low in the simplified environment but saw a slight increase in the realistic setting.

The drone exhibited a higher success rate in the simplified environment, reaching the target in 90% of the episodes. However, this rate dropped to 76.67% in the realistic environment. Notably, the drone experienced a more significant increase in collisions with moving obstacles in the realistic environment compared to the differential vehicle.

Two main factors could account for the observed differences in performance between the two environments:

Model Simplifications: The simplified environment was designed to represent the core dynamics of the system, with certain complexities omitted for ease of simulation. The simplified environment for the differential vehicle closely resembled its realistic counterpart. However, the drone's simplified environment did not capture the transients of the system as accurately. This discrepancy could explain the more significant drop in performance for the drone when transitioning from the simplified to the realistic environment.

Sensor Noise: The introduction of noise in the positioning sensor measurements and higher noise in the LiDAR measurements in the realistic environment can affect the agent's decision-making process. This noise introduces uncertainty, which can lead to increased collisions or other suboptimal behaviors.

The differential vehicle's performance in both environments suggests a strong capability to generalize its learned behaviors. The learned policy in the simplified environment seems to be robust enough to handle the added complexities of the realistic environment. The drone, while showcasing promising results, might require additional fine-tuning or training in a more representative environment before considering real-world deployment. However, since the drone was not physically prepared for real-world testing within the timeframe of this project and its physical availability for real-world deployment was not within the scope of this project, only the differential vehicle will be transferred to a real environment.

CHAPTER 4: REAL-WORLD METHODOLOGY

This chapter offers a comprehensive overview of the methodologies and technologies employed to bring the differential vehicle from the virtual simulations to a physical laboratory setting. Providing a description of the hardware and software components, sensors, and control systems. The real-world testing environment is characterized and the process of adapting and transferring the agent to the real environment is discussed.

4.1. Description of the Technologies

In the real-world implementation of the differential vehicle, a combination of hardware and software components is utilized to achieve autonomous navigation. The following sections details the specific technologies used in the differential vehicle system.

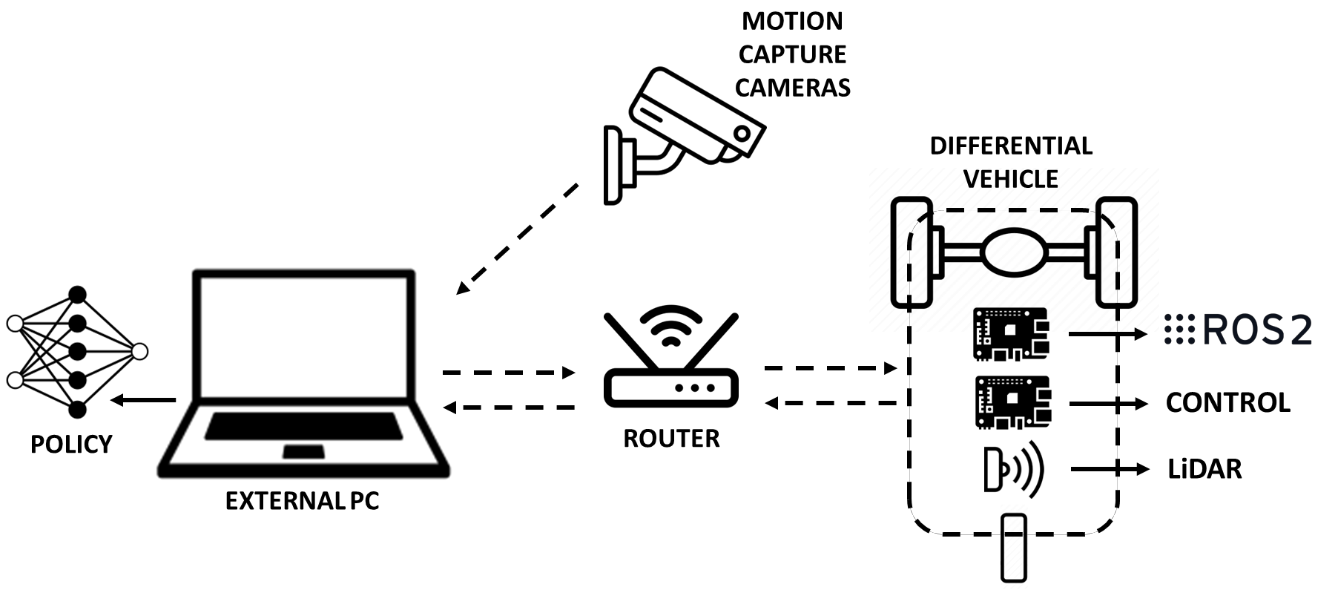


Figure 26: Real-World Environment Setup

<u>Hardware Components</u>

The differential vehicle is equipped with two Raspberry Pi units, both running Raspberry Pi OS. One Raspberry Pi is dedicated to the control system, while the other is responsible for communication purposes, interfacing with the ROS2. The latter Raspberry Pi also has available computational resources to run additional software as needed. The policy for the vehicle is implemented on an external computer and communicates with the vehicle via ROS2.

Software Components

The software implementation for the differential vehicle was designed with modularity and real-world deployment in mind. As a result, transitioning from the simulated model to the real-world vehicle was relatively straightforward. The primary addition to the software stack was the 'LiDAR node', which uses the rplidar Python library to collect raw measurements from the LiDAR sensor, process the data, and publish it in a format that the policy node expects. Additionally, modifications were made to the policy ROS2 node for deployment purposes.

Sensors

The vehicle's perception of its environment is enabled by the LiDAR sensor (RPLIDAR brand). As previously discussed, this sensor provides a two-dimensional "slice" of the surrounding environment, essential for obstacle detection and avoidance. Additionally, external motion capture cameras are employed. These cameras are designed to identify specific features or markers on the vehicle, allowing for accurate pose estimation.

Control System

The control systems for the differential vehicle are implemented using Matlab and Simulink. These control systems manage the vehicle's linear and angular velocities, employing the same control algorithm as in the realistic simulation environment.

4.2. Real Environment Description

The real-world testing environment for the differential vehicle is an indoor laboratory setting. The environment's characteristics are:

- **Physical Layout and Dimensions:** The laboratory is a square room with dimensions of 6 meters by 4 meters. The floor is flat, ensuring consistent traction and movement for the differential vehicle.
- **Obstacles:** Within the laboratory, two static obstacles are present, made up of cardboard boxes. These obstacles can be rearranged, offering a flexible layout for testing. However, the arrangement in the real-world setting is kept simpler than the configurations found in the simulation environments. In addition to the static obstacles, a moving obstacle was introduced in one of the tests. This dynamic obstacle was not considered in the final analysis of the multiple tests, serving primarily as an exploratory component of the study.
- **Velocity Constraints:** Due to control requirements, the differential vehicle's linear velocity is limited to 0.4 m/s and angular velocity is limited to π rad/s.
- **Data Collection:** Throughout the tests, the vehicle's trajectory was collected. This data provides insights into the vehicle's path and its interactions with the environment. Additionally, success rates were documented, focusing primarily on the vehicle's ability to navigate around the static obstacles without collisions.
- **Safety Measures:** To prevent potential collisions and ensure the safety of the vehicle, human intervention is made if a collision is imminent, preventing it and terminating the episode.

4.3. Agent Adaptation and Transfer to the Real Environment

The adaptation and transfer of the agent to the real environment involved several steps, including the integration of hardware components, the implementation of software components, and the modification of the agent's policy to accommodate real-world constraints.

4.3.1. LiDAR Node Implementation

The node's primary responsibility is to acquire raw data from the LiDAR sensor, process it, and then publish it in a format suitable for the policy node. This is achieved by the following process:

1. **Raw Data Acquisition:** The LiDAR node initiates communication with the LiDAR sensor through the specified port (/dev/ttyUSB0). Once the connection is established, the LiDAR motor starts, and scanning begins. The node continuously collects scans from the sensor using the iter_scans() method.
2. **Data Treatment:** Each scan from the LiDAR consists of multiple readings. These readings are sorted based on their angles. To ensure consistency with the dimensionality of the observation space, and therefore with the neural network's input layer structure, the total number of readings is downsampled, selecting uniformly distributed measurements in the horizontal plane from the total of the LiDAR readings. For each selected reading, the angle is converted from degrees to radians, and the distance is converted from millimeters to meters.
3. **Data Publishing:** The processed data is then packaged into a LaserScan message and published to the /lidar_scan topic. This topic serves as the interface between the LiDAR node and the policy node.

4.3.2. Policy Node Implementation

The Neural Node is responsible for determining the vehicle's actions based on sensor readings by carrying out the following actions:

1. **Data Subscription:** The policy node subscribes to three primary topics: /car_odom for the vehicle's current position and orientation, /target for the desired destination, and /lidar_scan for sensory data from the LiDAR node.
2. **Data Processing and Action Determination:** The relative position of the target with respect to the vehicle's reference frame is computed and concatenated with the LiDAR readings to form the input tensor for the neural network model. The actor network processes this input tensor to produce the desired linear and angular velocities for the vehicle.
3. **Action Publishing:** The determined are then packaged into a TwistStamped message and published to the /twist_ref topic, which the vehicle's control system subscribes to.

CHAPTER 5: REAL-WORLD EXPERIMENTAL RESULTS

In the real-world environment, a series of tests were conducted to evaluate the performance of the differential vehicle. The environment was characterized by its physical layout, obstacles, velocity constraints, data collection methods, and safety measures, as detailed in the previous chapter. The tests were structured as episodes, where each episode represented a single run of the vehicle from a starting point to a target destination. The objective for this chapter is to evaluate the vehicle's ability to navigate the environment autonomously, avoiding static obstacles and reaching the target within a specified time frame.

5.1. Success Rates in Agent Deployment

The differential vehicle was subjected to a total of 15 episodes within the real-world environment.

Table 5: Differential Vehicle's Success Rates in the Real Environment

Criteria	Episodes	Percentage (%)
Target Reached	11	73.33
Collision with Obstacle	3	20.00
Maximum Time Reached	1	6.67

The vehicle successfully navigated to the target without any collisions in 11 episodes, translating to a success rate of 73.33%.

5.2. Analysis of the Trajectories

Throughout the testing phase, the vehicle's trajectory was recorded. These trajectories were plotted on an xy plane for the most representative episodes.

In the episodes where the target was reached, the trajectories show smooth and direct paths towards the destination. These paths demonstrate the vehicle's ability to effectively interpret sensory data, make informed decisions, and successfully navigate the environment.

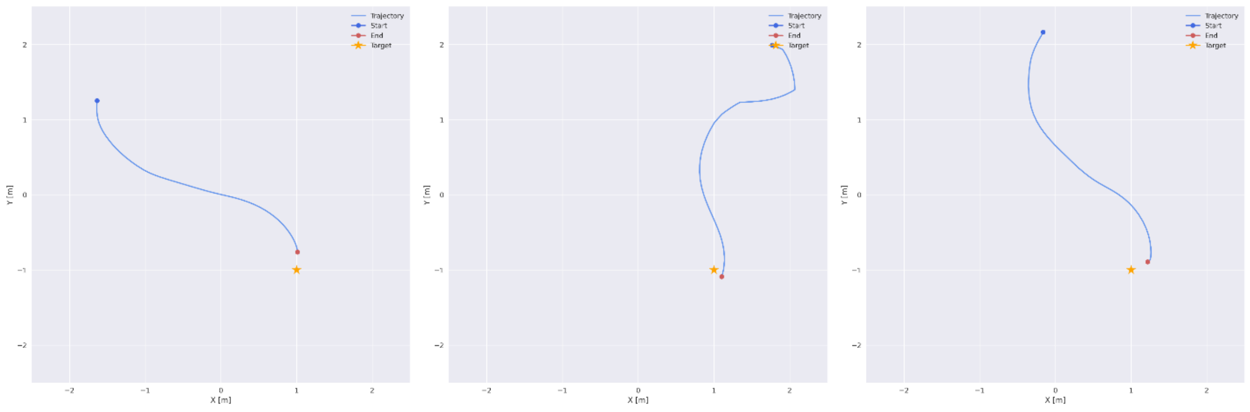


Figure 27: Episode Trajectories Where Target Was Reached

In episodes where a collision occurred, the trajectories show the vehicle approaching an obstacle and failing to navigate around it successfully. These instances highlight areas where the agent's policy may require further refinement or where the real-world environment introduced complexities not accounted for in the simulation. The episode where the maximum time was reached shows a trajectory where the vehicle encountered difficulties in decision-making, leading to prolonged navigation without reaching the target.

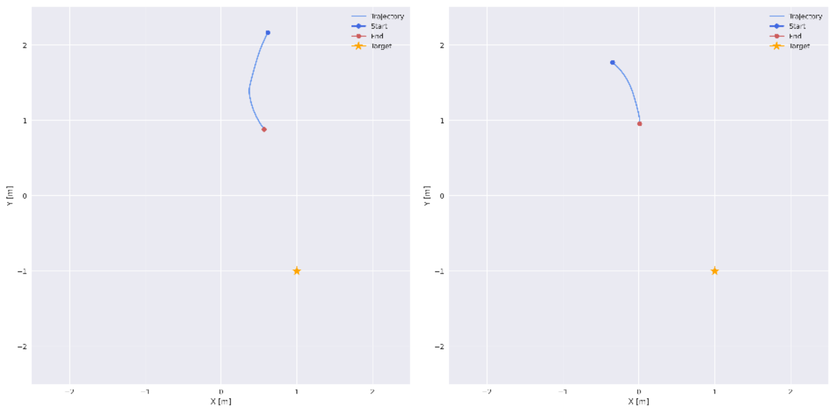


Figure 28: Episode Trajectories Where Collision Occurred

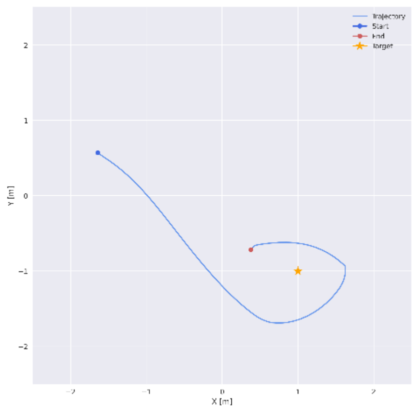


Figure 29: Episode Trajectory Where Maximum Time Was Reached

5.1. Comparison Between Simulated and Real Results

The following table presents the success rates of the differential vehicle in the simplified, realistic, and real-world environments.

Table 6: Differential Vehicle's Success Rates Comparison Across Different Environments

Criteria	Simplified	Realistic	Real-World
Target Reached (%)	83.33	78.33	73.33
Collision with Obstacle (%)	6.67	11.66	20.00
Maximum Time Reached (%)	10.00	10.00	6.67

5.2. Limitations and Challenges in the Real-World Implementation

Several factors can account for the observed discrepancies between the simulated and real-world results:

Differences in Obstacle Geometry: While the layout of obstacles was simplified in the real environment, the differences in geometry and layout of these obstacles might introduce new challenges. Variations in shape and size result in observations the agent has never encountered during training, leading to unexpected behaviors.

Vehicle Geometry and Dimensions: The real-world vehicle might have slight differences in its geometry and dimensions compared to its simulated counterpart. Such differences lead to the vehicle colliding with the edges of obstacles.

Sensor Noise and Inaccuracies: The real-world environment introduces additional sensor noise, which can affect the agent's decision-making process. Such noise introduces uncertainty, which might have been minimized in simulated environments.

Complexities of the Real World: The real-world environment inherently possesses complexities that are hard to replicate in simulations. Factors such as surface textures can influence the dynamics and transients of the real-world system**.**

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1. Conclusions and Findings

This thesis targeted the design and implementation of autonomous navigation and obstacle avoidance behaviors for two distinct types of vehicles: a differential vehicle and a drone, using reinforcement learning. The project's objectives extended beyond the design and simulation phases to real-world application, emphasizing the crucial simulation-to-real transition.

Autonomous Behaviors Development: Both the differential vehicle and the drone successfully learned autonomous navigation behaviors in simulated environments. The reward evolution graphs for both agents showed that they started with no prior knowledge of the environment and gradually adapted to it, making decisions that aligned with the desired outcomes. The loss function graphs indicated the convergence of the D4PG algorithm, with the agents' predictions of Q-values becoming more accurate over time.

Simulation-to-Real Transition: The differential vehicle was successfully transitioned to the real-world environment, where it achieved a success rate of 73.33% in reaching the target without collisions. The trajectories in successful episodes showed smooth and direct paths towards the destination, demonstrating the vehicle's ability to effectively interpret sensory data and navigate the environment.

Integration of Technologies: The seamless integration of various technologies, including Unity, Simulink, ROS2, Docker, and reinforcement learning frameworks, was successfully achieved. This interdisciplinary approach not only facilitated the development and simulation phases but also ensured reproducibility and compatibility across platforms, which is crucial for real-world deployment.

Differences Between Simulation and Real World: The discrepancies observed between the simulated and real-world results can be attributed to factors such as differences in obstacle geometry, vehicle geometry and dimensions, sensor noise, and the inherent complexities of the real world. These factors introduce new challenges and uncertainties that the agent may not have encountered during training.

Feasibility of Reinforcement Learning in Sim-to-Real: The project demonstrated the feasibility of using reinforcement learning for sim-to-real transfer in mobile robotic applications. The differential vehicle's success in the real-world environment suggests that reinforcement learning can be a viable approach for developing autonomous behaviors in mobile robots.

6.2. Contributions to the Field

Methodological Contributions: The project has successfully integrated multiple technologies and platforms, including Unity, Simulink, ROS2, Docker, and reinforcement learning frameworks, to create a complete system for autonomous navigation of mobile robots. This setup provides a standardized framework for designing, simulating, and deploying robotic applications and ensured reproducibility and compatibility across platforms.

Addressing the Simulation-to-Real Gap: The project has successfully addressed the simulation-to-real gap with a zero-shot approximation, a significant challenge in the fields of reinforcement learning and mobile robotics. The differential vehicle was trained in a simulated environment and then successfully transitioned to the real-world environment, where it achieved a success rate of 73.33% in reaching the target without collisions.

The successful transition from simulation to real-world validates the methods used in this project and demonstrates the feasibility of using reinforcement learning for sim-to-real transfer in mobile robotic applications. It highlights the importance of accurate modeling of the environments, robust training of the agents, and careful adaptation and transfer to the real environment.

6.3. Recommendations for Future Work

Integration of Additional Sensors: Incorporating cameras as additional sensors could provide richer sensory data, enabling the agents to better perceive their environment. This would imply the use of Convolutional Neural Networks (CNNs) to process the image data and extract relevant features.

Different Control Techniques: Training the differential vehicle agent with a position control strategy, similar to the drone agent, could provide a more direct comparison between the two agents and potentially improve the differential vehicle's performance.

Advanced Neural Network Architectures: Introducing Long Short-Term Memory (LSTM) or transformer architectures for the actor and critic networks could enhance the agents' ability to learn and remember temporal dependencies in the environment, potentially improving decision-making in dynamic environments.

Testing with Dynamic Obstacles: Extending the testing scenarios to include dynamic obstacles, such as moving vehicles, would provide a more comprehensive evaluation of the agents' performance. This would also make the system more applicable to real-world scenarios where dynamic obstacles are common.

Real-World Drone Implementation: Transferring the drone agent to a real-world environment would provide valuable information into the challenges and complexities of real-world aerial navigation. This would also validate the methodologies used for the drone in a real-world context and allow for a direct comparison of the performance of the differential vehicle and the drone in real-world settings.

Retraining and Adaptation: Given the discrepancies observed between the simulated and real-world results, it is recommended to consider fine-tuning the differential vehicle agent based on the real-world environment.

REFERENCES

[1] Guizzo, E. (2018) "How Robots Are Grasping the Art of Grip," *IEEE Spectrum*. Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/mspec.2018.8405398.

[2] Broadbent, E. (2017) "Interactions with Robots: The Truths We Reveal About Ourselves," *Annual Review of Psychology*. Annual Reviews, 68(1), pp. 627–652. doi: 10.1146/annurev-psych-010416-044021.

[3] Yang, G. Z. *et al.* (2020) "Combating COVID-19—The role of robotics in managing public health and infectious diseases," *Science Robotics*. American Association for the Advancement of Science (AAAS), 5(40). doi: 10.1126/scirobotics.abb5589.

[4] Murphy, R. R. (2014) "Disaster Robotics," *IEEE Intelligent Systems*. Institute of Electrical and Electronics Engineers (IEEE), 29(4), pp. 25–29. doi: 10.1109/mis.2014.48.

[5] Murphy, R. R. (2011) "Human–Robot Interaction in Rescue Robotics," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*. Institute of Electrical and Electronics Engineers (IEEE), 41(2), pp. 138–153. doi: 10.1109/tsmcc.2010.2046732.

[6] Wurman, P. R., D'Andrea, R. and Mountz, M. (2008) "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses," *AI Magazine*. Association for the Advancement of Artificial Intelligence, 29(1), pp. 9–20. doi: 10.1609/aimag.v29i1.2062.

[7] Guizzo, E. (2018) "How Robots Are Grasping the Art of Grip," *IEEE Spectrum*. Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/mspec.2018.8405398.

[8] Li, C., Chen, L. and Chen, B. (2022) "Analyzing tracked search and rescue robots," in El-Hashash, A. (ed.) *International Conference on Biomedical and Intelligent Systems (IC-BIS 2022)*. SPIE. doi: 10.1117/12.2661488.

[9] Weyler, J. *et al.* (2023) "Towards domain generalization in crop and weed segmentation for precision farming robots," *IEEE robotics and automation letters*. Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/lra.2023.3262417.

[10] Rong, J. *et al.* (2022) "Fruit pose recognition and directional orderly grasping strategies for tomato harvesting robots," *Computers and electronics in agriculture*. Elsevier BV. doi: 10.1016/j.compag.2022.107430.

[11] Disyadej, T. *et al.* (2020) "Smart Transmission Line Maintenance and Inspection using Mobile Robots," *Advances in Science Technology and Engineering Systems Journal*. ASTES Journal. doi: 10.25046/aj050361.

[12] van Hecke, K. *et al.* (2017) "Self-supervised learning as an enabling technology for future space exploration robots: ISS experiments on monocular distance learning," *Acta astronautica*. Elsevier BV. doi: 10.1016/j.actaastro.2017.07.038.

[13] Zhao, W. *et al.* (2021) "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey" arXiv [cs.CL]. Available at: https://arxiv.org/abs/2009.13303.

[14] Thorndike, E. L. (1911) *Animal intelligence: Experimental studies*. New York: The Macmillan Company.

[15] Bellman, R. (1957) *Dynamic Programming*. Princeton University Press.

[16] Sutton, R. S. (1988) "Learning to predict by the methods of temporal differences," *Machine Learning*, 3(1), pp. 9–44. doi: 10.1007/BF00115009.

[17] Peng, N. *et al.* (2022) "Urban multiple route planning model using dynamic programming in reinforcement learning," *IEEE transactions on intelligent transportation systems: a publication of the IEEE Intelligent Transportation Systems Council*. Institute of Electrical and Electronics Engineers (IEEE), 23(7), pp. 8037–8047. doi: 10.1109/tits.2021.3075221.

[18] Llorente, F. *et al.* (2021) "A survey of Monte Carlo methods for noisy and costly densities with application to reinforcement learning," *arXiv [cs.LG]*. Available at: https://arxiv.org/abs/2108.00490.

[19] Watkins, C. J. C. H. (1989) *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England.

[20] Tesauro, G. (1995) "Temporal difference learning and TD-Gammon," *Communications of the ACM*, 38(3), pp. 58–68.

[21] Mnih, V. *et al.* (2015) "Human-level control through deep reinforcement learning," *Nature*, 518(7540), pp. 529–533.

[22] Sun, Y. *et al.* (2022) "A Friend-or-Foe framework for multi-agent reinforcement learning policy generation in mixing cooperative–competitive scenarios," *Transactions of the Institute of Measurement and Control*. SAGE Publications, 44(12), pp. 2378–2395. doi: 10.1177/01423312221077755.

[23] Brown, T. B. *et al.* (2020) "Language models are few-shot learners," *arXiv [cs.CL]*. Available at: https://arxiv.org/abs/2005.14165.

[24] Arab, K. and Mp, A. (2012) "PID Control Theory," in Introduction to PID Controllers - Theory, Tuning and Application. InTech. doi: 10.5772/34364.

[25] Chen C. and Holohan, A. (2016) "Stability robustness of linear quadratic regulators", International journal of robust and nonlinear control. Wiley, 26(9), pp. 1817–1824. doi: 10.1002/rnc.3362.

[26] Schwenzer, M. et al. (2021) "Review on model predictive control: an engineering perspective," perspective," The international journal of advanced manufacturing technology. Springer Science and Business Media LLC, 117(5–6), pp. 1327–1349. doi: 10.1007/s00170-021-07682-3.

[27] Unity. Available at: https://unity.com/products/unity-engine.

[28] Simulink. Available at: https://es.mathworks.com/products/simulink.html.

[29] ROS2. Available at: https://docs.ros.org/en/humble/index.html..

[30] Docker. (No date). Available at: https://www.docker.com/products/docker-desktop/.

[31] PyTorch. Available at: https://pytorch.org/.

[32] D4PG PyTorch Implementation. Available at: https://github.com/schatty/d4pg-pytorch.

[33] Python. Available at: https://www.python.org/.

[34] Git. Available at: https://git-scm.com/.

[35] Silver, D. *et al.* (2016) “Continuous control with deep reinforcement learning,” *arXiv [cs.LG]*. Available at: https://arxiv.org/abs/1509.02971.

[36] Barth-Maron, G. *et al.* (2018) “Distributed distributional deterministic policy gradients,” *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/1804.08617.

[37] Cubillo Llanes, D. *et al.* (2022) “Navegación autónoma de un vehículo terrestre mediante una cámara lidar”. Repositorio Universidadd Pontificia Comillas. Available at: http://repositorio.comillas.edu/jspui/handle/11531/62114.

ANNEX I: PROJECT'S ALIGNMENT WITH THE SUSTAINABLE DEVELOPMENT GOALS

The United Nations' Sustainable Development Goals (SDGs) serve as a global blueprint to achieve a better and more sustainable future for all. This project, focusing on the development of autonomous navigation for differential vehicles and drones using reinforcement learning, aligns with several of these goals.

9 INDUSTRY, INNOVATION AND INFRASTRUCTURE
11 CIUDADES Y COMUNIDADES SOSTENIBLES
17 PARTNERSHIPS FOR THE GOALS

Figure 30: SDGs This Project Is Aligned to

SDG 9: Industry, Innovation, and Infrastructure

This project promotes innovation and the adoption of intelligent and sustainable technologies within the industry. The development of autonomous systems, such as drones and differential vehicles, capable of navigating efficiently and safely, can find broad applications across various industries. This includes logistics, agriculture, infrastructure inspection, surveillance, and more. By fostering these advancements, the project contributes to building resilient infrastructure, promoting inclusive and sustainable industrialization, and supporting innovation.

SDG 11: Sustainable Cities and Communities

Autonomous systems have the potential to enhance the sustainability of cities and communities. For instance, drones can be employed for goods delivery, reducing the reliance on ground vehicles, thereby decreasing traffic congestion and greenhouse gas emissions. Autonomous differential vehicles can be employed for sustainable mobility applications. By integrating these technologies, the project aids in making cities and human settlements inclusive, safe, resilient, and sustainable.

SDG 17: Partnerships for the Goals

By utilizing open-source platforms and collaborating across various disciplines, this project encourages cooperation and knowledge and technology exchange. The employment of reinforcement learning techniques and collaboration with the artificial intelligence community contributes to the development of novel solutions and technologies.