A logo with text on it

Description automatically generated

הפקולטה להנדסה

המעבדה לרשתות וחישוב

סימולטור רחפנים

במעבדה לרשתות וחישוב

שלמה אסף

תומר בכר

פרויקט שנה ד' לקראת תואר ראשון בהנדסה

מנחה: לי-און רביב

מנחה אקדמי: פרופ' אמיר לשם

אוקטובר 2023

Table of Contents

[Multi-Agents in A city Traffic Simulation High-Level Design 5](#_Toc163064832)

[Preface 5](#_Toc163064833)

[Simulation Concept – Model Description 6](#_Toc163064834)

[Environment 6](#_Toc163064835)

[Graphs 7](#_Toc163064836)

[GUI 7](#_Toc163064837)

[Agent 7](#_Toc163064838)

[Reinforcement Learning Platform 8](#_Toc163064839)

[Main 8](#_Toc163064840)

[Q-Learning 11](#_Toc163064841)

[Deep Q-Learning 12](#_Toc163064842)

[KD-tree Q-Learning 14](#_Toc163064843)

[Training method and results 15](#_Toc163064844)

[Conclusion 16](#_Toc163064845)

[Simulation Parameters 16](#_Toc163064846)

[Imported Data 16](#_Toc163064847)

[Configurable parameters - Drone 16](#_Toc163064848)

[Configurable parameters - PygameHandler 16](#_Toc163064849)

[Exported Data 16](#_Toc163064850)

[Simulation tutorial 17](#_Toc163064851)

[Prerequisites 17](#_Toc163064852)

[Requirements 17](#_Toc163064853)

[Simulator configuration 17](#_Toc163064854)

[Creates a map for the simulator 17](#_Toc163064855)

[Creates a graph for the current map 18](#_Toc163064856)

[Neo4j – how to run the graph server 18](#_Toc163064857)

[Drone server and main simulator 18](#_Toc163064858)

[Software Architecture 19](#_Toc163064859)

[Environment (environment/main.py) 19](#_Toc163064860)

[EnvDroneObj (environment/env\_drone.py) 19](#_Toc163064861)

[Drone Server (environment/ drones\_server.py) 21](#_Toc163064862)

[Pygame Handler (environment/ pygame\_handler.py) 21](#_Toc163064863)

[EnvironmentAPI (environment/env\_info\_api.py) 23](#_Toc163064864)

[EnvironmentInfoServer (environment/env\_info\_server.py) 23](#_Toc163064865)

[Graph Server (environment/graph\_server.py) 24](#_Toc163064866)

[Drone (drone/Drone.py) 25](#_Toc163064867)

[Internal GPS (utils/util\_classes.py) 26](#_Toc163064868)

[MotionControl (drone/motion\_controller.py) 27](#_Toc163064869)

[Buttery controller (drone/power\_management.py) 28](#_Toc163064870)

[TwoDRadar (drone/radar.py) 28](#_Toc163064871)

[Mission (drone/mission\_control.py) 31](#_Toc163064872)

[Path (drone/mission\_control.py) 31](#_Toc163064873)

[MissionControl (drone/mission\_control.py) 32](#_Toc163064874)

[VehicleMissionControl (drone/mission\_control.py) 32](#_Toc163064875)

[Drone Agent \*\*TODO – update from here 33](#_Toc163064876)

[Reinforcement learning platform 34](#_Toc163064877)

[Metric Logger (RL/metrics.py) 34](#_Toc163064878)

[Legacy 35](#_Toc163064879)

[Reinforcement learning platform 35](#_Toc163064880)

[Fake Env 35](#_Toc163064881)

[Replay Buffer 36](#_Toc163064882)

[Q-Network 36](#_Toc163064883)

[DQN Agent 37](#_Toc163064884)

[Drone Net 38](#_Toc163064885)

[Drone Agent 2 38](#_Toc163064886)

[Bibliography and other sources 40](#_Toc163064887)

[*Q-Learning: Theory and Applications - https://www.annualreviews.org/doi/10.1146/annurev-statistics-031219-041220* 40](#_Toc163064888)

[appendix A – SRS 41](#_Toc163064889)

[1. Introduction 42](#_Toc163064890)

[1.1 Purpose 42](#_Toc163064891)

[1.2 Theoretical Background 42](#_Toc163064892)

[1.3 Product Scope 46](#_Toc163064893)

[1.4 Additional References 47](#_Toc163064894)

[2. Overall Description 47](#_Toc163064895)

[2.1 Product Perspective 47](#_Toc163064896)

[2.2 Product Functions 47](#_Toc163064897)

[2.3 User Classes and Characteristics 48](#_Toc163064898)

[2.4 Operating Environment 48](#_Toc163064899)

[2.5 Design and Implementation Constraints 48](#_Toc163064900)

[2.6 User Documentation 48](#_Toc163064901)

[2.7 Assumptions and Dependencies 48](#_Toc163064902)

[3. External Interface Requirements 48](#_Toc163064903)

[3.1 User Interfaces 48](#_Toc163064904)

[3.2 Software Interfaces 48](#_Toc163064905)

[4. System Features 48](#_Toc163064906)

[4.1 Simulator - environment 48](#_Toc163064907)

[4.2 Simulator - drone agent 50](#_Toc163064908)

[4.3 RL platform 51](#_Toc163064909)

# Multi-Agents in A city Traffic Simulation High-Level Design

## Preface

This project aims to create a platform for the development and testing of reinforcement learning algorithms in the field of aerial vehicle control. This platform will allow researchers to design and evaluate RL algorithms in various simulated aerial environments, enabling them to explore the potential of these algorithms and identify their strengths and limitations. This software will support the advancement of RL techniques in this important and rapidly evolving field by providing a flexible and user-friendly tool for experimentation and analysis.

By applying reinforcement learning methods, we aimed to find a traffic management algorithm for multi-drone agents' environments. We expect the resulting algorithm to perform better than a non-AI algorithm. Also, we wanted to make a dynamic simulation environment so that it will be able to further research in the future, for example - improving drone shipping policies for networks of drones that carry the same packages simultaneously or improving missions’ assignment policies.



# Simulation Concept – Model Description

## Environment

In this project, the environment is implemented as a 2D simulation where the height of the terrain is visually represented by different colors. The simulation provides a top-down view of the landscape, allowing the user to observe and analyze the elevation variations in the simulated area. Different colors are used to represent different height levels, providing a clear visual indication of the terrain's topography. This representation helps the user to understand if the agent is behaving as we expect from it.

Regarding the environment, the simulator can either generate a random map or receive a 3D map's GIS data as input to create a learning environment for the drone. In the case of the latter option, the simulator also offers the ability to upload an online map. While this feature assists the user in visualizing the environment, it does not influence the learning process.

A picture containing black and white

Description automatically generated with medium confidence

Figure 2 GIS map generator

Figure 1 random map generator

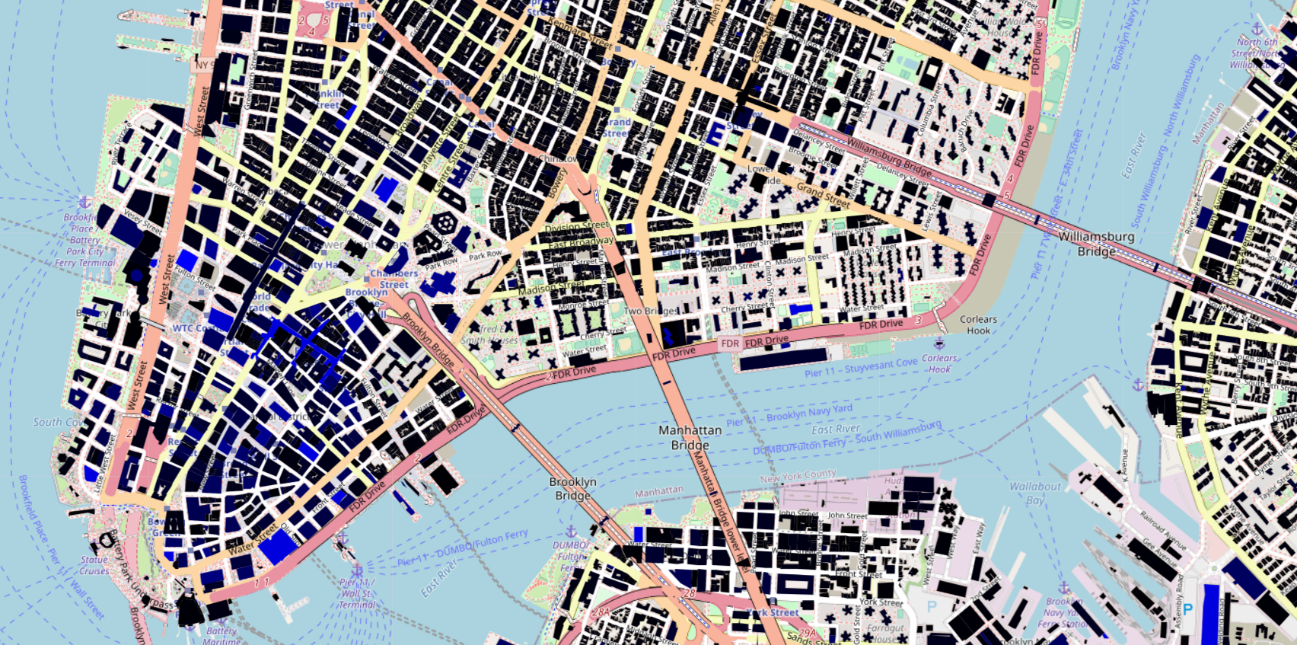


Figure 3 map of New York with GIS data on it

## Graphs

A major challenge encountered during the calculation of drone routes was the complexity of the 3D map, which proved too intricate to compute within a reasonable timeframe. To address this issue, we implemented a graph-based approach to explore all possible drone movement options. While the calculation process may take some time (depending on the user's computer), the resulting graph is stored on a server for future reuse, particularly when operating within the same map. For this purpose, we employed a service named neo4j for the graph server. Neo4j is a graph database management system that excels in storing and querying graph data structures, offering efficient traversal and manipulation capabilities essential for our drone route optimization tasks.

## GUI

The GUI, implemented using the pygame library, provides a graphical interface for users to interact with the 2D simulation environment. Through the GUI, researchers can visualize the height-based representation of the environment, observe the movements and interactions of the agents, and monitor the ongoing simulations. As demonstrated in the examples provided in the preceding paragraph, the simulator can either generate a random map or import an existing one. The pygame library aids in visualizing these maps.

The GUI facilitates real-time data visualization, displaying the agents' positions, **and** their interactions with the terrain and other objects or agents. Researchers can modify simulation parameters, such as the number of agents, mission settings, and environmental conditions, directly from the GUI. To optimize performance and computational resources, the GUI can be optionally shut down while the simulation runs. By closing the GUI, researchers can allocate more system resources to the simulation itself, enhancing its efficiency and enabling faster computations.

## Agent

The agents in the simulation operate within the environment, engaging with the terrain and other elements present. Equipped with sensors, these agents can detect obstacles within a specific range, allowing them to make well-informed decisions and adapt their actions based on elevation changes. Additionally, the agents are equipped with batteries that have an optimal flight duration of one hour.

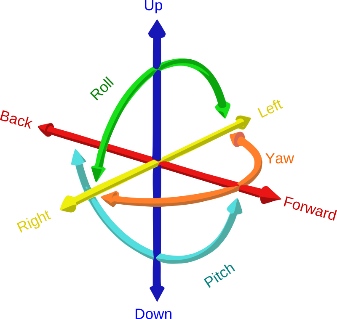
In most of the flight simulations, the flying object has **Six Degrees Of Freedom (6DOF)**. The 6DOF is a concept that refers to the six [mechanical degrees of freedom](https://en.wikipedia.org/wiki/Degrees_of_freedom_(mechanics)) of movement of a [rigid body](https://en.wikipedia.org/wiki/Rigid_body) in [three-dimensional space](https://en.wikipedia.org/wiki/Three-dimensional_space). Specifically, the body is free to change [position](https://en.wikipedia.org/wiki/Position_(geometry)) as forward/backward, up/down, left/right [translation](https://en.wikipedia.org/wiki/Translation_(physics)) in three [perpendicular](https://en.wikipedia.org/wiki/Perpendicular) [axes](https://en.wikipedia.org/wiki/Coordinate_axis), combined with changes in [orientation](https://en.wikipedia.org/wiki/Orientation_(geometry)) through [rotation](https://en.wikipedia.org/wiki/Rotation) about three perpendicular axes, often termed yaw, pitch, and roll.  
In the context of our simulator, the terms yaw, pitch, and roll do not play a significant role in training our drone agent. Instead, our simulator operates on a concept known as **Three Degrees Of Freedom (3DOF),** allowing objects to move exclusively along three axes: forward/backward, up/down, and left/right.

Figure 4 6DOF movement

To encourage efficient behavior, the agents receive a map of their operational environment, allowing the drone to learn the most efficient route to reach its destination. Once each drone receives its assignment and calculates the optimal path, it begins its flight. To prevent collisions, we employ reinforcement learning to penalize any drone that collides with another drone or building and reward drones that complete their missions without incidents. This reward system encourages the agent to learn the most efficient route to its destination and back.

## Reinforcement Learning Platform

### Main

Overall, the use of drones for delivery is a rapidly evolving field with significant potential to revolutionize the way goods are transported. There is a great deal of ongoing research in this area, and the findings of these studies will play a key role in shaping the future of drone delivery.

Academic research[[1]](#footnote-1)[[2]](#footnote-2) on drones for delivery has explored a wide range of topics, including technical challenges, such as limited range and endurance, as well as economic and logistical considerations, such as the cost and efficiency of drone delivery compared to traditional methods. Other research has focused on regulatory and safety issues, including developing standards and guidelines for delivery drone operation and integrating drones into existing air traffic systems. There have also been studies on the social impacts of drone delivery, including potential effects on employment and ethical considerations.

Other academic works have focused on the regulatory and safety issues surrounding drone delivery, including the development of standards and guidelines for the operation of delivery drones and the integration of drones into existing air traffic systems.

One way to address the challenges of using drones for delivery is using reinforcement learning algorithms. These algorithms allow drones to learn from their experiences and make decisions based on the rewards or consequences of their actions.

For example, a reinforcement learning algorithm could be used to optimize the flight path of a delivery drone to minimize the time and energy required to complete a delivery. The algorithm could learn from its past experiences and continually adjust its flight path based on the rewards or consequences of its actions.

Reinforcement learning algorithms have the potential to significantly improve the efficiency and effectiveness of drone delivery. They can help drones navigate complex environments, make decisions on the fly, and learn from their experiences to continually improve their performance.

Overall, the use of reinforcement learning algorithms is a promising approach for addressing the technical challenges of using drones for delivery and has the potential to significantly advance the field of aerial transportation.

Reinforcement Learning (RL) is a type of machine learning in which an agent learns to interact with an environment to maximize a reward signal. The agent receives a reward for performing actions that lead to desired outcomes and learns to select actions that maximize the cumulative reward over time. RL has been applied to a wide range of problems, including control, recommendation systems, and natural language processing.

Several key concepts are important to understand in RL:

Environment: The environment is the system that the agent interacts with. It can be physical or virtual, and it includes the states and actions available to the agent, as well as the rules and dynamics of the system.

Agent: The agent is the decision-making entity in the system. It takes actions in the environment to achieve some goal or maximize a reward.

State: A state is a snapshot of the current situation in the environment. It includes all the information the agent needs to decide.

Action: An action is a choice that the agent can make in each state. It can be a physical action, like moving a robot arm, or a more abstract action, like choosing which item to purchase.

Reward: A reward is a numerical value the agent receives in response to acting in a particular state. It is used to guide the agent's learning and help it determine which actions are more likely to lead to a positive outcome.

One of the key challenges in RL is balancing exploration and exploitation, which refers to the trade-off between trying out new actions to gather more information about the environment versus relying on current knowledge to maximize reward. There are several approaches to addressing this trade-off, including epsilon-greedy algorithms, which select a random action with a small probability, and upper-confidence-bound (UCB) algorithms, which select actions based on the maximum possible reward given the current state of knowledge.

Multi-agent reinforcement learning (MARL) refers to the scenario where multiple agents are learning to interact with each other and the environment simultaneously. The agents may be cooperative, meaning they work towards a common goal, or they may be competitive, meaning they have conflicting goals. MARL introduces additional challenges, such as the need to learn appropriate communication protocols and the difficulty of defining the overall reward signal in a way that aligns with the goals of the individual agents.

There are several approaches to MARL, including centralized learning, in which a central agent learns a policy that is then executed by the other agents, and decentralized learning, in which each agent learns its policy independently. Another approach is hybrid learning, which combines elements of centralized and decentralized learning.

Centralized learning has been used in MARL in a variety of scenarios, including resource allocation tasks[[3]](#footnote-3) and multi-robot coordination[[4]](#footnote-4). In these examples, a central agent learns a policy based on the actions and observations of all the agents in the system, and the other agents execute this policy. One advantage of centralized learning is that it can handle complex environments and large numbers of agents since the central agent can integrate information from all the agents in the system. However, it can be vulnerable to the failure of the central agent and may not scale well to very large systems.

Decentralized learning has also been applied in MARL, with a focus on scenarios where the agents are autonomous and must make decisions based on local information. Examples include multi-robot exploration[[5]](#footnote-5) and traffic control[[6]](#footnote-6). In these examples, each agent learns its policy based on its observations and actions without relying on a central agent. Decentralized learning can be more resilient to the failure of individual agents since the other agents can continue to operate even if one agent fails. However, it can be more difficult to learn effective policies in decentralized systems, especially in complex environments.

Hybrid learning combines elements of centralized and decentralized learning, and it has been applied in a range of MARL scenarios, including multi-robot coordination[[7]](#footnote-7) and recommendation systems[[8]](#footnote-8). In these examples, the system includes both a central agent and decentralized agents, and the agents can learn both centralized and decentralized policies depending on the situation. Hybrid learning can combine the benefits of centralized and decentralized learning, but it can also introduce additional complexity.

Deep learning has also been applied to RL and MARL, resulting in significant performance improvements on a range of tasks. In particular, deep Q-networks (DQN) have been successful in learning policies for complex environments with high-dimensional state spaces. In MARL, deep learning has been used to learn communication protocols between agents and to learn decentralized policies.

Overall, RL and MARL are active areas of research with many open challenges and potential applications. Further research is needed to improve the sample efficiency and stability of RL algorithms, as well as to develop MARL methods that can effectively handle complex and dynamic environments with many agents.

In conclusion, in this project, we would like to use RL and MARL algorithms to generate a multi-agent network of delivery drones that will work together in each civil area with maximum efficiency and safety and according to the regulations of the state.

### Q-Learning

Q-learning (Watkins, 1989) is a form of model-free reinforcement learning. It can also be viewed as a method of asynchronous dynamic programming (DP). It provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains.  
In Q-learning, the agent's experience consists of a sequence of distinct stages or episodes.[[9]](#footnote-9)

In the episode, the agent:

* observes its current state
* selects and performs an action
* observes the subsequent state
* receives an immediate payoff
* adjusts its values using a learning factor , according to:

Where:

In our simulation, we chose to train our model with Q-learning because Q-learning is a popular reinforcement learning algorithm that has been used in various applications.   
Like any algorithm, it has its own set of advantages and disadvantages:

**Advantages of Q-learning:**

1. Model-Free Approach: Q-learning is a model-free reinforcement learning algorithm, which means it doesn't require prior knowledge of the environment's dynamics or transition probabilities.

2. Ease of Implementation: Q-learning is relatively easy to understand and implement. It involves updating a Q-table based on observed rewards and transitions.

3. Convergence: Under certain conditions, Q-learning is guaranteed to converge to the optimal Q-values. This means that with enough exploration and sufficient time, it can find the best policy for an agent to maximize its rewards.

4. Off-Policy Learning: Q-learning is an off-policy algorithm, which means it can learn from experiences generated by a different policy than the one it's currently following. This property can be advantageous in situations where exploration and exploitation need to be balanced effectively.

**Disadvantages of Q-learning:**

1. Curse of Dimensionality: Q-learning can suffer from the curse of dimensionality when dealing with high-dimensional state spaces. The Q-table grows exponentially with the number of state-action pairs, making it impractical for complex environments.

2. Continuous Action Spaces: It is primarily designed for discrete action spaces. Adapting Q-learning to work with continuous action spaces can be challenging and often requires discretization or function approximation methods like Deep Q-Networks (DQNs).

3. Exploration-Exploitation Dilemma: Q-learning's exploration strategy can be inefficient, especially in large state spaces. Balancing exploration and exploitation to ensure adequate exploration can be challenging.

4. Requires Knowledge of State Space: Q-learning assumes that you have complete knowledge of the state space, which may not be feasible in some real-world scenarios. Partial observability can lead to suboptimal or inefficient policies.

5. Slow Convergence: In practice, Q-learning can converge slowly, especially when dealing with noisy or stochastic environments. It may require many iterations to converge to an optimal policy.

The primary obstacle we encountered in our simulation revolved around the vast state space we had to contend with. Within a 1x1 kilometer map, there were 100 possible positions an agent could occupy, and at each position, there were 23 parameters that determine the current state{radar data (18), relative angle, velocity magnitude, target angle, target magnitude, battery level}. Consequently, the total state space exploded to a staggering 100^23 combinations, rendering it impractical for traditional Q-Learning.

In response to this challenge, we pursued two distinct strategies to address the problem: Deep Q-Learning and Q-Learning integrated with an evolutionary KD-tree structure (referred to as KD-tree Q-Learning). In the subsequent sections, we will delve into the details of these approaches.

### Deep Q-Learning

Deep Q-Learning (DQL) is a powerful reinforcement learning technique that extends the traditional Q-Learning algorithm to handle environments with high-dimensional or continuous state spaces. It combines Q-Learning with artificial neural networks, allowing it to approximate the Q-values rather than maintaining a tabular representation of them.

DQL uses Neural Network Approximation Instead of using a Q-table to store Q-values for each state-action pair, DQL uses a neural network to approximate the Q-function. The neural network takes the state as input and outputs Q-values for all possible actions. DQL typically employs an experience replay buffer to store past experiences (state, action, reward, next state). During training, batches of experiences are randomly sampled from this buffer to break the temporal correlations in the data. This improves learning stability.[[10]](#footnote-10)

To stabilize the training, DQL introduces the concept of a target network. This is a separate neural network that is periodically updated with the parameters of the main Q-network. The target network helps prevent divergence during training by providing more stable target Q-values.   
DQL uses a target Q-value for training. The target Q-value is a combination of the immediate reward and the maximum estimated Q-value from the target network for the next state. This helps in propagating accurate value estimates backward through the network. DQL uses gradient descent optimization to update the Q-network's parameters and improve the Q-value estimates over time. Training continues until convergence or a predefined stopping criterion is met.

**Advantages of Deep Q-Learning:**

1. Handles High-Dimensional State Spaces: DQL can effectively handle environments with large and continuous state spaces, making it suitable for real-world applications.

2. Generalization: The neural network can generalize from observed states to unseen ones, improving exploration and reducing the need for an exhaustive state representation.

3. Complexity: DQL can handle complex environments and tasks, such as playing video games, robotics control, and autonomous driving.

4. State-of-the-Art Performance: DQL has achieved state-of-the-art results in various domains, surpassing human-level performance in some cases.

However, DQL also has its challenges and potential drawbacks, including issues related to instability during training, hyperparameter tuning, and the need for a substantial amount of data for learning.

When we attempted to employ DQN for training the agents within the simulation, we encountered a significant challenge: the learning process failed to converge even after more than 48 hours of training. Our analysis leads us to attribute this issue to the exceedingly large state space involved.

### KD-tree Q-Learning

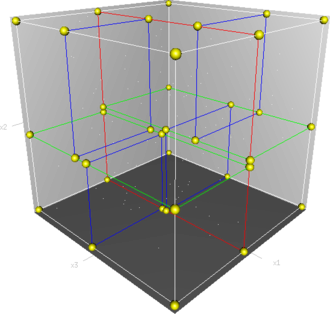
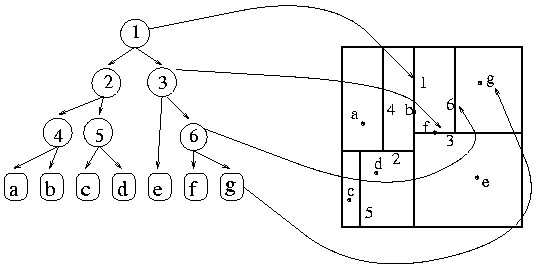
The KD-tree, a pivotal data structure in computational geometry and machine learning, facilitates efficient multidimensional space partitioning, particularly for applications necessitating rapid nearest-neighbor searches and range queries. Conceptually, a KD-tree employs recursive binary space division. Given a set of k-dimensional points, an axis (e.g., x, y, z for three dimensions) is chosen for initial partitioning. The points are then sorted along this axis, with the median point becoming the root of the tree. Subsequently, the space is bifurcated into two half-spaces: one encompassing points to the left of the median and the other containing those to the right. Each of these subspaces undergoes further subdivision utilizing the subsequent axis in the sequence. This recursive process persists until a predetermined termination condition is met, such as a specified maximum tree depth or a minimum threshold of points within a leaf node. In query operations, like determining the nearest neighbor to a given point, the KD-tree is traversed adhering to the axis-based splitting rules. This enables efficient pruning of subtrees that cannot contain closer points. The algorithm continues its recursive exploration until the closest point is identified or until it is ascertained that no closer points can exist in the unexplored regions. In summation, the KD-tree stands as a versatile data structure, adept at organizing points in multidimensional space, and offers swift retrieval of information contingent upon spatial proximity, finding broad utility across domains including computer graphics and machine learning.

Figure 5 KD-tree

As detailed in preceding sections, the primary challenge encountered when employing Q-learning for agent training stemmed from the vast state space. To address this issue, we employed a KD-tree as a means of function approximation within the Q-learning algorithm.

Figure 6 divides space into KD-tree.

KD-Q-learning employs a step function within a hierarchical discretization framework of the state space to approximate the quality function. It integrates state-splitting methodologies into a tree descent process, enhancing the efficiency of reinforcement learning, particularly in the initial learning stages, where information utilization surpasses that of state splitting. The core concept underlying this approach lies in utilizing states characterized by an inherent hierarchical organization. This facilitates concurrent learning at various spatial levels of abstraction. [[11]](#footnote-11)

# Training method and results

Following the completion of the simulator and a successful simulated drone flight, our next effort was to implement a controller utilizing reinforcement learning to instruct the agent in navigating to its destination and returning.

## Deep Q-Learning

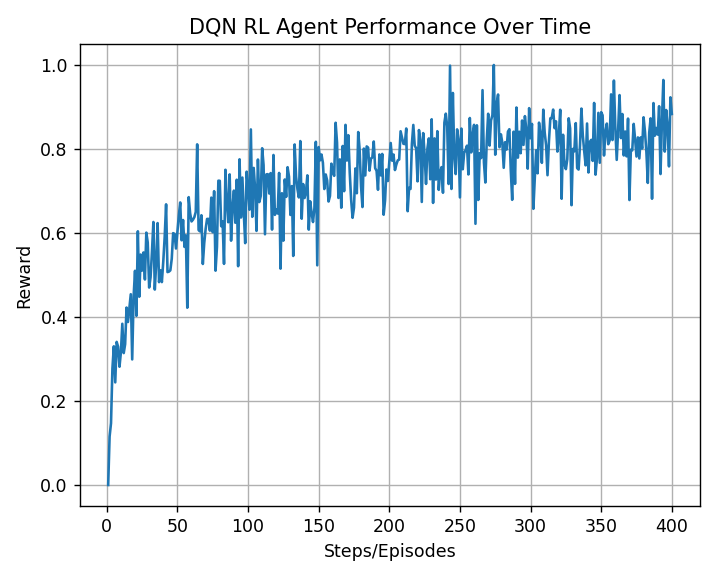
Initially, a controller tailored for a singular drone was instantiated, utilizing Deep Q-Networks (DQN) for training purposes. Subsequently, following an exhaustive 48-hour training phase without notable advancement, it was ascertained that the DQN framework was ill-suited for the intended objective, primarily due to the extensive action vector available for selection by the drone within each state. Despite prolonged training efforts, the drone demonstrated proficiency in avoiding collisions with buildings but failed to reach the designated destination. Ultimately, it was concluded that the DQN paradigm was inadequate for addressing the nuanced requirements of our problem domain.

Figure 7 Drone learning graph in DQN algorithm

## KD-tree Q-Learning

A graph with blue lines

Description automatically generatedSubsequently, we developed a second controller, this time employing KD-Q-Learning for training the agent. While there was some improvement in the learning process for this drone, as it adapted to the environment and began avoiding buildings over time, it failed to learn how to reach the designated destination or effectively avoid other drones. Our analysis indicated that the primary reason for this failure was the inability of the drone to reach its intended destination. Due to a problem with the classification of possible actions for a given state, where the KD-tree treats vectors with low Euclidean distance similarly and assigns the same action to both scenarios, the drone encountered difficulties distinguishing between scenarios where it encountered obstacles and those where it navigated freely. Consequently, the agent struggled to learn the appropriate actions accurately, particularly in navigating towards the destination. Thus, despite its effectiveness in obstacle avoidance, KD-Q-Learning fell short in adequately training the agent for precise destination navigation.

Figure 8 Drone learning graph in KD-Tree Q-Learning algorithm

## Conclusion

A red line going up

Description automatically generatedFollowing these initial attempts, we devised a novel training approach that we subsequently implemented in our simulator. This approach involved the creation of a new controller capable of managing all agents (drones) within the simulation. In the preflight phase, the controller receives a map as input, which it then transforms into a 3D graph. For each drone, the controller calculates the shortest path on this graph and communicates the path to the respective drone. As the drones initiate their flights to their destinations and back, the controller assumes a supervisory role, ensuring that the drones adhere to their designated paths while avoiding collisions with buildings. When two or more drones are on a collision course, the controller has the discretion to either maintain the original path or make collision-avoidance adjustments. This is where KD-Q-Learning comes into play. In this approach, we train the controller to guide the drones in a manner that minimizes deviations from their assigned shortest paths while effectively avoiding collisions with other drones.

Figure 9 drone movement on the map

# Simulation Parameters

### Imported Data

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Calculated value** |
| GeoJSON | GIS data from open-source | JSON |

### Configurable parameters - Drone

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Valid values** |
| Name | Every drone shell has a unique name | String |
| GPS | An object that calculates the location of the drone, can be with or without noise (in version 0.1, only without) | InternalGPS() |
| Power management | An object that calculates the battery consumption of the drone | BatteryController(): Capacity(in mAh) voltage mode(consumption rate) |
| radar | The radar has a range that it can sense | meters |
| Max speed | What are the drone capabilities | Pixel per second |
| Max height | What are the drone capabilities | meters |
| Size | A safety polygon around the drone, on the map we will address it as the drone itself | meters |
| Motion control | An object that calculates the acceleration of the drone | MotionControl() |

### Configurable parameters - PygameHandler

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Valid values** |
| Clock | Can determine the simulation clock rate |  |
| Window | Screen width and height | How many pixels in the width and height |
| Map | An image of the simulation | A path of a map that we give to the simulation |
| Drones | A list of the drones that simulating can change during the simulation | List of drones |

### Exported Data

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Calculated value** |
| LOG | TDB |  |

# Simulation tutorial

## Prerequisites

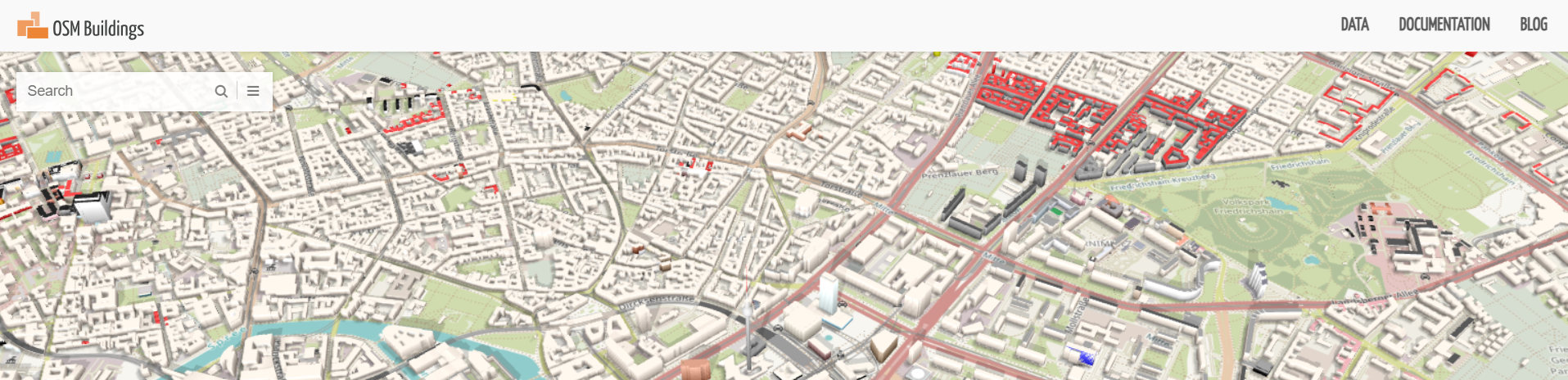
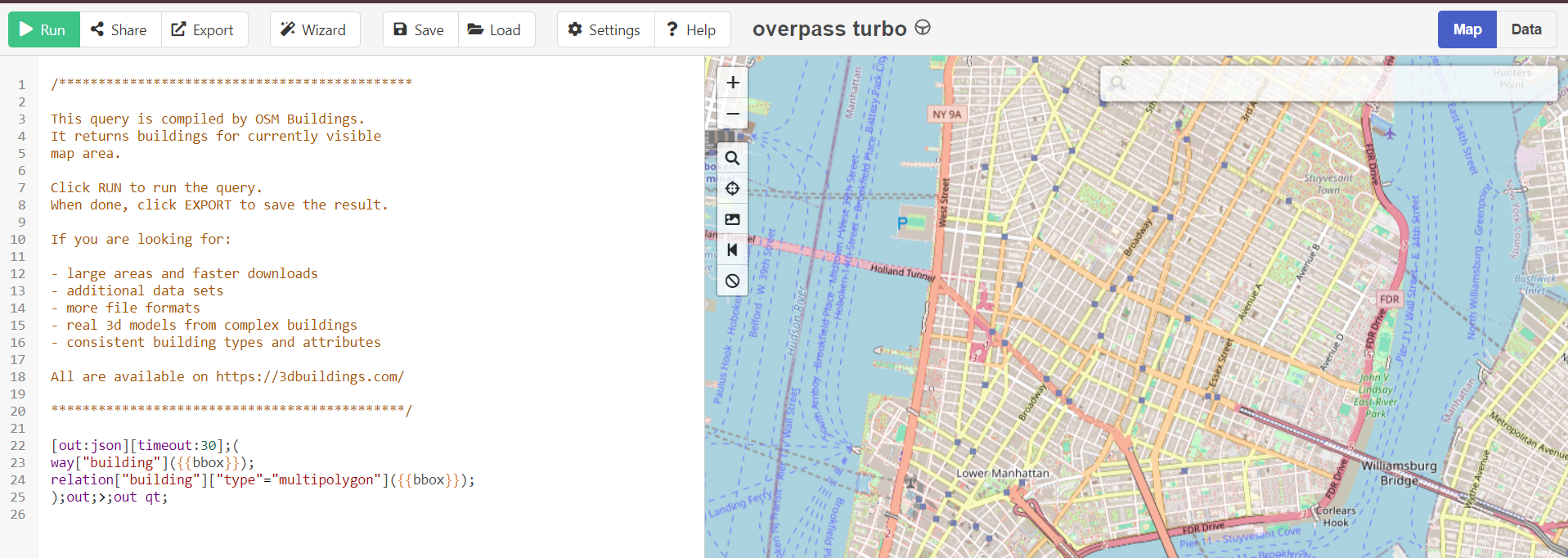
### Requirements

To operate our simulator, it must be executed within a Unix environment, as the simulation was developed and verified specifically in this environment.   
In our simulator we have a file called requirements.txt, to install all the requirements for this simulator open the terminal in PyCharm and run this line: *pip install -r requirements.txt*.

### Simulator configuration

The simulator has a configuration file named ‘Consts.py’ in the ‘utils’ folder, this file contains all the variables that the simulator uses and can be changed before running.

### Creates a map for the simulator

1. Find a city for the simulator, in our project we chose New York City, and already downloaded all the necessary data and uploaded it into the GIT.
2. Download a geojson of the area you chose from [OSM Buildings website.](https://osmbuildings.org/?lat=52.52111&lon=13.41078&zoom=15.0&tilt=30)  
   
3. Go to DATA (in the right-up part of the website) and choose “free download” button.
4. Then select the area in the map on the right and click on “run”  
   
5. The geojson that was created is saved in 'data/geojson/’ and changes the ‘GEOJSON\_PATH’ in the Consts.py for the simulator to work.

### Creates a graph for the current map

As discussed earlier, to reduce both runtime and computational complexity, the simulator employs a graph generated from the environment map used by the drone. This approach enables the drone to compute potential routes to its destination more efficiently.

To generate such a graph, the graph server needs to be activated. If the graph already exists, the server will recognize it and activate it without recomputing the graph. However, if a new map is constructed following the steps outlined in the "creating a map for the simulator" section, the graph server will first generate the graph and store it in a Neo4j server. Subsequently, it will serve as a platform for agents to access the graph.

### Neo4j – how to run the graph server

As explained, in our simulator we use Neo4j server to save the graph data for each map we want to use, to work with Neo4j in the simulator download Neo4j for desktop from the [website of Neo4j](https://neo4j.com/docs/operations-manual/current/installation/osx/), then start for free and set the password for the Neo4j to be: 11559900.

Working with Neo4j in the simulator involves the following steps:   
- Activate Neo4j on your computer by opening a terminal and entering the command::  
neo4j-admin server start  
- If this is the first time Neo4j is being run on the computer, execute the following command in a terminal:  
*neo4j-admin database import full --nodes=/Users/shlomoassaf/reps/DroneSim/nodes.csv --verbose --relationships=/Users/shlomoassaf/reps/DroneSim/relations.csv --delimiter=, --overwrite-destination*This command imports data from specified CSV files and initializes the Neo4j database- Once the Neo4j server is up and running, open the simulator and execute: *environment/graph\_server.py*This starts the graph server within the simulator environment.  
- After the graph server is up and running, execute: *environment/* *env\_info\_server.py*This command starts the environment information server, which interacts with the graph server.

### Drone server and main simulator

The drone server located at *environment/drone\_server.py*. This server manages communication with the various drones simulated within the environment. Afterward, execute *environment/main.py* to commence the simulation.

# Software Architecture

## Environment (environment/main.py)

Main

Inputs: None

Outputs: None

Method: The simulation begins in the environment, where the server and the initial client (drone) are launched, along with the Pygame handler that oversees the simulation.

## EnvDroneObj (environment/env\_drone.py)

This class acts as a client to the drone server. Whenever a new drone initiates communication with the server, the server instantiates an 'EnvDroneObj' and preserves the drone's status within it.

Adjust drone color

Inputs: Height (of the drone)

Outputs: None

Method: This method determines the color of the drone according to its height.

Check-in viewport

Inputs: X, Y viewports (what part of the map is shown), zoom factor

Outputs: None

Method: The function verifies the visibility of the drone in the application and updates the "in\_viewport" data attribute. This attribute is set to true if the drone is visible and false otherwise.

Getters (velocity, location, battery, target vector)

Inputs: None

Outputs: velocity, location, battery, and target vector respectively

Method: This getter function sends a request to a socket connection to retrieve the data. It receives the serialized data, deserializes it, assigns it to the object's attribute, and returns the deserialized data.

Accelerate

Inputs: X, Y, Z

Outputs: None

Method: Transmit an acceleration request through the socket using a vector [x, y, z], where each component of the vector specifies the acceleration in its respective direction.

Accelerate2

Inputs: Direction (angle)

Outputs: None

Method: Transmit an acceleration request through the socket using a relative angle for the acceleration.

Update

Inputs: None

Outputs: None

Method: Send an update request via the socket to prompt the receiving drone to compute its battery status and GPS location.

Set imitate

Inputs: Action

Outputs: None

Method: When the drone is in learning mode, it can replicate an action performed by a trained agent (or a user choice). This involves obtaining an action and transmitting it to the agent through a socket, enabling the drone to execute the action accordingly.

Turn to

Inputs: Direction (angle)

Outputs: None

Method: Transmit a request for changing direction through the socket using a relative angle to determine the new direction.

Start learning

Inputs: None

Outputs: None

Method: Initiate the learning process for the agent by sending a request through the socket.

## Drone Server (environment/ drones\_server.py)

Start Server

Inputs: None

Outputs: None

Method: Starting the server for the Drones

Server thread

Inputs: Clients (list)

Outputs: None

Method: Connect a client to the main server, this method is called only once in a thread.

## Pygame Handler (environment/ pygame\_handler.py)

The Pygame handler oversees the simulation by managing the assignment of orders to drones and transmitting those orders. Additionally, it takes charge of the graphical user interface (GUI) and the visualization of data.

Start simulation

Inputs: None

Outputs: None

Method: It's an infinite loop responsible for continuously running the GUI, handling user events, drawing the map, and displaying the legend if the simulation is in manual mode.

Handle events

Inputs: None

Outputs: None

Method: Handle different modes of the simulation, that it gets from “change mode”  
also, listen to the keyboard to see if the user presses ‘a’ to add drones

Change mode

Inputs: None

Outputs: None

Method: Change the controller mode using the keyboard:  
Space – for map\drone control  
c – for drone choosing  
f – for focus mode

Handles choose mode.

Inputs: event (key from user)

Outputs: None

Method: After the user presses ‘c’ he needs to choose the drone by its name(number) and press enter

Handle map control

Inputs: event (key from user)

Outputs: None

Method: This function assists the user in determining where to focus on our map. The user can navigate the map using the arrow keys to move in different directions and utilize the 'z' and 'x' keys to zoom in and out, as needed.

Handle drone control

Inputs: event (key from the user or the RL platform)

Outputs: None

Method: This function handles the drone's movement based on user or RL platform events. The pygame\_handler has control over one drone at a time, and this method dictates the movement of the currently controlled drone.

Drew functions (map\status\on screen\menu\heat legend\drones)

Inputs: None

Outputs: None

Method: All these functions are tasked with displaying information on the GUI. Some elements, such as the map and the drones, are crucial, while others, like the legend or the heat map, hold less significance for the user or the learning process.

Add drone

Inputs: number of drones (int)

Outputs: None

Method: Call a constructor of a drone and add it to the simulation process.

## EnvironmentAPI (environment/env\_info\_api.py)

An API for environment severs.

get\_close\_env

Inputs: pos ([x,y,z])

Outputs: array (the environment as NumPy array)

Method: It sends a request to the environment info server to retrieve the environment data from the drone's current position. This function specifically requests a limited range of the environment based on the drone's perspective. However, this function is currently inactive or unused.

get\_env

Inputs: pos ([x,y,z])

Outputs: array (the environment as NumPy array)

Method: It sends a request to the environment info server to retrieve the environment data from the drone's current position to get the drone perspective.

## EnvironmentInfoServer (environment/env\_info\_server.py)

This server is tasked with calculating the environment from the perspective of a drone, as well as determining a reward for the drone's actions.

get\_close\_env

Inputs: pos ([x,y,z])

Outputs: map (MapObject)

Method: It sends the environment data from the drone's current position. This function specifically sends a limited range of the environment based on the drone's perspective.

get\_env

Inputs: pos ([x,y,z])

Outputs: map (MapObject)

Method: It retrieves the environment data from the drone's current position to get the drone perspective.

get\_reward

Inputs: pos, target, velocity ([x,y,z])

Outputs: reward (double)

Method: calculate the drones reward:  
reward = -10000000 -> if the drone hit the ground or got out of boundary.  
reward = (10 – velocity\_magnitude) \* 1000 -> if the drone is getting close to the target.  
reward = (90 – |relative\_angle| \* 100) \* velocity\_magnitude -> if the drone is getting farther from the target.  
reward = 10000000000 -> if the drone got to the target.

## Graph Server (environment/graph\_server.py)

This server employs the Neo4j service to store and retrieve graphs efficiently. In addition, it utilizes Flask, a Python web framework, to handle server requests.

Load graph

Inputs: None

Outputs: None

Method: Attempt to load the graph from Neo4j, and if it doesn't exist, this function creates the graph.

calculate

Inputs: data (as a json vie request API)

Outputs: result (json)

Method: get request from the user and send it to perform\_claculation, returns what perform\_ claculation returns.

perform\_calculation

Inputs: data (json)

Outputs: result (json)

Method: calculate the path between two nodes on the graph.

## Drone (drone/Drone.py)

The drone acts as the client within the simulation and serves as the foundational class for the agent we aim to train. Within the drone, various classes are encompassed, such as GPS, motion control, Battery Controller, and radar.

Figure 10 Drone structure

Getters (GPS\Velocity\name)  
Inputs: None

Outputs: None

Method: Calls GPS to get position or velocity and return it, or returns the name of the drone.

Setters (GPS\Velocity\name)  
Inputs: vector [X, Y, Z] for GPS or Velocity or name

Outputs: None

Method: Send the internal GPS the vector or set the name of the drone.

Calculate GPS   
Inputs: None

Outputs: None

Method: Call the GPS.calculae\_position, after calculating the new position, the function checks if the drone passed the max height or crushed into the ground, and prints it to the user.

Calculate power consumption

Inputs: None

Outputs: None

Method: Call power\_controller.calculate\_battery of the power management to calculate the drone’s remaining energy.

## Internal GPS (utils/util\_classes.py)

Getters (GPS\Velocity\initiael location)  
Inputs: None

Outputs: None

Method: Getters according to names.

Setters (GPS\Speed\VelX\VelY\VelZ)  
Inputs: vector [X, Y, Z] or an Integer for velocity in a specific direction

Outputs: None

Method: Setters according to names.

Calculate position   
Inputs: None

Outputs: None

Method: Calculate the position of the drone by adding to its current location its current velocity multiplied by the time that passed from the previous calculation:  
   
work the same for the Y and Z axis.

## MotionControl (drone/motion\_controller.py)

Accelerate   
Inputs: accelerate vector (x, y, z)

Outputs: None

Method: This function is responsible for controlling the drone's motion. It takes an acceleration vector as input and adjusts the drone's speed along each axis based on the vector.

Accelerate2  
Inputs: direction (angle)

Outputs: None

Method: This function is responsible for controlling the drone's motion. It takes an angle as input and adjusts the drone's velocity magnitude according to the receiving angle.

Turn to  
Inputs: direction (angle)

Outputs: None

Method: This function is responsible for controlling the drone's motion. It takes an angle as input and adjusts the drone's movement so it will head to the new angle.

Go to point  
Inputs: target ([x,y,z])

Outputs: None

Method: get a 3D point and adapt the drone velocity to go to that point

stop  
Inputs: None

Outputs: None

Method: set drones velocity to [0,0,0].

## Buttery controller (drone/power\_management.py)

Getters (Mode\battery percentage\ battery capacity)  
Inputs: None

Outputs: None

Method: Getters according to names.

Setters (Mode\battery percentage\ battery capacity)  
Inputs: mode (int)\percentage (float)\capacity(float)

Outputs: None

Method: Setters according to names.

Calculate battery

Inputs: velocity vector and accelerate vector [X, Y, Z]

Outputs: None

Method: calculate the power consumption of the drone according to its speed vector and its battery mode using a static function called **power consumption**. the ‘calculate battery’ updates the battery capacity and the battery percentage with this calculation:  
every simulation cycle the simulation subtracts the current power in the battery from the consumption rate:

As for the battery mode:  
emergency = 0  
conservation = 1  
performance = 2

With this formula, the drone will be able to fly in conservation mode for approximately 1 hour.

## TwoDRadar (drone/radar.py)

The radar class emulates the drone's radar system, dividing its field of view into six regions and three zoom scopes. This division aids the drone in reducing its potential actions and states, enabling the utilization of deep Q-learning (which will be elaborated on later) for effective decision-making.

Figure 11 drone regions

Figure 12 drone ranges

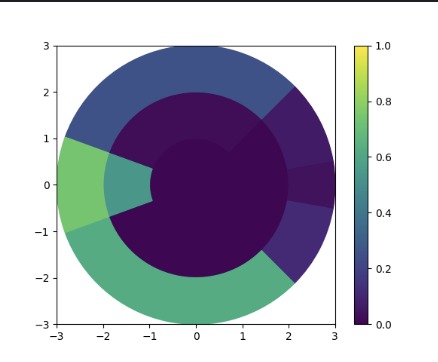


Figure 13 Plot of the radar in action

It is important to note that the range and angles of the radar can be adjusted as needed. The values provided here are the ones utilized during the RL process. This method involves the radar saving and updating two CSV files, namely 'angles.csv' and 'distances.csv', which contain the necessary data for obstacle calculations.

Calculate relative angles  
Inputs: direction angle(int)

Outputs: None

Method: Responsible for updating the radar's indices dictionary based on the provided direction angle.

1. The method checks if the direction angle is the same as the previous angle. If they are the same, it returns without making any updates.
2. The angles are adjusted by subtracting the direction angle from the radar's angles array.
3. Any angles that become less than -180 are wrapped around to the positive range by adding 360.
4. The distances are stored in a separate variable.
5. The code iterates over the predefined regions and scopes.
6. For each combination of region and scope, the corresponding indices in the indices dictionary are determined based on specific conditions.
7. If the region is not 'BACKWARD', the indices are calculated based on the range of angles and distances specified.
8. If the region is 'BACKWARD', the indices are calculated based on specific conditions that account for angles in both positive and negative ranges.
9. Finally, the last\_angle variable is updated with the direction\_angle for future comparisons. angles(np.array) – direction angle

Update sense circle  
Inputs: input\_nap(np.array), direction\_angle(int)

Outputs: None

Method: This function updates the sense circle of the drone based on the input map and the direction angle of the drone's velocity.

1. Verifies that the shape of the input map matches the expected size of (). If the shape is different, a ValueError is raised.
2. The function calls the calculate\_relative\_angles() method, which updates the indices\_dict with relative values based on the direction\_angle.
3. The function iterates over the predefined regions and scopes.
4. For each combination of region and scope, it retrieves the corresponding indices from the indices\_dict and assigns the corresponding portion of the input\_map to the sensor\_date\_dict.
5. Additionally, it calculates the logarithm base-2 of the sum of non-zero elements in the input\_map for each region-scope combination and assigns the result to the sensor\_compact\_date\_dict.
6. Finally, the function ends with the pass statement, which signifies that no additional code execution is needed at this point.

Pre calculation  
Inputs: None

Outputs: None

Method: This function generates the angle matrix and the distance matrix, both with a size of   
 These matrices are saved as CSV files ('angles.csv' and 'distances.csv').  
Then, the function retrieves the new data from the CSV files and updates the angles and distances attributes accordingly.   
Finally, for the initial iteration, the function calls calculate\_relative\_angles() with a direction angle of 0.

Get sensor data  
Inputs: compact (Boolean), as\_vector (Boolean)

Outputs: the sensor data (can be compact if the flag is on)

Method: Based on the provided flags, this function returns the sensor\_data\_dict computed in the update\_sense\_circle() function. The data can be returned either as a vector or as a dictionary, and it can be in a compact form or not.

## Mission (drone/mission\_control.py)

Set path  
Inputs: None

Outputs: None

Method: set a path to the agent, the path is pre calculated by the graph server.

## Path (drone/mission\_control.py)

Set next point  
Inputs: None

Outputs: None

Method: check if the current node isn’t the end and if not continue to the next node.

get next point  
Inputs: None

Outputs: None

Method: check if the current node is not the end and if not return to the next node.

get path from server  
Inputs: start, target ([x,y,z])

Outputs: Path

Method: send a request to the graph server for t path between the start and the target, the geraph server will calculate the shortest path and send it back to Path.

## MissionControl (drone/mission\_control.py)

Find nearest drone  
Inputs: mission.

Outputs: None

Method: Still unused, this method searches for the nearest drone to the mission's starting point. The closest available drone will be assigned the mission and directed to the starting point.

Generate mission  
Inputs: start, target ([x,y,z])

Outputs: mission

Method: assign a mission to a drone.

## VehicleMissionControl (drone/mission\_control.py)

Set mission  
Inputs: mission.

Outputs: None

Method: assign a mission to a drone via server.

Mission start  
Inputs: None

Outputs: None

Method: assign a drone mission status = start.

Mission step  
Inputs: None

Outputs: None

Method: while the mission still in progress and the drone didn’t get to the destination, the drone will go to the next point in the path

## Reinforcement learning platform

For enhanced flexibility with our simulator, we've developed a file called `rl\_util.py` which houses essential functions and classes for the RL platform. In this book, we won't delve into it extensively as the functions are straightforward.

## Metric Logger (RL/metrics.py)

This class is used to track and log various metrics during the training of a reinforcement learning agent.

Log step

Inputs: reward, loss, q

Outputs: None

Method: Logging Individual Steps by recording information for each step within an episode, including rewards, loss, and Q-values.

Log episode  
Inputs: None

Outputs: None

Method: At the end of an episode, this method calculates and logs the total reward, episode length, average loss, and average Q-value for the episode.

Init episode  
Inputs: None

Outputs: None

Method: Resets the metrics for a new episode.

record  
Inputs: None

Outputs: None

Method: Calculates and records moving averages for the last 100 episodes of rewards, lengths, losses, and Q-values. It also prints and logs the metrics to the specified log file. And plots and saves the moving averages for rewards, lengths, losses, and Q-values.

## Legacy

To enhance the elegance and readability of the project, we have removed certain functions and classes from the branch to which we have been assigned. All classes within this section remain in the main branch on Git and will be accessible for use by other researchers.

### Reinforcement learning platform

The machine learning platform incorporates a range of methods including Q-Learning, DQN, and KD-tree. Some of these methods were experimented with during the research phase and were determined to be unsuitable for addressing the research question. Nevertheless, to retain the knowledge and potentially apply it in future research, these methods are retained in the code.

### Drone Agent

This class represents a reinforcement learning agent responsible for controlling a drone within a simulated environment. This agent uses DQN to learn.

Get state

Inputs: None

Outputs: Current state

Method: Constructs and returns the current state of the environment as a concatenated array of radar data, velocity magnitude, target magnitude, target angle, and battery level.

step  
Inputs: Action

Outputs: Action

Method: Takes an action and applies it to the drone. The action corresponds to accelerating in x, -x, y, -y, or doing nothing. The method updates the drone's position and radar information, and checks if the drone is out of bounds.

Train  
Inputs: None

Outputs: None

Method: Trains a DQN agent to control the drone within episodes. It sets up the environment, specifies state and action sizes, and iterates over episodes, updating the agent's knowledge through experiences.

Connect to server  
Inputs: None

Outputs: None

Method: Establishes a socket connection to a server with a specified host and port. Starts a thread to handle communication with the server.

Communicate with server  
Inputs: None

Outputs: None

Method: Listens for commands from the server and executes corresponding actions, such as getting location, velocity, battery status, or executing accelerations.

### Fake Env

The 'fakeEnv' class serves as the agent's environment, responsible for computing the agent's perspective of the environment and assigning rewards based on the agent's actions.

Get env  
Inputs: Position [X, Y, Z]

Outputs: Map image

Method: Takes a pos parameter representing a position in a 3D space and returns a portion of the map's image centered around that position with a range specified by RadarSpec.RANGE.

Get close env  
Inputs: Position [X, Y, Z]

Outputs: Map image

Method: Similar to ‘get env’, but the range is determined by a constant Consts.CLOSE\_RANGE.

Get reward  
Inputs: position, target, velocity [X, Y, Z], battery capacity(float)

Outputs: None

Method: This method Computes a reward based on the current position, target position, velocity, and battery level.

Get source target  
Inputs: None

Outputs: None

Method: Static method that reads drone positions from a CSV file specified by Consts.DRONE\_POSITIONS\_PATH, and randomly selects a source and target position. It returns these positions as tuples.

### Replay Buffer

This class is used to implement a replay memory buffer, which is an important component in training deep reinforcement learning models. It allows the algorithm to learn from past experiences by sampling random batches of data to reduce the correlation between consecutive experiences.

Add experience  
Inputs: Experience [state, action, reward, next\_state]

Outputs: None

Method: This method is used to add experiences to the replay buffer. It takes an experience and appends it to the buffer. If the length of the buffer exceeds the maximum capacity, it removes the oldest experience (FIFO behavior).

Sample batch  
Inputs: batch size

Outputs: List of experiences randomly selected from the buffer

Method: This method is used to sample a batch of experiences from the buffer. the batch size specifying the number of experiences to be sampled.

### Q-Network

This class defines a Q-network architecture with three fully connected layers. It takes a state as input and outputs Q-values for each possible action in the environment.

Forward  
Inputs: state

Outputs: Q-Value

Method: This method defines the forward pass of the neural network. In a neural network, a forward pass refers to the process of moving input data through the network's layers to generate an output or prediction.

### DQN Agent

This class encapsulates the functionality of a DQN agent, including its Q-network, replay buffer, training procedure, and methods for loading/saving weights.

Store experience  
Inputs: [state, action, reward, next state]

Outputs: None

Method: Stores a tuple representing an experience (state, action, reward, next state) in the replay buffer.

Select action  
Inputs: State

Outputs: Action

Method: Given a state, this method either chooses a random action (with probability epsilon) or selects the action with the highest Q-value according to the current Q-network.

Train  
Inputs: Batch size

Outputs: None

Method: This method performs a single training step for the Q-network. It samples a batch of experiences from the replay buffer, computes the loss, and updates the Q-network's weights.

Modify learning rate  
Inputs: None

Outputs: None

Method: Adjust the learning rate of the optimizer. It reduces the learning rate gradually over time.

Load  
Inputs: Name (of weights file)

Outputs: None

Method: Attempts to load pre-trained weights for the Q-network from a file with the given name.

Save  
Inputs: Name (of weights file)

Outputs: None

Method: Saves the current weights of the Q-network to a file with the given name.

### Drone Net

This class defines a simple feedforward neural network with three hidden layers for use in reinforcement learning tasks, specifically in the context of a DQN. The online and target networks have the same architecture, but their parameters are kept separate, and the target network is not updated during training.

Forward

Inputs: Input, model

Outputs: online\target network

Method: This is the forward pass of the network. It takes an ‘input’ tensor and a ‘model’ argument indicating whether to use the online or target network.

### Drone Agent 2

This class represents a reinforcement learning agent responsible for controlling a drone within a simulated environment. This agent uses KD-tree with Q-Learning to learn.

Get state

Inputs: None

Outputs: Current state

Method: Constructs and returns the current state of the environment as a concatenated array of radar data, velocity magnitude, target magnitude, target angle, and battery level.

step  
Inputs: Action

Outputs: Action

Method: Takes an action and applies it to the drone. The action corresponds to accelerating in x, -x, y, -y, or doing nothing. The method updates the drone's position and radar information, and checks if the drone is out of bounds.

Train  
Inputs: None

Outputs: None

Method: Trains a DQN agent to control the drone within episodes. It sets up the environment, specifies state and action sizes, and iterates over episodes, updating the agent's knowledge through experiences.

Connect to server  
Inputs: None

Outputs: None

Method: Establishes a socket connection to a server with a specified host and port. Starts a thread to handle communication with the server.

Communicate with server  
Inputs: None

Outputs: None

Method: Listens for commands from the server and executes corresponding actions, such as getting location, velocity, battery status, or executing accelerations.

# Bibliography and other sources

IEEE-SA, *IEEE 802.11n-2009—Amendment 5: Enhancements for Higher Throughput,* 2009.

IEEE-SA, *IEEE 802.11-2007,* 2007.

# *Q-Learning: Theory and Applications - https://www.annualreviews.org/doi/10.1146/annurev-statistics-031219-041220*

Source for base map https: python package called 'folium'.

source for GIS data https://osmbuildings.org/

Graph server: https://neo4j.com/

# appendix A – SRS

**Table of Contents**

**Table of Contents**

**Revision History**

**1. Introduction**

1.1 Purpose

1.2 Theoretical Background

1.3 Product Scope

1.4 Additional References

**2. Overall Description**

2.1 Product Perspective

2.2 Product Functions

2.3 User Classes and Characteristics

2.4 Operating Environment

2.5 Design and Implementation Constraints

2.6 User Documentation

2.7 Assumptions and Dependencies

**3. External Interface Requirements**

3.1 User Interfaces

3.2 Software Interfaces

**4. System Features**

4.1 Simulator - environment

4.2 Simulator - drone agent

4.3 RL platform

# Introduction

## Purpose

The goal of this project is to create a platform for the development and testing of reinforcement learning algorithms in the field of aerial vehicle control. This platform will allow researchers to design and evaluate RL algorithms in various simulated aerial environments, enabling them to explore the potential of these algorithms and identify their strengths and limitations. By providing a flexible and user-friendly tool for experimentation and analysis, this software will support the advancement of RL techniques in this important and rapidly-evolving field.

## Theoretical Background

Overall, the use of drones for delivery is a rapidly evolving field with significant potential to revolutionize the way goods are transported. There is a great deal of ongoing research in this area, and the findings of these studies will play a key role in shaping the future of drone delivery.

Academic research[[12]](#footnote-12)[[13]](#footnote-13) on drones for delivery has explored a wide range of topics, including technical challenges, such as limited range and endurance, as well as economic and logistical considerations, such as the cost and efficiency of drone delivery compared to traditional methods. Other research has focused on regulatory and safety issues, including the development of standards and guidelines for delivery drone operation and the integration of drones into existing air traffic systems. There have also been studies on the social impacts of drone delivery, including potential effects on employment and ethical considerations.

Other academic works have focused on the regulatory and safety issues surrounding drone delivery, including the development of standards and guidelines for the operation of delivery drones and the integration of drones into existing air traffic systems.

One way to address the challenges of using drones for delivery is through the use of reinforcement learning algorithms. These algorithms allow drones to learn from their experiences and make decisions based on the rewards or consequences of their actions.

For example, a reinforcement learning algorithm could be used to optimize the flight path of a delivery drone in order to minimize the time and energy required to complete a delivery. The algorithm could learn from its past experiences and continually adjust its flight path based on the rewards or consequences of its actions.

Reinforcement learning algorithms have the potential to significantly improve the efficiency and effectiveness of drone delivery. They can help drones navigate complex environments, make decisions on the fly, and learn from their experiences in order to continually improve their performance.

Overall, the use of reinforcement learning algorithms is a promising approach for addressing the technical challenges of using drones for delivery and has the potential to significantly advance the field of aerial transportation.

Reinforcement Learning (RL) is a type of machine learning in which an agent learns to interact with an environment in order to maximize a reward signal. The agent receives a reward for performing actions that lead to desired outcomes and learns to select actions that maximize the cumulative reward over time. RL has been applied to a wide range of problems, including control, recommendation systems, and natural language processing.

There are several key concepts that are important to understand in RL:

Environment: The environment is the system that the agent interacts with. It can be physical or virtual, and it includes the states and actions available to the agent, as well as the rules and dynamics of the system.

Agent: The agent is the decision-making entity in the system. It takes actions in the environment in order to achieve some goal or maximize a reward.

State: A state is a snapshot of the current situation in the environment. It includes all the information the agent needs in order to make a decision.

Action: An action is a choice that the agent can make in a given state. It can be a physical action, like moving a robot arm, or a more abstract action, like choosing which item to purchase.

Reward: A reward is a numerical value the agent receives in response to taking action in a particular state. It is used to guide the agent's learning and help it determine which actions are more likely to lead to a positive outcome.

One of the key challenges in RL is balancing exploration and exploitation, which refers to the trade-off between trying out new actions to gather more information about the environment versus relying on current knowledge to maximize reward. There are several approaches to addressing this trade-off, including epsilon-greedy algorithms, which select a random action with a small probability, and upper-confidence-bound (UCB) algorithms, which select actions based on the maximum possible reward given the current state of knowledge.

Multi-agent reinforcement learning (MARL) refers to the scenario where there are multiple agents learning to interact with each other and the environment simultaneously. The agents may be cooperative, meaning they work towards a common goal, or they may be competitive, meaning they have conflicting goals. MARL introduces additional challenges, such as the need to learn appropriate communication protocols and the difficulty of defining the overall reward signal in a way that aligns with the goals of the individual agents.

There are several approaches to MARL, including centralized learning, in which a central agent learns a policy that is then executed by the other agents, and decentralized learning, in which each agent learns its own policy independently. Another approach is hybrid learning, which combines elements of centralized and decentralized learning.

Centralized learning has been used in MARL in a variety of scenarios, including resource allocation tasks[[14]](#footnote-14) and multi-robot coordination[[15]](#footnote-15). In these examples, a central agent learns a policy based on the actions and observations of all the agents in the system, and the other agents execute this policy. One advantage of centralized learning is that it can handle complex environments and large numbers of agents since the central agent can integrate information from all the agents in the system. However, it can be vulnerable to the failure of the central agent and may not scale well to very large systems.

Decentralized learning has also been applied in MARL, with a focus on scenarios where the agents are autonomous and must make decisions based on local information. Examples include multi-robot exploration[[16]](#footnote-16) and traffic control[[17]](#footnote-17). In these examples, each agent learns its own policy based on its own observations and actions without relying on a central agent. Decentralized learning can be more resilient to the failure of individual agents since the other agents can continue to operate even if one agent fails. However, it can be more difficult to learn effective policies in decentralized systems, especially in complex environments.

Hybrid learning combines elements of centralized and decentralized learning, and it has been applied in a range of MARL scenarios, including multi-robot coordination[[18]](#footnote-18) and recommendation systems[[19]](#footnote-19). In these examples, the system includes both a central agent and decentralized agents, and the agents can learn both centralized and decentralized policies depending on the situation. Hybrid learning can combine the benefits of centralized and decentralized learning, but it can also introduce additional complexity.

Deep learning has also been applied to RL and MARL, resulting in significant performance improvements on a range of tasks. In particular, deep Q-networks (DQN) have been successful in learning policies for complex environments with high-dimensional state spaces. In MARL, deep learning has been used to learn communication protocols between agents and to learn decentralized policies.

Overall, RL and MARL are active areas of research with many open challenges and potential applications. Further research is needed to improve the sample efficiency and stability of RL algorithms, as well as to develop MARL methods that can effectively handle complex and dynamic environments with many agents.

In conclusion, in this project, we would like to use RL and MARL algorithms to generate a multi-agent network of delivery drones that will work together in a given civil area with maximum efficiency and safety and according to the regulation of the state.

## Product Scope

Environment, ETA - 01.02.2023:

* build a 3D environment with physical and dynamic components.
* build a GUI platform
* build a dedicated database to maintain the simulation state
* form a log file for RL & GUI purposes
* form a config file to control the 3D environment; the config file will contains: buildings, area size, trees, topographic type
* form a config file to determine simulation scenarios; the config file will contain: the number of agents, missions, delivery centers, wind (direction, speed), restricted flight areas
* find and implement an algorithm to create a mission and deliver it

Agents, ETA - 01.04.2023:

* from a config that contains: battery capacity, max speed, radar range, communication range, safety range from static objects, safety range from dynamic objects, and weight.
* simulate drone communication system
* simulate drone collision avoidance system
* simulate a drone according to a config file

RL Platform, ETA - 01.08.2023:

* find and implement an algorithm to reword an agent
* find and implement an algorithm for a penalty
* train the model, and improve the algorithm if needed
* test the model, and report the results

Analysis of conclusions, ETA - 18.09.2023:

* half-way presentation - 26.03.2023
* create a poster with conclusions about the research - 18.09.2023
* final presentation - 18.09.2023
* the project book - 18.09.2023

## Additional References

* Richard S. Sutton and Andrew G. Barto,“[Reinforcement Learning: An Introduction](https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf)” (2015)
* Wei Wang, “[UAV Swarm Intelligence: Recent Advances and Future Trends](https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9214446)” (2020)

* [Godwin Asaamoning](https://sciprofiles.com/profile/1528484), [Paulo Mendes](https://sciprofiles.com/profile/1579230), [Denis Rosário](https://sciprofiles.com/profile/32989), and [Eduardo Cerqueira](https://sciprofiles.com/profile/1638698), “[Drone Swarms as Networked Control Systems by Integration of Networking and Computing](https://www.mdpi.com/1424-8220/21/8/2642)” (2021)

* [USA regulation for drone operation in civil area](https://www.faa.gov/regulations_policies/rulemaking/committees/documents/media/UAS_BVLOS_ARC_FINAL_REPORT_03102022.pdf)

# Overall Description

## Product Perspective

We aim to find a traffic management algorithm for multi-drone agents' environments by applying reinforcement learning methods. We expect the resulting algorithm to perform better than a non-AI algorithm.

Also, we want to make a dynamic simulation environment so that it will be able to further research in the future, for example - improving drone shipping policies for networks of drones that carry the same packages simultaneously or improving missions assignment policies.

## Product Functions

To reach the goals of this project, we’ll create a 3D simulator that enables the simulation of the traffic of 1-5K agents. The simulator will include GUI (graphic user interface) and will provide log files for analyzing the drones' movements.

The simulation will be dynamic and will contain various obstacles to create the most realistic environment as possible.

The simulation will simulate each drone separately, and they will share information through a simulated network that will be provided to them.

Reinforcement learning platform - the platform will use data from the simulator to develop efficient algorithms for drone path planning and flight control.

## User Classes and Characteristics

The users of the simulation will be mostly analysts, salesmen, and managers.

## Operating Environment

The simulation will run on a PC or laptop, with a windows environment using DOCKER.

## Design and Implementation Constraints

The database should support hundred to thousands of agents simultaneously

## User Documentation

* installation guide
* user guide

## Assumptions and Dependencies

* we assume that the drones will have communication systems and radar
* the mission assignment is determined in advance

# External Interface Requirements

## User Interfaces

* + user configuration
  + graphics (GUI)
  + data graphics

## Software Interfaces

* Database

# System Features

## Simulator - environment

4.1.1 **Description**

The system will represent a detailed three-dimensional environments that contain:

physical components such as-

* buildings
* topography
* trees

dynamic components such as-

* birds
* wind
* other drones

**Environment attributes-**

The environment will provide a timeline-based simulation at large scale for multi-agent networks.

The 3D environment map will be generated from a config file, every map that the generator will produce will be saved in a database for reuse.

every component in the environment (from solid objects to the drones) will share its location with the environment, this way, the agents will be able to “see” objects within a given range.

**Mission Assignment-**

As mentioned in section 2.7 we use the following assignment policy:

For a given set of destinations and packages, the system will allocate a drone object that can carry the package weight and have a sufficient battery level. Each drone will also have a unique departure time for safety assurance.

**GUI - graphic user interface -**

The GUI platform will enable the user to observe drones moving in a 3d environment. Since the project focus is on the simulator and the RL platform, we will define two phases for the GUI platform: Phase 1 will include a **2D** bird's eye view of the simulation, and in phase 2, we’ll consider the development of a more detailed GUI system.

4.1.2 Stimulus/Response Sequences

**Pre Running-**

user

* upload a 3d map or to provide an environment config file.
* provide a simulation config file.

system

* generate a map if needed.
* generate a simulation object.

**During Run Time -**

* mission assignment
* simulate sensing and communicating of the agents
* generate a detailed log (snapshot every X[ms]) for GUI and RL purposes.

4.1.3 Functional Requirements

REQ-1: build a 3D environment with physical and dynamic components.

REQ-2: build a GUI platform

REQ-3: build a dedicated database to maintain the simulation state

REQ-4: form a log file for RL & GUI purpose

REQ-5: form a config file to control the 3D environment, the config file will

contain: building, area size, trees, topographic type

REQ-6: form a config file to determine simulation scenarios, the config file will

contain: number of agents, missions, delivery centers, wind (direction, speed), restricted flight areas

REQ-7: find and implement an algorithm to create a mission and deliver it

## Simulator - drone agent

4.2.1 Description

**Drone agent model - flight capabilities and constraints-**

* flight - a drone agent will have the ability to successfully move between two points in space if it has enough battery.
* A mission is a delivery defined by a set of “packages” and destination.
* drone flight range will be determined by battery capacity and power consumption policy that will be applied to the simulation.
* drone agent flight is simplified so it can move between to planned points regardless of rotors' power balancing, vibration, etc.
* ascending and descending will be performed on the vertical axis.

note: the flight model will not contain real-world flight details such as pitch angle etc.

**Communication** -

Drone agents will have the ability to share information with nearby cooperative agents.  
Information that can be shared (as will determine in the design phase) is for example estimated location, current flight direction, battery status and wind data.  
The mechanism of the communication will be determined in the design phase (clusters, main control, etc.).

**Sensing and Collision Avoidance -**

Drone agents will have the ability to “see” and avoid obstacles in a way that abstract Radar capabilities

**Information storing and sharing**-

every agent will maintain the following information:

* start point
* destination point
* battery status
* current location (XYZ)
* current speed vector

every agent will share the following information:

* current location
* current speed vector

4.2.2 Stimulus/Response Sequences

Pre Running-

* add configuration file for different drones
* generate drone objects

During Run Time -

* simulate an auto-pilot drone that will be able to complete its mission and to improve its efficiency by learning the environment.

4.2.3 Functional Requirements

REQ-1: from a config that contains: battery capacity, max speed, radar range, communication range, safety range from static objects, safety range from dynamic objects, and weight.

REQ-2: simulate drone communication system

REQ-3: simulate drone collision avoidance system

REQ-4: simulate a drone according to a config file

## RL platform

4.3.1 Description

The RL platform will enable the development and testing of an algorithm for drone deliveries in a given area while avoiding obstacles in the fastest and safest way.

4.2.2 Stimulus/Response Sequences

The RL platform will reward an agent according to the route it chose, proximity to hazardous objects, and battery status at the end of a mission.

4.2.3 Functional Requirements

REQ-1: find and implement an algorithm to reword an agent

REQ-2: find and implement an algorithm for a penalty

1. Eichleay, Margaret, et al. "[Using the unmanned aerial vehicle delivery decision tool to consider transporting medical supplies via drone."](https://www.ghspjournal.org/content/7/4/500.short) Global Health: Science and Practice 7.4 (2019): 500-506.‏ [↑](#footnote-ref-1)
2. Shi, Ziji, and Wee Keong Ng. ["A collision-free path planning algorithm for unmanned aerial vehicle delivery."](https://ieeexplore.ieee.org/abstract/document/8453346?casa_token=mOYZOxFw-jMAAAAA:peGp8w19TNkQFbzRhxlQiD3PhWFg9WIPR6FuRSuzEuP_IuQGNjPwfc-IpdnldYVM-Nn_K6U) 2018 International Conference on Unmanned Aircraft Systems (ICUAS). IEEE, 2018.‏ [↑](#footnote-ref-2)
3. [Abhijit Gosavi](https://www.pdfdrive.com/search?q=Abhijit+Gosavi+%28auth.%29), “[Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning](https://www.pdfdrive.com/simulation-based-optimization-parametric-optimization-techniques-and-reinforcement-learning-e175248200.html)” (2003) [↑](#footnote-ref-3)
4. Boutilier, Craig, Thomas Dean, and Steve Hanks. "[Decision-theoretic planning: Structural assumptions and computational leverage](https://www.researchgate.net/publication/265702479_Decision-Theoretic_Planning_Structural_Assumptions_and_Computational_Leverage)." *Journal of Artificial Intelligence Research* 11 (1999): 1-94.‏ [↑](#footnote-ref-4)
5. Tambe, Milind. "[Towards Flexible Teamwork Extended Abstract](https://www.aaai.org/Papers/Symposia/Fall/1997/FS-97-02/FS97-02-030.pdf)” (1997) [↑](#footnote-ref-5)
6. Huang, Yi‐Sheng, et al. "[Control strategies for solving the problem of traffic congestion.](https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2016.0003)" *IET intelligent transport systems* 10.10 (2016): 642-648.‏ [↑](#footnote-ref-6)
7. Wang, Ying, and Clarence W. de Silva. "[A machine-learning approach to multi-robot coordination](https://www.sciencedirect.com/science/article/abs/pii/S0952197607000693)." Engineering Applications of Artificial Intelligence 21.3 (2008): 470-484 [↑](#footnote-ref-7)
8. ZHANG, SHUAI, et al. "[Deep Learning Based Recommender System: A Survey and New Perspectives Deep Learning Based Recommender System: A Survey and New Perspectives.](https://www.researchgate.net/publication/318671349_Deep_Learning_Based_Recommender_System_A_Survey_and_New_Perspectives)"‏ [↑](#footnote-ref-8)
9. Watkins, Dayan "[Machine Learning](https://link.springer.com/article/10.1007/BF00992698) .” *Kluwer Academic publisher* (1992): 279-292.‏ [↑](#footnote-ref-9)
10. Hester, Vecerik, Pietquin, Schaul, Piot, Horgan, sendonaris, Osband, Dulac-Arnold,   
    “[Deep Q-Learning from Demonstrations](https://ojs.aaai.org/index.php/AAAI/article/view/11757)” (2018) [↑](#footnote-ref-10)
11. Vollbrect, “[Hierarchic function approximation in kd-Q-learning](https://ieeexplore.ieee.org/abstract/document/884090)” (2002) [↑](#footnote-ref-11)
12. Eichleay, Margaret, et al. "[Using the unmanned aerial vehicle delivery decision tool to consider transporting medical supplies via drone."](https://www.ghspjournal.org/content/7/4/500.short) Global Health: Science and Practice 7.4 (2019): 500-506.‏ [↑](#footnote-ref-12)
13. Shi, Ziji, and Wee Keong Ng. ["A collision-free path planning algorithm for unmanned aerial vehicle delivery."](https://ieeexplore.ieee.org/abstract/document/8453346?casa_token=mOYZOxFw-jMAAAAA:peGp8w19TNkQFbzRhxlQiD3PhWFg9WIPR6FuRSuzEuP_IuQGNjPwfc-IpdnldYVM-Nn_K6U) 2018 International Conference on Unmanned Aircraft Systems (ICUAS). IEEE, 2018.‏ [↑](#footnote-ref-13)
14. [Abhijit Gosavi](https://www.pdfdrive.com/search?q=Abhijit+Gosavi+%28auth.%29), “[Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning](https://www.pdfdrive.com/simulation-based-optimization-parametric-optimization-techniques-and-reinforcement-learning-e175248200.html)” (2003) [↑](#footnote-ref-14)
15. Boutilier, Craig, Thomas Dean, and Steve Hanks. "[Decision-theoretic planning: Structural assumptions and computational leverage](https://www.researchgate.net/publication/265702479_Decision-Theoretic_Planning_Structural_Assumptions_and_Computational_Leverage)." *Journal of Artificial Intelligence Research* 11 (1999): 1-94.‏ [↑](#footnote-ref-15)
16. Tambe, Milind. "[Towards Flexible Teamwork Extended Abstract](https://www.aaai.org/Papers/Symposia/Fall/1997/FS-97-02/FS97-02-030.pdf)” (1997) [↑](#footnote-ref-16)
17. Huang, Yi‐Sheng, et al. "[Control strategies for solving the problem of traffic congestion.](https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2016.0003)" *IET intelligent transport systems* 10.10 (2016): 642-648.‏ [↑](#footnote-ref-17)
18. Wang, Ying, and Clarence W. de Silva. "[A machine-learning approach to multi-robot coordination](https://www.sciencedirect.com/science/article/abs/pii/S0952197607000693)." Engineering Applications of Artificial Intelligence 21.3 (2008): 470-484 [↑](#footnote-ref-18)
19. ZHANG, SHUAI, et al. "[Deep Learning Based Recommender System: A Survey and New Perspectives Deep Learning Based Recommender System: A Survey and New Perspectives.](https://www.researchgate.net/publication/318671349_Deep_Learning_Based_Recommender_System_A_Survey_and_New_Perspectives)"‏ [↑](#footnote-ref-19)