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

**Type chapter title (level 1)1**

Type chapter title (level 2)2

Type chapter title (level 3)3

**Type chapter title (level 1)4**

Type chapter title (level 2)5

Type chapter title (level 3)6

# 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

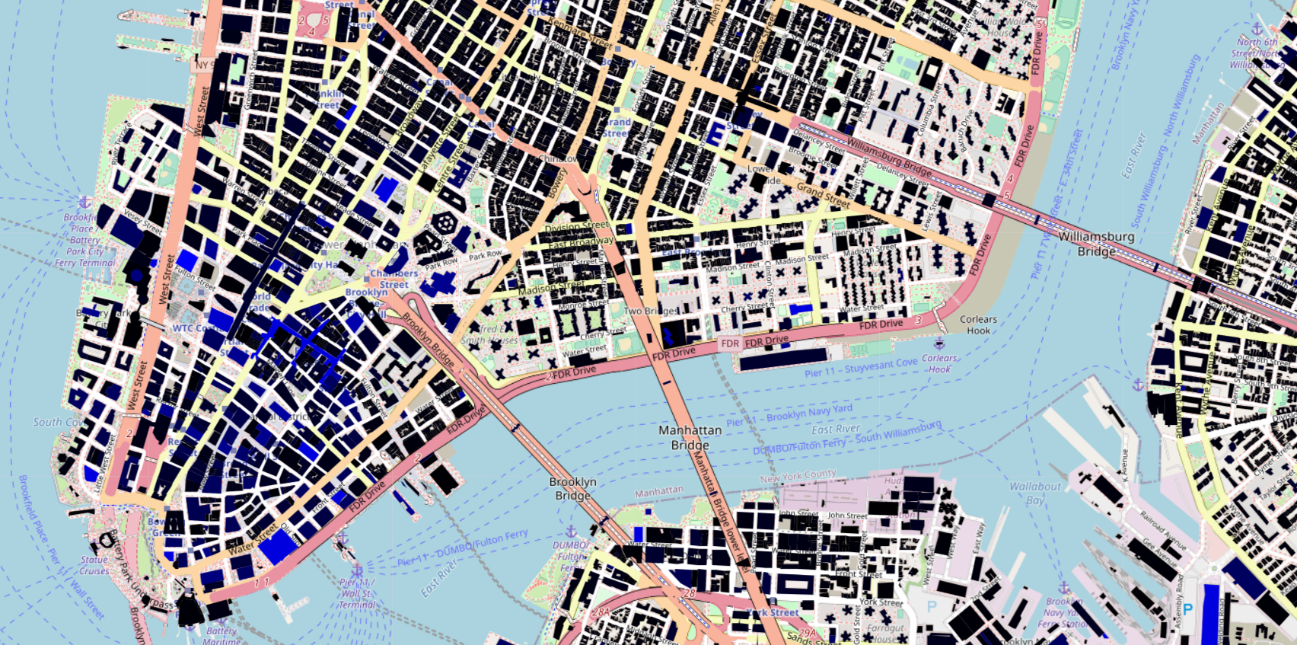
In this project, the environment is implemented as a 2D simulation where the height of the terrain is visually represented by different colors. The simulation provides a top-down view of the landscape, allowing the user to observe and analyze the elevation variations in the simulated area. Different 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.

 [random map generator] [GIS map generator]

A picture containing black and white

Description automatically generated with medium confidence

[map of New York with GIS data on it]

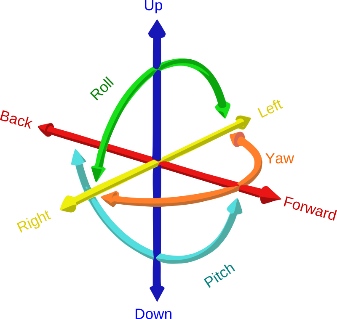
**GUI**

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 our simulation, the agent (drone) has **Six degrees of freedom (6DOF)**. The 6DO 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.

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, as it aims to maximize its reward by conserving battery power while completing the mission successfully.

## 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 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 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.

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.

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-three Q-Learning

# Training results

# Simulation Parameters

### Imported Data

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Calculated value** |
| Orthophoto | A satellite photo imported from the internet using “tile\_Extractoe.py” | photo |
| Map | A Map created by mapbox.com | photo |
| geoJSON | GIS data from open-source | JSON |

### Fixed parameters

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Value** |
| TDB |  |  |
| TDB |  |  |

### 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 |  |

# Software Architecture

## Class Diagram (Unified Modeling Language)

<make new one>

## Environment

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

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

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

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.

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.

## Drone

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.

Getters (GPS\Velocity\name)  
Inputs: None

Outputs: None

Method: Calls GPS to get position or velocity and return it, or return 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

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.

## Internal GPS

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.

## Buttery controller

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

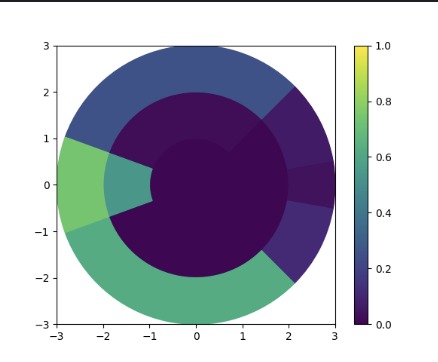
Method: calculate the power consumption of the drone according to its speed vector and its battery mode using a static function called **power consumption**. the ‘calculate battery’ updates the battery capacity and the battery percentage with this calculation:  
every simulation cycle the simulation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

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.



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.

## 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.

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 fairly straightforward.

**Fake Env <in use?>**

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 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.

## Bibliography

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*

https://link.springer.com/article/10.1007/BF00992698

Deep Q-Learning from Demonstrations https://ojs.aaai.org/index.php/AAAI/article/view/11757

Source for base map https://docs.mapbox.com/

Source for orthophoto \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

source for GIS data \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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)