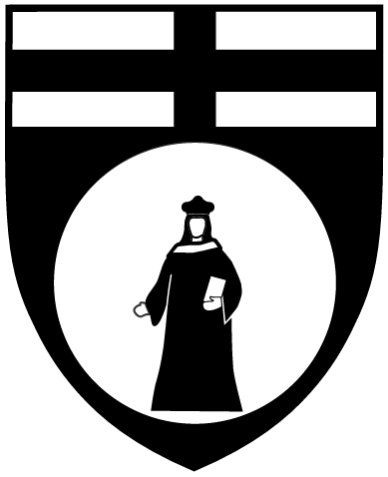


Dibris Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

**Virtual Reality for Robotics - a.y. 2023-2024**

---

**Project**

# Autonomous Drone Swarm Navigation

---

Manuel Delucchi, Matteo Cappellini

UNIVERSITY OF GENOA, JULY 2024

# Contents

# 1 Introduction

In this project, we address the challenge of autonomous drone swarm navigation within a simulated environment, using the A* algorithm for pathfinding. Our work is divided into several phases: creating a voxel map, extracting a 2.5D map and path planning through terrain with obstacles. The goal is to enable drones to move between specified waypoints while taking into account battery constraints and obstacles. This system has the potential to evolve and perform specific tasks, such as seeding and monitoring plants conditions.

# 2 Tools

Link to our Github repository for the complete setup and installation of the project: here.

## 2.1 Colosseum

Colosseum is an open-source simulation platform for testing autonomous systems, specifically drones, in a virtual environment. It provides a high-fidelity simulation environment that accurately models the physics and dynamics of aerial vehicles, as well as a variety of sensor models for testing perception and control algorithms. Using the AirSim Python API we were able to develope a swarm of drones, using Colosseum for control and simulation.

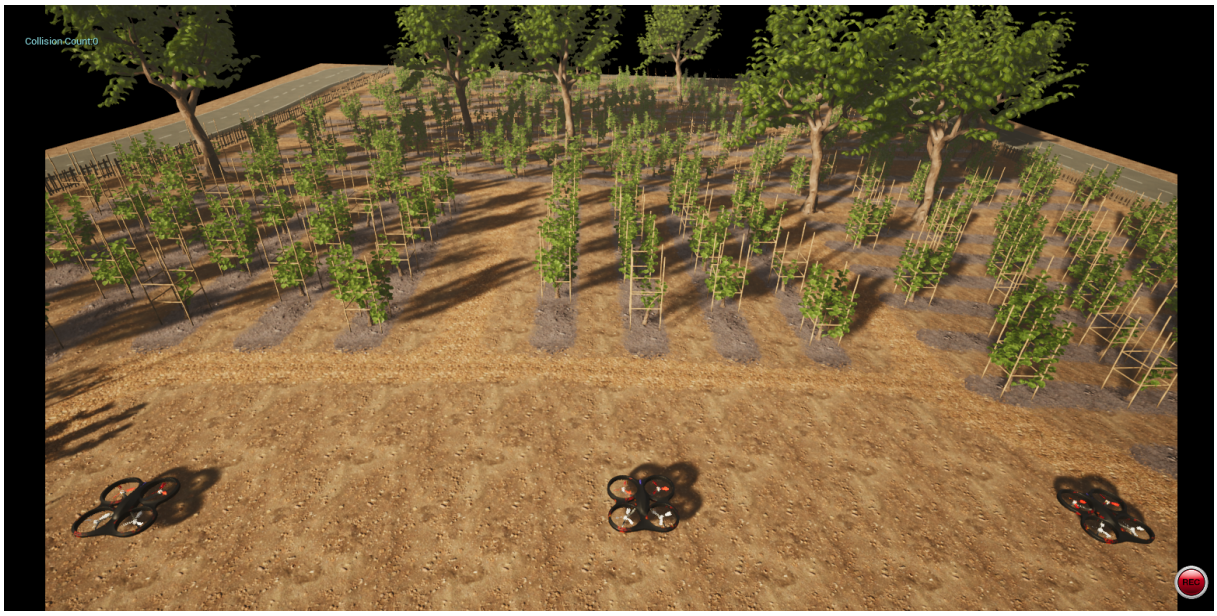

***Note1***: From now on, references to the Colosseum documentation will pertain to the AirSim documentation, as they are the same.
***Note2***: Unreal Engine version 5.2 was used for this project, as it was the most updated version compatible with Colosseum at the time.

## 2.2 Anaconda

The AirSim documentation recommends using Anaconda for efficient set up of Python tools and libraries. To manage multiple drones independently, a main program is executed within the Anaconda environment. Meanwhile, three drone scripts are executed in separate terminal windows as ***subprocesses***. This approach allows more effective control and monitoring of each drone's behavior, while also facilitating debugging.

## 2.3 Unreal Engine Environment

In our project, we aimed to simulate a realistic agricultural environment to test and develop our swarm of drones. We chose a specific portion of the Azienda Agricola Durin, a vineyard located in Ortovero (Albenga, SV), as our simulation environment and we gathered detailed information in order to replicate its layout and to build an accurate heightmap in Unreal Engine.
To bring the environment to life, we used the UltimateFarming plugin that allowed us to add a variety of farming elements, such as crops with and without grapes and different landscape materials, allowing us to replicate soil, dirt and other textures. We then meticulously placed rows of grapevines to mirror the layout and vegetation of the real vineyard.




## 2.4 Voxel Grid Generator

Colosseum provides a feature that constructs ground truth voxel grids directly from Unreal Engine, providing a detailed representation of the environment's occupancy. A voxel grid discretizes the space into cells of a specific size and records whether each cell is occupied or not.

The API call from Python to create a voxel grid follows this structure *simCreateVoxelGrid(self, position, x, y, z, res, of)*:

- ***'position'*** *(Vector3r)*: Global position around which the voxel grid is centered.
- ***'x, y, z'*** *(float)*: Size of each dimension of the voxel grid dimension in meters.
- ***'res'*** *(float)*: Resolution of the voxel grid in meters.
- ***'of'*** *(str)*: Name of the output file to save the voxel grid.

The process iterates over all discretized cells to calculate the occupancy of the environment. This operation can be computationally intensive, depending on the resolution of the cells and the total size of the area being measured. For static environments, it is possible to generate the voxel grid once and reuse it, however, for dynamic environments, it is recommended to generate the voxel grid for a small area around the robot and use it for local planning.
An helpful tool to visualize a created binvox file is viewvox. For our environment, we selected a 56x56x56 voxel grid with a resolution of 1 meter. This choice ensures that the environment from Unreal Engine, when divided into cells, matches perfectly with the size of the voxel map cells, resulting in an accurate representation of the environment's occupancy grid. This also ensures that our drone can align perfectly with the size of each cell (agent radius = 0.5m), facilitating precise navigation and interaction within the simulated environment.

***Note***: A resolution of 0.5 meters would be better and more precise but the number of nodes in the grid greatly increases. Also, as we already mentioned, a resolution of 1 meters ensures the dimensions of the drone and the cells of the Unreal Engine environment to match perfectly.

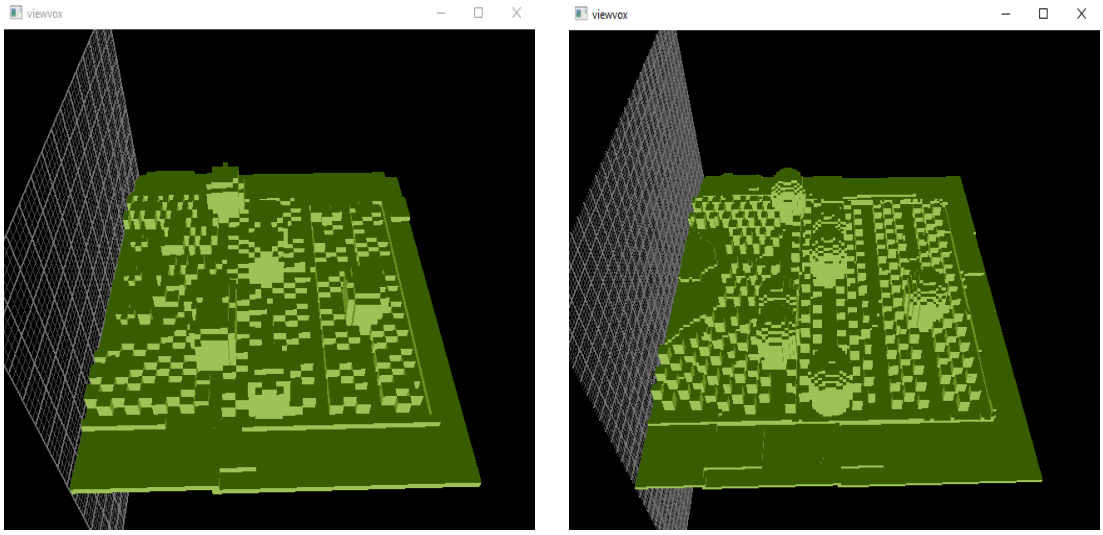



Figure 1: "Voxel Grid Representation of our Vineyard Environment. On the left side the voxel map with resolution 1 meters, while on the right side the voxel map with resolution 0.5 meters."

# 3 Algorithms

Before talking about the algorithms used, let's discuss how we managed the map representation. Initially, the voxel data is loaded from the generated .binvox file using a function that reads the file and returns the voxel data as a 3D NumPy array, representing the volumetric data of the environment. Then, a function is employed to create a 2.5D map by summing the values along the z-axis of the voxel data. This operation collapses the third dimension, resulting in a 2D array where each element represents the cumulative height values of the corresponding grid cell. Finally, the resulting 2.5D map is saved to a CSV file. This process ensures that the spatial information of the environment is preserved in a format suitable for further processing and analysis. Additionally, the map data is divided into three smaller maps. This division facilitates the assignment of specific areas to each drone, ensuring efficient coverage and management of the entire environment.

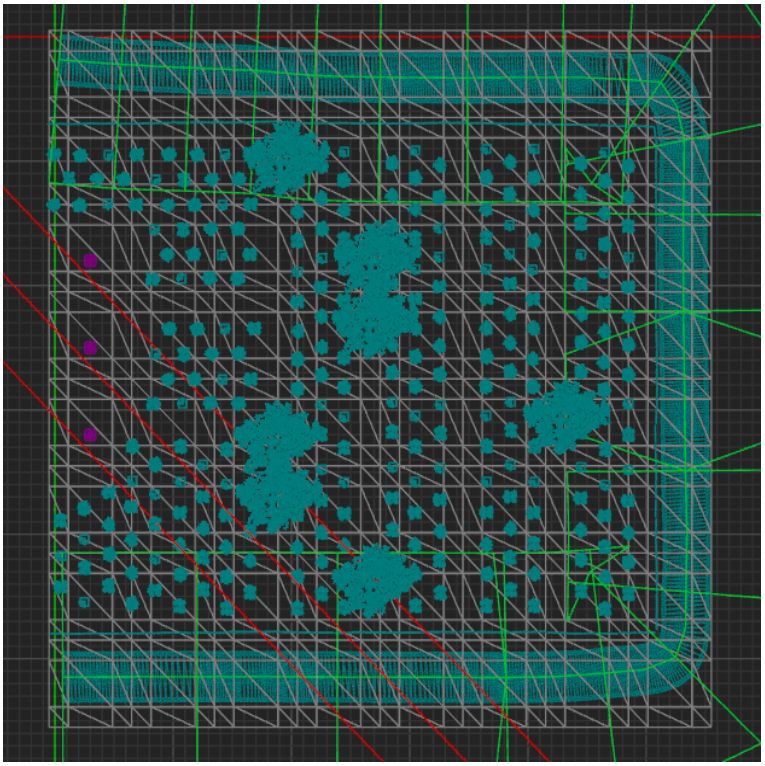


Figure 2: "Top view of the Unreal Engine map. The red horizontal axis represents the x-axis, and the green vertical axis represents the y-axis. The three purple objects are the drones."

As shown in the map in Fig. 2, we have a 56x56 grid within the unreal engine environment. This grid is divided into three subgrids:

1. ***map1***: $y \leq 20$
2. ***map2***: $21 \leq y \leq 31$

3. ***map3***: y $\geq$ 32

To enhance scalability, we decided to work with global coordinates. This required setting the player start position to coincide with the cell (0,0) because the global (0,0) is relative to where the player start is set.

Using the ***settings.json*** file, each drone is spawned at its designated position within Unreal Engine relative to the player start location (as shown in Fig. 2):

1. ***drone1***: X = 3, Y = 18
2. ***drone2***: X = 3, Y = 25
3. ***drone3***: X = 3, Y = 32

However, the issue arises because each drone's local coordinates are set to (0,0) upon spawning. To address this, we needed to convert the local coordinates to global coordinates. This conversion was done by subtracting the starting position of each drones to its current coordinates.

***Note***: All AirSim API uses **NED** (Nord East Down) coordinate system, i.e., +X is North, +Y is East and +Z is Down. All units are in SI system. Notice that this is different from coordinate system used internally by Unreal Engine. In Unreal Engine, +Z is upwards and length unit is in centimeters instead of meters. AirSim APIs takes care of the appropriate conversions. The starting point of the vehicle is always coordinates (0, 0, 0) in NED system.

## 3.1 Finite State Machine

The control system for the drones is implemented in Python, using the AirSim API. Each drone follows a defined sequence of states to achieve its navigation objectives:

1. ***INIT***:
   (a) The drone establishes a connection with the AirSim simulator.
   (b) API control is enabled, allowing the Python script to control the drone.
   (c) The drone is armed, making it ready for takeoff.
2. ***TAKEOFF***:
   (a) The drone takes off from its current position.
   (b) It ascends to a predefined target altitude, ensuring it hovers at a safe height above the ground.
3. ***NAVIGATE***:
   (a) The drone begins its journey towards the specified waypoints.
   (b) It calculates the optimal path using the A* algorithm, taking into account obstacle avoidance and battery conservation.

(c) The drone moves along the calculated path, adjusting its altitude relative to the terrain height to maintain a constant flying height.

(d) The drone continuously monitors its battery status. If the battery level falls below a threshold, it transitions to the $RETURN\_HOME$ state.

***Note***: The ***moveOnPathAsync*** function plays a key role in guiding the drone along a predefined trajectory defined by a list of waypoints. It facilitates smooth traversal of these waypoints, ensuring precise navigation for the drone.

4. ***RETURN_HOME***:

   (a) When instructed or when the battery is low, the drone calculates the path back to its home position using the A* algorithm.

   (b) It follows this path, ensuring it navigates around obstacles and conserves enough battery to reach home.

   (c) If the path to home is successfully completed, the drone transitions to the $LAND$ state.

   (d) If the path is not feasible due to critical battery levels, it transitions to the $EMERGENCY\_LAND$ state.

5. ***LAND***:

   (a) The drone descends gradually to land at its home position.

   (b) Once on the ground, it disarms and disables API control, completing its mission safely.

6. ***EMERGENCY_LAND***:

   (a) If the battery level is critically low and the drone cannot return home, it initiates an emergency landing procedure.

   (b) The drone lands immediately at its current position to prevent a crash due to battery depletion.

This state machine ensures that the drones operate autonomously, efficiently navigating through waypoints, avoiding obstacles and managing their battery levels effectively. The inclusion of an emergency landing state adds an extra layer of safety, allowing drones to handle unexpected situations such as critical battery levels.

## 3.2 A* algorithm

The A* algorithm is a popular and efficient path-finding and graph-traversal algorithm, particularly notable for its efficiency in finding the shortest path between multiple nodes in a graph.

### 3.2.1 How Does It Work?

Consider a grid populated with various obstacles. Given a starting node and a goal node, the objective is to reach the destination as quickly as possible. The A* algorithm operates by evaluating nodes based on a function called ***f(n)***. At each step, it selects the node with the lowest f(n) value.

$$f(n) = g(n) + h(n)$$

Where:

1. ***g(n)*** is the cost to move from the starting point to a given square on the grid, following the path generated to reach that point.
2. ***h(n)*** is the estimated cost to move from that specific square on the grid to the final destination. This is often referred to as the heuristic, which is an intelligent guess about the distance remaining.

***Note***: In our project, the values are defined as: ***f(n)*** = *priority*, ***g(n)*** = *new_cost*, ***h(n)*** = *heuristic(next_node, goal)*

### 3.2.2 Algorithm Execution

In the grid in Fig. 3, the A* algorithm starts at the initial position (blue node) and examines all adjacent cells. Once the list of adjacent cells has been populated, it filters out those which are inaccessible (obstacles, marked in red). Then, it picks the cell with the lowest estimated f(n) value. This process is recursively repeated until the shortest path (highlighted in green) to the target (star) is found.

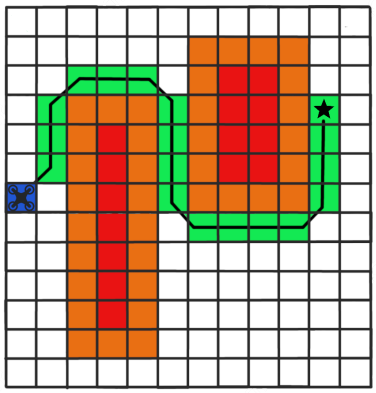


Figure 3: "Graphic visualization of the A* Algorithm"

For the calculation of ***g(n)***, we construct a weighted graph of the map, starting from the generated map. Each cell is assigned a cost and the calculation for moving from one node to the next is defined as:

$$\text{new\_cost} = \text{cost\_so\_far[current]} + \text{move\_cost} + \text{height\_diff}$$

Where:

- ***cost_ so_ far[current]***: The accumulated cost to reach the current node from the starting point.
- ***move_ cost***: The cost of movement from the current node to the next node, which is either 1 for an horizontal/vertical movement or $\sqrt{2}$ for a diagonal movement.
- ***height_ diff***: The absolute height difference between the current node and the next one, representing the additional cost derived from changes in terrain elevation.

The heuristic ***h(n)*** is calculated using the ***Manhattan distance***, which measures the total number of squares moved horizontally, vertically and diagonally to reach the target node from the current node n.

$$h = |x_{start} - x_{goal}| + |y_{start} - y_{goal}|$$

Instead of the Manhattan distance, the Alternatively, the ***Euclidean distance*** can be used for potentially greater accuracy. If we run both heuristics on the same maze, the Euclidean pathfinder favors a path along a straight line. This approach is more accurate but also slower because it explores a larger area to find the path.

$$h = \sqrt{(x_{start} - x_{goal})^2 + (y_{start} - y_{goal})^2}$$

### 3.2.3 Obstacle Avoidance

The A* algorithm has a crucial feature: its inherent obstacle avoidance capability. Given that the vineyard is on an almost flat surface, we can assume that any cell in our environment with an height value exceeding a manually defined threshold can be labeled as an obstacle. These cells are excluded when determining viable paths for the drones. Additionally, cells neighboring these obstacles (highlighted in orange in Fig. 3) are also treated as obstacles to account for potential inaccuracies in polygon edge recognition due voxel map's limited resolution. This precaution ensures drones do not get trapped in corners, thereby enhancing the overall reliability of the pathfinding process.

### 3.2.4 Altitude Control

Here, the 2.5D map plays a crucial role in our navigation system. Specifically, we set the target altitude ***'z'*** to -4 meters in NED (North-East-Down) coordinates. By subtracting the current terrain height of the cell from this target altitude, the drone adjusts its flight altitude to ensure a consistent elevation of -4 meters above the terrain surface.

### 3.2.5 Battery Consumption

Battery management is a critical aspect of the drone control system, ensuring that each drone can complete its assigned tasks and return safely to its home base. The system uses a dynamic battery consumption model that takes into account the distance traveled and the altitude difference between consecutive grid cells.

The A* algorithm incorporates battery optimization by continuously monitoring the battery level during path planning. The battery level is scaled by a factor of 0.5 to ensure it does not deplete too quickly. The optimization is performed according to the following formula:

$$\text{new\_battery\_level} = \text{battery\_level[current]} - 0.5 * (\text{move\_cost} + \text{height\_diff})$$

Before finalizing the path, the algorithm ensures that the battery level is sufficient to reach the goal. This is essential for maintaining operational efficiency and preventing mid-flight power failures. If the battery level falls below a critical threshold, the algorithm triggers appropriate actions:

- If the battery level is insufficient to reach the next waypoint or goal, the drone initiates a ***RETURN_HOME*** state to return to the home base for recharging.
- If the battery level is critically low and the drone cannot return home safely, it initiates an ***EMERGENCY_LAND*** state, landing at the current location and generating a failure message.

# 4 Crop Seeding

Planting seeds is a critical factor in producing the perfect harvest, making it essential to consider using crop seeding drones for efficient seed distribution across fields. In our setup, when a drone reaches a designated waypoint, the DroneController triggers a simple Blueprint event in Unreal Engine's *'Level Blueprint'* that triggers an event to drop a seed precisely at that location. This event activates a specialized Blueprint (*BP_Seed*) equipped with a projectile movement component, allowing the drones to drop seeds precisely at the waypoint location.

In the provided Unreal Engine Blueprint in Fig. 4, we have three custom events that are used to trigger seed drops by three different drones (***BP_Flying_Pawn***). For each event, an ***ExecuteConsoleCommand*** node is used to trigger the respective ***SeedDrop*** custom events. For each event, the ***GetAllActorsOfClass*** node is used to retrieve all instances of the drone class. After obtaining the list of actors, the location of the specific drone that is supposed to drop the seed is fetched via index. Finally, the ***SpawnTransform*** is set to the drone's location and the ***SpawnActorBPSeed*** node is used to spawn the seed actor at the location of the selected drone.

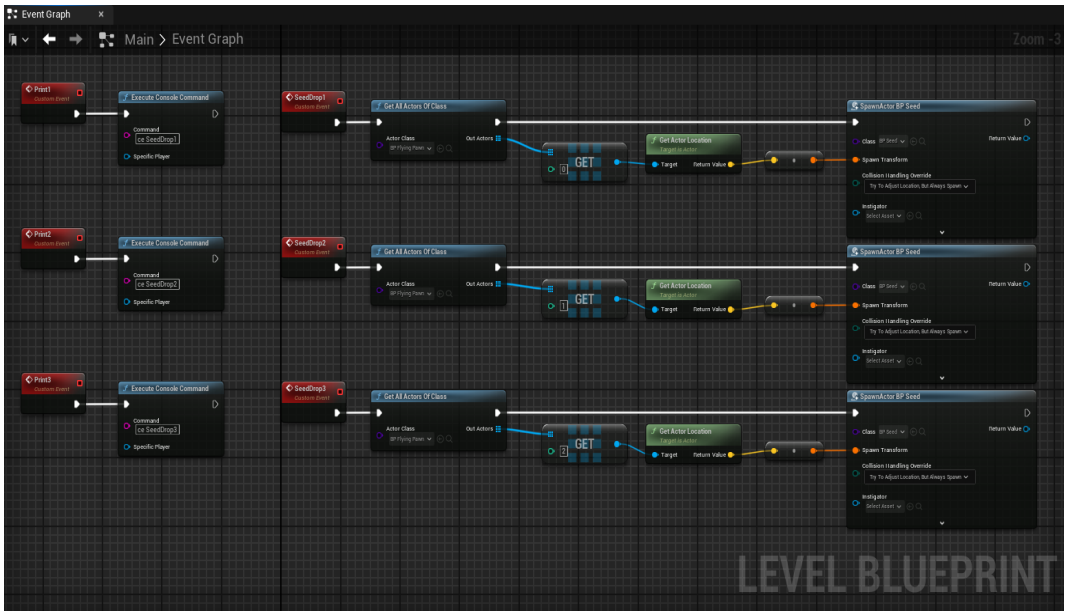



Figure 4: "Seed dropping blueprint"

# 5 Conclusions

## 5.1 Challenges

***Coordinate System***: One of the primary challenges we faced was related to the coordinate system. Drones are spawned relative to the position of the Player Start. Consequently, if the Player Start is not placed at the (0, 0) position of the environment, it becomes difficult to convert coordinates to a global reference frame accurately. We attempted to retrieve the position of the Player Start to address this issue but were unsuccessful. Additionally, it appears that the only reliable method to obtain the exact position of the drones relative to the Player Start is through the *'simGetObjectPose'* function. Other, more intuitive functions, such as *'simGetVehiclePose'*, consistently return the local coordinates of the drones, which are always (0, 0) relative to their own reference frame.

## 5.2 Possible Improvements

1. ***Advanced Obstacle Avoidance***: Implementing advanced obstacle avoidance techniques using camera-based sensing can significantly imporve the drones' navigation capabilities. By integrating real-time visual data, drones can detect and avoid obstacles more effectively, ensuring safer and more efficient pathfinding. This approach also eliminates the need to hardcode a threshold for considering defining obstacles. While our current implementation performs well in a relatively obstacle-free environment, these advanced techniques would offer greater flexibility and robustness in more complex settings.
2. ***Integration of GPS Coordinates***: Incorporating GPS coordinates into the navigation system can

provide more accurate and reliable position data for the drones, especially if used with a voxel map with higher resolution. By using GPS, the drones can navigate over larger areas with precise waypoint tracking and location awareness. This improvement would enable tasks such as surveying expansive agricultural fields or conducting search and rescue operations with improved spatial accuracy and efficiency.

3. ***ROS (Robot Operating System)***: Transitioning to ROS from Windows-based Python environments can offer numerous benefits. ROS provides a standardized framework for robotic applications, offering robust tools and libraries for tasks such as sensor integration, path planning and communication between multiple drones. This migration can enhance the scalability, modularity and interoperability of the drone swarm system, facilitating easier integration of new features and technologies as the project evolves.