Redes e Sistemas Autónomos

Final project report

Bruno Gomes, Diogo Silva

Universidade de Aveiro

Redes e Sistemas Autónomos

DEPARTAMENTO DE ELETRÓNICA
TELECOMUNICAÇÕES E INFORMÁTICA

Final project report

Bruno Gomes, Diogo Silva
(103320) brunofgomes@ua.pt, (104341) diogobranco.as@ua.pt

17/06/2025

Abstract

In this report, will talk about the steps taken to develop our cooperative drone fire monitoring project for detection of wildfires. In this project, the drones are running in a simulated environment with the purpose of working collaboratively to cover large areas of forest regions in an efficient way. The main idea is having an intelligent drone which already has a trained model to detect forest fires, upon detecting a fire the drone can request assistance from other drones to increase the area of surveillance. Our solution implements different recon strategies depending on the mission that is being carried out, we developed strategies that use spiral and grid-based search patterns and a more dynamic strategy that uses Voronoi partitions for effective area coverage. The simulator used for this project was the "fleet-manager" simulator, *QGroundControl* was used for mission monitoring and we used an already trained YOLOv5 model to detect the fires. The project effectively shows the potential of autonomous drone networks in supporting rapid response to forest fires and minimizing their spread.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Motivation

Wildfires pose a serious environmental and economic threat, especially in areas with high forest density. Early detection and rapid area coverage are critical to preventing widespread fire detection. Autonomous drones offer an efficient and effective solution to constantly monitor high risk areas.

1.2 Project Objective

The main objective of this project is to design, simulate, and evaluate a network of autonomous drones capable of detecting and coordinate themselves to provide an increased area of coverage to help detect other points of interest, this is done using a pre-trained model to detect fires.

1.3 Report Overview

In the following chapters we will get into the details of how we implemented each drone mission and how we integrated the computer vision algorithm to detect the fires.

Chapter 2

System Architecture

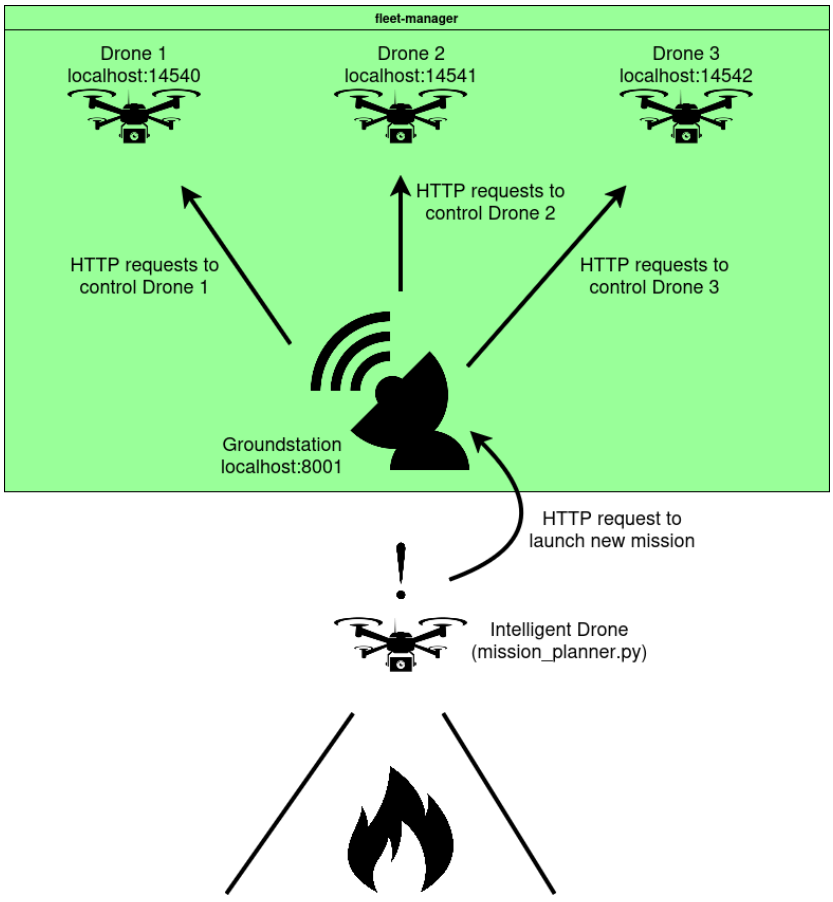

Figure 2.1: General system architecture

To start with, it is important to note that due to simulation purposes we could not actually put the drone that contained the fire detection algorithm running inside the simulator, in reality the intelligent drone is just a python script that generates missions for the simulated drones that are actually running inside the fleet-manager simulator. Additionally, as seen on figure 2.1 the drones never communicate directly between themselves they always use the groundstation as a mediator.

The flow of the system starts by the computer vision algorithm being called by the "mission_planner.py" script, the algorithm will then analyze a video which contains aerial images of forest fires, if a fire is detected a mission will be generated by the "mission_planner.py" script, the mission that will be generated will depend on the user's input. We developed 3 missions that involve different drone coordination strategies, we will get into detail for each mission on chapter 3.

The sections of code that are responsible for triggering and launching the missions are the following:

- **Detection algorithm**

```
...
if not fire_alert_sent and names[c].lower() == "fire":
    fire_alert_sent = True
    LOGGER.info("Fire detected! Sending alert to
        groundstation...")
...
```

Listing 2.1: detect.py script section that activates a flag when a fire is detected

The "**fire_alert_sent**" is a boolean variable that only gets set when a fire is detected in the running video. The print "**FIRE_DETECTED**" will appear on the terminal if there is a fire. The names list contains the name of the objects detected in the input video.

- **Detect fire function**

```
def detect_fire(self, video_source):
    ...
    if "FIRE_DETECTED" in result.stdout.upper():
        self.log("Fire detected in the video")
        return True
    return False
    ...
```

Listing 2.2: detect fire function from the mission_planner.py script

This function is responsible to call the "detect.py" script and pass the input video to that algorithm as an argument, if the output of the the script contains the string "**FIRE_DETECTED**", the function `detect_fire()` will return "True" and then a mission can be generated and sent to the groundstation.

- **Mission Upload function**

```
def upload_mission(self, mission_script, mission_type):
    mission_path = f"mission{mission_type}_temp.groovy"

    with open(mission_path, "w") as f:
        f.write(mission_script)
    with open(mission_path, "rb") as f:
        file_content = f.read()
        headers = {"Accept": "application/json"}
        mission_response = requests.post(f"{self.gs_url
            }/mission", data=file_content, headers=
            headers)
        if mission_response.ok:
            print("Mission launched:", mission_response.
                text)
        else:
            print(f"Mission launch failed: {
                mission_response.status_code} {
                mission_response.text}")
```

Listing 2.3: upload mission function from the mission_planner.py script

This function is responsible for sending the generated missions to the groundstation, it starts by writing the mission script that was passed as an argument to a temporary groovy file, after that the file is read in binary mode and a HTTP POST request is sent to the groundstation using the endpoint "`http://localhost:8001/mission`" with the groovy file sent in the request body. Finally, the groundstation responds, and the script prints whether the mission was launched successfully.

All things considered, the "mission_planner.py" script serves as the central component of the entire system. It is responsible for both detecting fires and generating the corresponding missions. Missions are only triggered if a fire is detected, and the generated mission data is sent to the groundstation, which then coordinates the drones accordingly. The drone movements and mission execution can be monitored through the *QGroundControl* software.

Chapter 3

Drone Missions

3.1 Mission 1 - Fire Detection and Dynamic Drone Dispatch

This mission represents a scenario where a fire is detected by the "intelligent drone", which then sends a mission to the groundstation to deploy another drone near the fire location. The secondary drone expands the reconnaissance area by spiraling around the fire's coordinates.

3.1.1 Implementation

Steps taken to implement mission 1:

1. **Fire detection**
    - The mission planner starts by calling the `detect_fire()` function which calls a YOLOv5-based fire detection model which is executed by the "detect.py" script;
    - If a fire is detected in the input video, mission 1 will be generated,
    - The detected fire coordinates are hard-coded on the "mission_planner.py" script since we could not generate coordinates from a video capture.

2. **Spiral Waypoint Generation**

```
# spiral parameters
num_loops = 4          # number of spiral loops
points_per_loop = 8    # waypoints per loop
spacing_m = 20         # distance between loops (meters)

waypoints = []
for loop in range(1, num_loops + 1):
    radius = loop * spacing_m  # each loop wil be
        further out
    for i in range(points_per_loop):
        angle = 2 * math.pi * i / points_per_loop
        dlat = (radius * math.cos(angle)) / 111320
        dlon = (radius * math.sin(angle)) / (111320 *
            math.cos(math.radians(detected_lat)))
        wp_lat = detected_lat + dlat
        wp_lon = detected_lon + dlon
        waypoints.append((wp_lat, wp_lon))
```

Listing 3.1: Spiral waypoint generation code

- After the fire is detected (`detected_lat` and `detected_lon`), the coordinates of it are set as the center of the spiral;
- The code generates a series of waypoints arranged in concentric loops around the fire location;
- Each loop increases its radius by a fixed distance (`spacing_m`), and each loop contains a fixed number of waypoints (`points_per_loop`), evenly distributed by an angle that is calculated using this mathematical formula ($\text{angle} = \frac{2\pi i}{\text{points\_per\_loop}}$);
- The latitude and longitude of each waypoint are calculated using basic trigonometry and converted from meters to degrees using these formulas:
  - $\text{dlat} = \frac{\text{radius} \cdot \cos(\text{angle})}{111320}$
  - $\text{dlon} = \frac{\text{radius} \cdot \sin(\text{angle})}{111320 \cdot \cos(\text{radians}(\text{detected\_lat}))}$
  - $\text{radius} \cdot \cos(\text{angle})$ gives the X offset in meters (east-west) from the center of the fire coordinates.
  - $\text{radius} \cdot \sin(\text{angle})$ gives the Y offset in meters (north-south) from the center of the fire coordinates.
  - `dlat` represents the change in latitude, converted from meters to degrees.
  - `dlon` represents the change in longitude, converted from meters to degrees.
  - We use the value `111320` because it is the approximate number of meters in one degree of latitude on Earth. This allows us to

convert distances in meters to degrees of latitude or longitude when calculating geographic coordinates for waypoints.

- The offset of the waypoints is calculated by adding `dlat` and `dlon` to the coordinates of the fire center;
- Finally, the waypoints get added to a list of waypoints that the drone will have to traverse when executing the mission;
- If the coordinates of the fire change this code will still work since all the calculations are based on the current coordinates of the fire;
- The number of loops (`num_loops`), waypoints per loop (`points_per_loop`), and spacing (`spacing_m`) can be easily adjusted to change the density and extent of the spiral, making the approach adaptable to different fire sizes or drone capabilities.

3. **Mission Script Generation**

- The generated missions are formatted into a Groovy mission script;
- The script instructs the drone to:
  - Arm and take off to a safe altitude;
  - Visit each waypoint in the spiral pattern;
  - Return to base after finishing the recon.

```
#generate mission script
waypoints_groovy = ",\n    ".join(
    [f"[lat: {lat:.6f}, lon: {lon:.6f}]" for lat,
        lon in waypoints]
)
mission_script = textwrap.dedent(f"""\
drone = assign 'drone01'
arm drone
takeoff drone, 5.meters
takeoff_alt = drone.position.alt
waypoints = [
    {waypoints_groovy}
]
for (wp in waypoints) {{
    move drone, lat: wp.lat, lon: wp.lon, alt:
        takeoff_alt
}}
home drone
""")
```

Listing 3.2: Mission 1 groovy script

4. **Mission Upload**

   - The mission script is written to a temporary Groovy file;
   - The file is uploaded to the groundstation via an HTTP POST request to the `"http://localhost:8001/mission"` endpoint;
   - The groundstation receives and executes the mission on the assigned drone.

3.2 Mission 2 - Perimeter Surveillance and Fire Containment Monitoring

This mission represents a scenario where a fire has already been detected by the "intelligent drone" and 1 or 2 drones are sent to patrol a square area drawn from the epicenter of the square. Eventually one of the drones patrolling the perimeter will "detect" a new fire and the groundstation, upon receiving that information, will cancel the current patrol mission and create a new one with a bigger area.

The main intention for this mission would be to have drones dynamically draw a "danger zone" autonomously that adapts to new fires with the intention of helping fire prevention forces.

3.2.1 Implementation

1. **Helper functions:**

   Before we start describing the mission code itself, we would like to explain some of the functions that we created to help us more dynamically communicate between the groundstation and the drones.

   - **get_available_drones(self):**
     In this function we make an HTTP GET request to the groundstation endpoint: **{gs_url}/drone**. This let's us collect information from all the drones in the simulation. From there we take the state of the drone and if it is 'ready' or 'readyinair' (this is important because when we cancel the first mission their status is not ready but readyinair) we get their droneId.

     This way we can always know the drones we have available for new missions and do not need to hardcode their id's when creating the mission script.
   - **stop_mission(self, mission_id):**
     We use this to make an HTTP DELETE request to the groundstation endpoint: **{gs_url}/mission/{mission_id}**. With this we can issue a mission cancel order to all drones performing the perimeter patrol upon a new fire being detected on it's edge.

- **get_drone_mission(self,drone_id):**
  This script let's us know the mission a current drone has been assigned to by making an HTTP GET request to the groundstation endpoint: **{gs_url}/drone/{drone_id}?data=info**.
  We use this to get the name of the mission the drone that detects the fire was performing so that we can call the previously explained **stop_mission**
- **get_drones_postions(self):**
  Finally, we use this last helper function to make a request to **{gs_url}/drone?data=telem** to obtain all of the drones id's and their respective coordinates. We use this to simulate the detection of a fire and it's position from the drones sensors, in a real life scenario we would use something similar but also with the data from the drones camera telling if a fire had been detected.

2. **Square waypoint generation**

```
    def generate_mission_2(self, fire_lat, fire_lon,
        side_m, is_restart=False):
        all_drones = self.get_available_drones()
        if not all_drones:
            print("No available drones found. Exiting
                mission generation.")
            return None
        drones = all_drones[:2]  # Take first 2 drones

        half_side = side_m / 2
        delta_lat = half_side / 111320
        delta_lon = half_side / (111320 * math.cos(math.
            radians(fire_lat)))

        corners = [
            (fire_lat + delta_lat, fire_lon - delta_lon)
                ,  # NW
            (fire_lat + delta_lat, fire_lon + delta_lon)
                ,  # NE
            (fire_lat - delta_lat, fire_lon + delta_lon)
                ,  # SE
            (fire_lat - delta_lat, fire_lon - delta_lon)
                ,  # SW
        ]
        if len(drones) == 2:
            # Split perimeter between 2 drones (half
                each)
            drone1_waypoints = [corners[0], corners[1],
                corners[2]]
            drone2_waypoints = [corners[2], corners[3],
                corners[0]]
            drone_paths = {
                drones[0]: drone1_waypoints,
                drones[1]: drone2_waypoints
            }
        else:
            # Single drone: Full perimeter patrol
            drone_paths = {drones[0]: corners + [corners
                [0]]}
```

Listing 3.3: Mission 2 Square perimeter waypoints

The main algorithm that mission 2 uses is the one responsible for creating waypoints from the fire epicenter that together make a square.

In our code we first use the previously mention function to get the available drones and get the first two. We then take use the longitude and latitude coordinates of the fire to get the offsets for each of the corners of the square we will use. We used 111320 as the number of meters in a degree

of latitude, for longitude it varies depending on how close we are to the poles so we used:

$$\text{Meters per degree of longitude} = 111320 \times \cos(\text{latitude in radians})$$

After getting the offsets we use them together with the fire coordinates to generate each corner of the square.

Then if we only have one drone available we will have him do the full square perimeter. If we have 2, we will divide the square in 2 sides and give one of each to the drones:

(drone0: NW → NE → SE, drone1: SE → SW → NW).

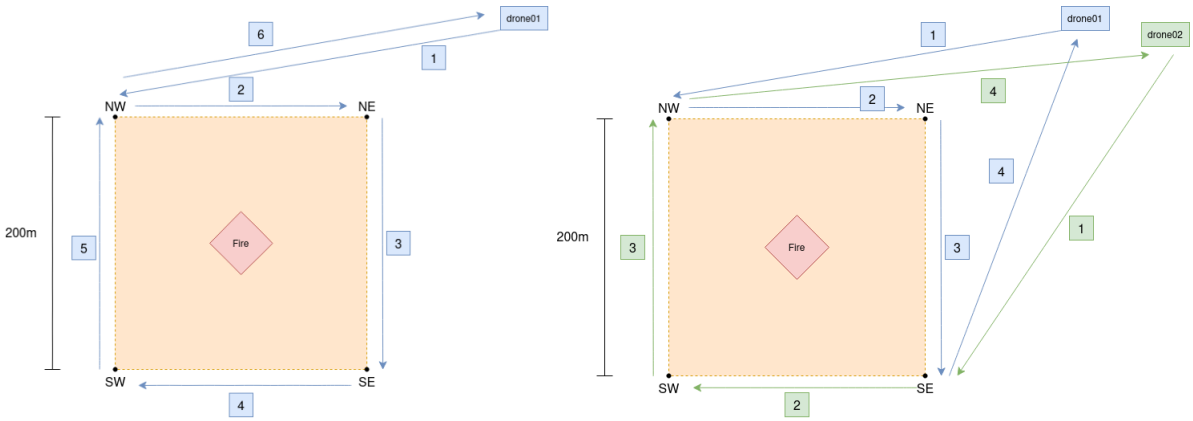

Figure 3.1: Perimeter waypoint generation with 1 and 2 drones with a square side size of 200m

3. **Mission Script generation:**

```
      /*
    * PARALLEL PERIMETER PATROL (MAX 2 DRONES)
    * Drones split perimeter coverage for efficiency and
        adapt to new fire coordinates.
    */
    drone01 = assign 'drone01'
    drone02 = assign 'drone02'
    def missions = run(
    {
                    arm drone01
                    takeoff drone01, 5.meters
                    takeoff_alt = drone01.position.alt
                    waypoints = [
                        [lat: 40.679832, lon:
                            -8.723907],
                        [lat: 40.679832, lon:
                            -8.721538],
                        [lat: 40.678036, lon: -8.721538]
                    ]
                    for (wp in waypoints) {
                        move drone01, lat: wp.lat, lon:
                            wp.lon, alt: takeoff_alt
                    }
                    home drone01
                },
    {
                    arm drone02
                    takeoff drone02, 5.meters
                    takeoff_alt = drone02.position.alt
                    waypoints = [
                        [lat: 40.678036, lon:
                            -8.721538],
                [lat: 40.678036, lon: -8.723907],
                [lat: 40.679832, lon: -8.723907]
                    ]
                    for (wp in waypoints) {
                        move drone02, lat: wp.lat, lon:
                            wp.lon, alt: takeoff_alt
                    }
                    home drone02
                }
)
    wait missions[0]
wait missions[1]

    """
```

Listing 3.4: Mission 2 groovy script

This is an example of the script we generate in a situation where drone01 and drone02 were the first 2 available drones found. The groovy script is set in a way that let's the groundstation issue mission commands for both drones at the same time instead of waiting for the first one to end and then start the other. This way we can cover the same perimeter in half the time one drone would take.

in the case that we were issuing a similair patrol mission after having already issued one before in the same sortie (the situation where a new fire is detected), the script generated will not have the arm and takeoff commands since the drone(s) will already be in air.

4. **Mission execution:**

   The general flow of mission 2 is as follows:

   - **Yolo fire detection** is ran as to simulate a fire being detected
   - **Initial mission generation**:

     A new mission is generated with the fire coordinates 40.678933 lat, -8.722722 lon and 200m for the sides of the square.
   - **Monitoring loop:**

     As the drone(s) execute the mission the groundstation will periodically check their position each 5 seconds, if the drones are near the second fire coordinate (40.678465 lat, -8.723905), the yolo fire detection algorithm is ran again as to simulate the drones sensors detecting the fire.
   - **Updated mission generation:**

     At this point, we get the mission the drone that detected the fire was executing and the groundstation issues an order to all drones in that mission to cancel and gives it the new generates a new mission with the epicenter of the fire based on the new one that was detected, duplicating the sides of the square perimeter (in our case it would be 400m) and setting the restart variable to true as to not generate the arm and takeoff commands in the groovy script.
   - **Mission end**

     At this point the drones will patrol the new perimeter and after not finding any fires return home ending the Mission 2.

3.3 Mission 3 - Drone Swarm Mapping after Fire Detection (Post-Fire Assessment)

This mission is intended to represent a situation where the drones do an assessment to evaluate the damage that was done and the area of forest that was burnt after the fire was extinguished. It aims to give authorities a more clear view of what needs to be done (reforestation strategies, house rebuilding, etc.) therefore making it easier to take action quicker and plan ahead for the future.

There are two variants of this mission, if the drones start close to each other a static partitioning path will be assigned to each drone and if they are in distinct parts of the map each drone will be assigned a more dynamic path based on their current coordinates. Each path assignment technique will be explained with further detail on the section below.

It is important to note that neither of the simulated drones have computer-vision algorithms to check burnt areas, as we could not integrate these algorithms in the simulated drones. This mission focuses more on exploring proper recon paths to implement on drones to cover the damaged areas in an efficient way.

3.3.1 Implementation

Steps taken to implement mission 3:

1. **Fire detection**
   - The mission beginning is the same, it starts by running a computer vision algorithm on the provided video source to detect fire.
   - If the fire is detected mission 3 will be created;
2. **Grid Waypoint Generation**
   - When the fire is detected, the "mission_planner.py" script generates a square grid of waypoints centered on the fire coordinates;
   - The grid size (`grid_size`) and spacing (`spacing_m`) between waypoints are parameters that can be changed according to the mission requirements;
   - The latitude and longitude of each grid cell is calculated using the following formulas:
     - $\text{delta\_lat} = \frac{\text{spacing\_m}}{111320}$
     - $\text{delta\_lon} = \frac{\text{spacing\_m}}{111320 \cdot \cos(\text{latitude\_in\_radians})}$
     - These formulas convert a distance in meters to degrees of latitude and longitude, accounting for the Earth's curvature.
   - `waypoints_grid` (2D Grid): The 2D grid structure is essential for static partitioning, where each drone is assigned specific rows of the grid. This makes it easy to allocate contiguous areas to each drone in a structured and balanced way;

- `waypoints` (flat list): The flat list will store all waypoint coordinates of the generated grid.

```
grid_size = 5 # 5x5 grid of waypoints around the
    fire
spacing_m = 40 # meters between waypoints
delta_lat = spacing_m / 111320
delta_lon = spacing_m / (111320 * math.cos(math.
    radians(detected_lat)))
# build grid of waypoints
waypoints_grid = []
waypoints = []
offset = grid_size // 2
for z in range(-offset, offset + 1):
    row = []
    for m in range(-offset, offset + 1):
        if z == 0 and m == 0:
            continue
        wp_lat = detected_lat + z * delta_lat
        wp_lon = detected_lon + m * delta_lon
        row.append((wp_lat, wp_lon))
        waypoints.append((wp_lat, wp_lon))
    waypoints_grid.append(row)
```

Listing 3.5: Grid waypoint generation code

3. **Adaptive Partitioning Strategy**

- The system retrieves the current positions of all available drones using the GET method on the endpoint `http://localhost:8001/drone`;
- Then it calculates the distance (in meters) between drones using the Haversine formula which is used to compute the distance between two points on the Earth's surface;
- The `"R"` variable corresponds to the Earth's estimated radius in meters;

```
def haversine(lat1, lon1, lat2, lon2): # returns distance in
     meters between two lat/lon points
    R = 6371000
    phi1, phi2 = np.radians(lat1), np.radians(lat2)
    dphi = np.radians(lat2 - lat1)
    dlambda = np.radians(lon2 - lon1)
    a = np.sin(dphi/2)**2 + np.cos(phi1)*np.cos(phi2)*np.sin
        (dlambda/2)**2
    return 2*R*np.arcsin(np.sqrt(a))
```

Listing 3.6: Haversine calculation function

- If all drones are close together (distance below a threshold), static row-based partitioning is used, which means that each drone will be assigned specific rows of the grid;
- In this case, drone 0 will patrol row 0 and 3, drone 1 will patrol row 1 and 4, drone 2 will patrol row 2;
- The `drone_waypoints` dictionary is used to flatten the assigned rows into a single list of waypoints, in this way it is easier to traverse the list;

```
if max_dist < DIST_THRESHOLD:
    # --- Static partitioning (row assignment) ---
    drone_rows = {
        drone_names[0]: [waypoints_grid[0],
            waypoints_grid[3]],
        drone_names[1]: [waypoints_grid[1],
            waypoints_grid[4]],
        drone_names[2]: [waypoints_grid[2]],
    }
    drone_waypoints = {
        drone: [wp for row in rows for wp in row]
        for drone, rows in drone_rows.items()
    }
```

Listing 3.7: Static partioning code

- If drones are far apart, Voronoi partitioning will be used, this means that for each waypoint, the Euclidean distance to each drone's starting position is computed, and the waypoint is assigned to the closest drone. This ensures that the drones cover the area nearest to them, minimizing overlap and travel time;
- The algorithm starts by flattening the 2D grid of waypoints into a single list (`all_waypoints`), making it easier to process each waypoint individually for assignment (we won't be worrying about rows for this partioning scheme);
- The starting positions (latitude, longitude) of all drones are put into a NumPy array (`drone_points`) for Euclidean distance calculations;
- An empty list (`drone_waypoints`) is created for each drone to store its assigned waypoints;
- For each waypoint, the Euclidean distance to each drone's starting position is calculated using `np.linalg.norm`;
- The drone with the minimum distance to the waypoint is selected using `np.argmin`;
- The waypoint is appended to the list of the closest drone;
- In this way each, drone receives a unique set of waypoints that are closest to its starting position, minimizing travel distance and balancing the workload;

- This method adapts automatically to drones positions, making it robust for real-world deployments;

```
else:
    # --- Voronoi partitioning --- (more dynamic)
    all_waypoints = [wp for row in waypoints_grid
        for wp in row]
    drone_points = np.array([drone_homes[d] for d in
         drone_names])
    drone_waypoints = {d: [] for d in drone_names}
    for wp in all_waypoints:
        dists = np.linalg.norm(drone_points - np.
            array(wp), axis=1)
        closest_drone = drone_names[np.argmin(dists)
            ]
        drone_waypoints[closest_drone].append(wp)
```

Listing 3.8: Voronoi partioning code

- To conclude, we made the decision of having two distinct approaches when doing the patrolling not only because it allowed for a more dynamic simulation but also because if we applied the voronoi partioning to the drones when they were too close to each other, each drone would end up having the same path waypoints because the calculations are made based on the drone's current location, therefore making their paths overlap completely which is not ideal. This is why we had to come up with a solution (static partitioning) that mitigated this problem and made the system more robust.

4. **Mission script creation**

- For each drone, a block of Groovy code is generated that:
  - Arms and takes off the drone.
  - Sends it to each of its assigned waypoints.
  - Returns it home after completing its route.

```
# assign all drones at once
assign_line = f"({', '.join(drone_names)}) = assign
    {', '.join([repr(d) for d in drone_names])}"

# build parallel run blocks for each drone
run_blocks = []
for drone, waypoints in drone_waypoints.items():
    waypoints_groovy = ",\n        ".join(
        [f"[lat: {lat:.6f}, lon: {lon:.6f}]" for lat
            , lon in waypoints]
    )
    run_blocks.append(f"""{{
    arm {drone}
    takeoff {drone}, 5.meters
    takeoff_alt_{drone} = {drone}.position.alt
    waypoints_{drone} = [
        {waypoints_groovy}
    ]
    for (wp in waypoints_{drone}) {{
        move {drone}, lat: wp.lat, lon: wp.lon, alt:
            takeoff_alt_{drone}
    }}
    home {drone}
}}""")
```

Listing 3.9: Run blocks for each drone

- All drone blocks are executed in parallel using the `run()` and `wait` instructions in the Groovy mission script.

```
# compose the mission script using run/wait for
    parallel execution
mission_script = textwrap.dedent(f"""\
/*
* multi_drone_multi_waypoint_scout_parallel.groovy
* Sends three drones to different waypoints and
    returns them home in parallel.
*/
{assign_line}
(mission1, mission2, mission3) = run(
    {',\n        '.join(run_blocks)}
)
wait mission1
wait mission2
wait mission3
""")
```

Listing 3.10: Mission script that will compose the code blocks for each drone

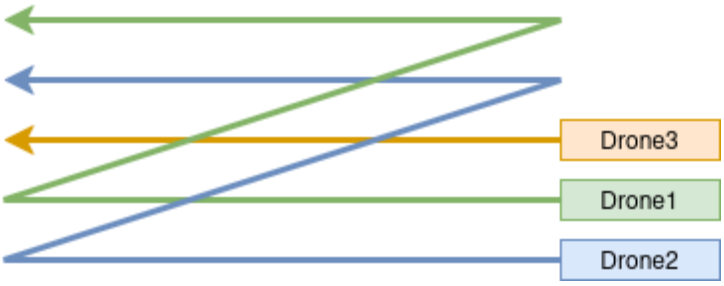

Figure 3.2: Static partitioning drone paths

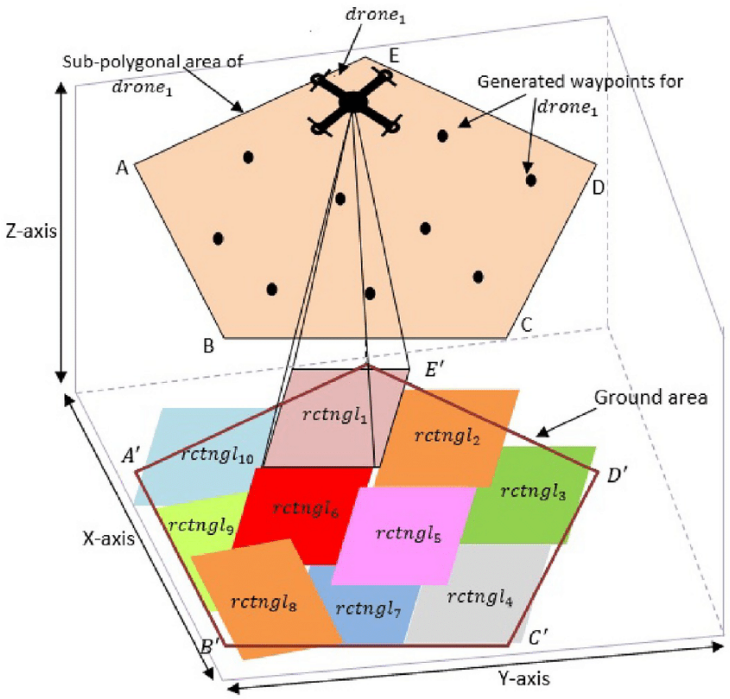

Figure 3.3: Voronoi partitions assigned to a drone

5. **Mission Upload**

- The complete mission script is written to a temporary file and uploaded to the groundstation via an HTTP POST request to the `"http://localhost:8001/mission"` endpoint;
- The groundstation receives and executes the mission, coordinating all drones for efficient area coverage.

Chapter 4

Results

In this project we successfully implemented an autonomous drone network for detecting and responding to forest fires efficiently. With the help of a trained YOLO model for fire detection and implementing collaborative drone strategies, our project showcased how multiple drones can potentially work together to cover large areas in a real life scenario.

For future improvements, we would look into applying our algorithms into real hardware and having the YOLO fire detection model running inside a drone instead of simulating it. For mission 2 we would try to find a more adaptable algorithm than just expanding the perimeter (for example we could make squares from the epicenters of fires which when overlapping would form new polygons for the drones to patrol). For mission 3 we would like to apply computer vision algorithms to obtain terrain mapping images.

Author's Contribution

Each member contributed equally to the development of this project, therefore, we decided to allocate 50% to each.

References & Links

- Fire detection algorithm repository
- Mission 1 video
- Mission 2 video
- Mission 3 video: Voronoi Partitioning
- Mission 3 video: Static Partitioning
- Project repository
- QGroundControl app
- Voronoi partitions image