1 Simulations

In this chapter, we present the implementation details of the experiments, the values of the employed parameters and the results obtained from the simulations. In particular, the parameters relative to the UAVs physical characteristics used in the simulations are from the real Draganflyer IV quadrotor [2], these values are summarized in Table 1:

Table 1: **Drone Physical Parameters**

Symbol	Value	Unit	Description
m	0.4	kg	Total mass of the UAV
J_x	3.8278×10^{-3}	$\text{kg} \cdot \text{m}^2$	Moment of inertia about x-axis
J_y	3.8278×10^{-3}	$\text{kg} \cdot \text{m}^2$	Moment of inertia about y-axis
J_z	7.1345×10^{-3}	$\text{kg} \cdot \text{m}^2$	Moment of inertia about z-axis
ω_{max}	260	rad/s	Maximum velocity of each rotor
l	0.205	m	Distance from center to rotor
c_t	6.354×10^{-4}	–	Coefficient for thrust generation
c_d	0.048	–	Coefficient for torque generation

1.1 Implementation

The implementation of this work is done in *Matlab* and it is structured into multiple sections and files, to increase the modularity and readability of the code while simplifying the debugging process at the same time. The entire code is divided into the following categories:

- A folder containing the **RLS** algorithm in the case where the sources do not influence one another;
- A folder containing the complete **PSO** algorithm integrated with the PID control (the main focus of our work);
- A file with the complete **NSS** analysis of rotational invariance and symmetry properties;
- A file with the **PID** control applied to tracking the initial trajectory the UAVs follow for setup.

The latter contains the same controller and model functions of the `Controller` class of the PSO algorithm. The only difference consists in the desired velocities and positions, which are given by a predefined trajectory and by the PSO algorithm respectively.

In both the last two experiments (trajectory tracking and PSO with control) the controller gains are chosen with values as described in Table 2:

Table 2: **Control Gains for Position and Attitude Control**

Proportional Gain (k_p)		**Derivative Gain** (k_d)		**Control Type**
Symbol	**Value**	**Symbol**	**Value**	
k_{xp}	0.1	k_{xd}	0.5	Position x
k_{yp}	0.1	k_{yd}	0.54	Position y
k_{zp}	1.0	k_{zd}	2.5	Position z
$k_{p\phi}$	2.0	$k_{d\phi}$	2.5	Roll ϕ
$k_{p\theta}$	2.0	$k_{d\theta}$	2.5	Pitch θ
$k_{p\psi}$	2.0	$k_{d\psi}$	4.0	Yaw ψ

The complete PSO with Control algorithm has the following main components:

- Setup and **`main loop`** of the simulation;
- The **`Particle`** class;
- The **`Controller`** class;
- The **`ARTVAs`** class;
- The **`Plotter`** class.

The main structure of the simulation is the following: we first perform a setup step in which we initialize the UAVs, the transmitting ARTVAs and other useful variables and constants. Then, the simulation is divided into two phases:

- **Exploration** phase: the UAVs are asked to position themselves uniformly in the search space by following a predefined trajectory;
- **Exploitation** phase: the UAVs are moved by the modified complete PSO algorithm to localize the multiple avalanche victims (ARTVAs).

If we have successfully estimated the position of all the transmitting ARTVAs we stop the simulation, otherwise if any of the victims cannot be found the simulation stops after a predefined (`n_iterations`) maximum number of iterations.

The implementation begins with the set-up of many variables and constants that will be useful throughout the whole experiment (see Table 3); and the initialization of the classes and theirs crucial parameters (UAV physical quantities, controller gains and particles' data).

The most important hyperparameters include the number of UAVs (`n_drones`), the number of iterations (`n_iterations`), search space boundaries (`bounds`), and maximum velocity

Table 3: **PSO Hyperparameters**

Symbol	Value	Unit	Description
$n_{\text{iterations}}$ (`n_iterations`)	600	–	Number of iterations
ω (`inertia`)	1.05	–	Inertia weight
c_1 (`cognitive_factor`)	2.05	–	Cognitive factor
c_2 (`social_factor`)	1.5	–	Social factor
`NSS_threshold`	290000	–	NSS signal locating threshold
β (`velocity_randomness`)	0.5	–	Velocity randomness
$\Omega_{x,y}$ (`bounds`)	$[-80, 80]$	m	Search space limits
v_{max} (`max_velocity`)	1.5	m/s	Maximum velocity
r_{comm} (`communication_radius`)	10	m	Communication radius
α (`step_size`)	120	m	Move away distance
r_{excl} (`exclusion_zone_radius`)	2.5	m	Exclusion zone radius
δt (`dt`)	0.04	s	Control time step
Δ (`iteration_duration`)	1	s	PSO iteration duration

(`max_velocity` or v_{max}). The sources (`p_sources`) are fixed at predefined positions, and their number (`n_sources`) determines how the UAVs are grouped during initialization, therefore we suppose we have at least a prior knowledge of the maximum number of victims. Instead, the PSO algorithm employs the inertia (`inertia` or ω), cognitive (`cognitive_factor` or c_1), and social (`social_factor` or c_2) factors to balance exploration and exploitation, ensuring UAVs explore the search space efficiently before converging to optimal solutions. Other relevant parameters include the `velocity_randomness` (β), which determines the level of persistency of excitation, the number of groups (`n_groups`) which matches the number of sources and the `communication_radius` (r_{comm}), which sets the maximum range for UAV data transfer. Instead, the `step_size` (α) defines the meters the UAVs need to travel to move away from an exclusion zone they entered. The `exclusion_zone_radius` (r_{excl}) prevents redundant exploration near detected sources and it avoids multiple UAVs converging on the same source. It is important in determining the "resolution" of the algorithm, since no more than one victim can be found in that radius by the UAVs, only by the human rescuers. Instead, the `NSS_threshold` has been carefully finetuned so that the UAVs are precise enough to determine the value of the victims, but also fast enough to create an exclusion zone so that no other UAVs "get stuck" around the same source. Time step (`dt`) represents the τ_s control time, while the `iteration_duration` is the Δ of 1 second in Section **??**, determining the interval between one PSO update and another. Each UAV has an initial position, velocity, and group affiliation, while a data structure is preallocated to store its trajectory throughout the search process. In addition to this, we also initialize the `Plotter` class, whose job is to display in real-time UAV positions, trajectories, and exclusion zones in real-time.

Then the two phases begin: the exploration phase uses a radial pattern to evenly partition the search space, dividing the space uniformly between the UAVs. They move in straight lines toward their goals at maximum allowed speed.

Instead, during the main PSO loop, UAVs evaluate their positions based on the NSS signal received from the `ARTVAs` class. If a UAV detects NSS values higher than a (`NSS_threshold`), it flags the localization of a UAV. Drones that enter exclusion zones move away from them and create a new group, with their velocities, personal best (`p_best` or $\mathbf{pbest}$), and group best (`g_best` or $\mathbf{gbest}$) resettled.

After the real-time simulation stops visualizing the final positions of UAVs and sources, plotting their trajectories, and computing errors between detected and true source positions.

1.1.1 `ARTVAs` class

The sole purpose of the `ARTVAs` class is to emit the signal received by a UAV, the NSS. The signal of a single ARTVA source is "emitted" using the *send_signal_NSS()* function, which takes the position of the ARTVA and the UAV as input and returns the relative gradient matrix, computed using the analitical method with Eq.**??**. Then the *superpositionNSS()* method sums the gradient matrices and computes the eigenvalues of the resultant total magnetic gradient, following the same steps used to obtain Eq.**??**.

The main parameter of this class is `C`, a scaling factor that determines the magnitude of the NSS signals, which is set to a very high value of 10^7, since we do not consider the other physical parameters involved and the signal has very low values with increasing distances.

1.1.2 `Plotter` class

The `Plotter` class is the responsible component for the visualization of the simulation in real-time during: it opens a separate window at the beginning of the simulation and efficiently displays the simulation inside it. It also provides methods for initializing plots, updating UAV positions, displaying exclusion zones, and presenting trajectories. This visualization aids in monitoring the algorithm's progress and understanding of group dynamics. This class naturally includes properties to manage the figure sizes, axes limits, UAV and source markers, color schemes, legends, and bounds of the plot. It also updates the plot title with the current iteration number and simulation time.

The *draw()* method dynamically updates the positions of the UAVs and displays the fixed positions of the sources. Since the draw calls of the `Plotter` can be the real bottleneck in the main loop of the simulation, we carefully optimized the code relative to this task so that

redrawing the scene only updates the parts that changed between subsequent time steps and everything else is kept as is.
Note that the simulation waits a defined time-step at each loop, from which all the computational time is subtracted, so to have faster simulations. However, this does not affect the time used to comment on the results or for other calculations used in the algorithm, which remains the actual time it takes the UAVs to localize the victims. After this optimization, we managed to make the `Plotter` run fast enough that the simulation seems effective in real time (since the draw calls do not run in a separate thread and therefore are blocking calls).

In particular, the `Plotter` displays:

- The exploration phase trajectory lines in radial shape with respect to an imaginary circular boundary;
- The UAVs, each assigned their unique color based on their group affiliation;
- The positions of the real ARTVAs;
- The position of the multiple estimations of the ARTVA, each with the same color as the corresponding group;
- The exclusion zones, represented as circles whose centers are determined by the UAV's position when the signals surpassed the threshold.

Note that when a new group is created during the simulation, a new color is dynamically generated and added to the palette. The UAV colors and the plot legend are updated accordingly.

In the final stages of the simulation, the *plot_best()* method displays the best estimates for source positions achieved by each group. These estimates are marked with group-specific colors, and the legend is updated to include them. Similarly, the *plot_trajectories()* method visualizes the paths traveled by all UAVs, providing insight into their trajectories during the PSO algorithm, since during the real-time simulation the positions are shown only at each PSO time step τ_s.

1.1.3 `Controller` class

The `Controller` class contains the complete quadrotor dynamics and the PID control laws. We define inside it the physical properties of the UAV, such as its mass, the inertia matrix, and the torque and drag coefficients (see Table 1). Furthermore, we define the control proportional and derivative gains both for the x, y, z positions and the the roll (ϕ) and pitch (θ) angles.

The *quadrotor_full_dynamics()* method computes the UAV's translational and rotational

dynamics, which are based on its current state, external forces, and torques. The state derivative is computed using the Equations **??**.

A second method, *control_laws()*, is responsible for calculating the input force and torques required to control the quadrotor. This method first computes the position and velocity errors, which are used to determine the desired thrust like in Eq. **??**. From this desired thrust, the method calculates its x and y components T_x and T_y, which are used to derive the desired roll ϕ_d and pitch θ_d angles with Equations **??**, **??** respectively. The desired yaw torque is computed separately to align the quadrotor with the desired heading, ensuring accurate orientation control. These desired angles and the associated angular velocity errors are then combined with the controller gains to calculate the commanding input torques, following Equations **??**, **??**, **??**.

Finally, the *compute_rotor_velocities()* method determines the rotor speeds needed to achieve the commanding thrust and torques. The speeds are found inverting $\mathbf{A}$ matrix as in Eq.**??**. The computed rotor speeds are checked to ensure non-negative values, as negative speeds are physically invalid.

1.1.4 `Particle` class

The `Particle` class, which is also called `Drone` class, models each UAV in the complete PSO algorithm integrated with the PID control and positioning phase.
Each particle stores its current position and velocity, as well as its personal best position (`p_best`), the corresponding highest NSS value found so far (`nss_best`), their group index (`group_idx`), the `victim_found_flag` to remember the found source and data structures to remember the exclusion zones that have been shared with and by the UAV. The constructor of the class initializes the position at the center of the search space, and the personal best to this position.

This class contains the most important methods: the *update_velocity()* method modifies the particle's velocity using the formula described in **??**. Then, random noise is added to the velocity as in Eq.**??**, while constraints ensure the velocity does not exceed the `max_velocity` following Eq.**??**.

Instead, the *update_state()* function is the one responsible of the implementation of the complete quadrotor dynamics and control laws, by calling the respective methods we have seen in the `Controller` class 1.1.3. This is achieved through the initialization of a `Controller` class in every `Drone` class.

Furthermore, the *evaluate_nss()* method calculates the NSS value at the particle's current position by passing the `ARTVAs` class appropriate function, as described in Algorithm 1. Its

personal best is updated if higher than the previous best. and if the particle discovers a source it sets the flag to create an exclusion zone to true. The exclusion zone is maintained until the end of the simulation. In this last case, the inertia is reduced to decrease the excitation level.

Algorithm 1: `evaluate_nss` (MATLAB function)

Input: $\mathbf{p}_r$: current position of the particle.
Input: `p_sources`: array of source positions.
Input: $superpositionNSS$: function to compute the total NSS at a given position.
Output: `nss_value`: NSS value at the particle's current position.
Output: `p_best`: personal best position.
Output: `nss_best`: personal best NSS value.
Output: `my_exclusion_zone`: center of the created exclusion zone.

Function `evaluate_nss`($\mathbf{p}_r$, *p_sources*)**:** /* */
    Compute the NSS value at the current position:
1: `nss_value` ← *superpositionNSS*($\mathbf{p}_r$, `p_sources`);
2: **if** `nss_value` > `nss_best` **then**
3: `p_best` ← $\mathbf{p}_r$;
4: `nss_best` ← `nss_value`;
5: **end**
6: **else if** `nss_value` > `NSS_threshold` **and** `victim_found_flag` = `false` **then**
7: `victim_found_flag` ← `true`;
8: `my_exclusion_zone` ← $\mathbf{p}_r$;
9: `inertia` ← 0.5;
10: **end**
11: **end**

Communication between particles is allowed only by the *can_I_communicate_with()* method, which determines if two particles are within the predefined communication radius, which is condition **??**.

The *share_exclusion_zones()* method allows the UAV to communicate with its neighbouring UAVs the position of the found "supposed" source.

Finally, the *check_if_in_exclusion_zone()* method verifies if a particle is inside any exclusion zone the UAVs know of and the *move_away_from_exclusion()* function makes the UAV take a big "jump" in the same incoming direction to get as far as possible from the zone, the goal position is computed using Eq.**??**.

1.1.5 `Main Loop`

The first step in the `main loop` involves the initialization of the UAVs (particles) and in particular the sorting of the UAVs in different groups as described in Algorithm 2. The UAVs

are grouped firstly by filling all the available spots, then by adding randomly the UAVs in excess to the groups and then they are sorted for improved performances.

Algorithm 2: `sort_drones_in_groups` (MATLAB function)

Input: **`n_drones`**: number of UAVs available for assignment.
Input: **`n_sources`**: number of sources to which UAVs need to be assigned.
Output: **`group_indices`**: array of indices indicating the assigned group.
Output: **`n_groups`**: the obtained number of groups.

Function `sort_drones_in_groups(`*`n_drones, n_sources`*`):` `/*  */`

1: **if** `n_drones` $>$ `n_sources` **then**
2: **Assign** at least one UAV to each group, following mathematical order;
3: **Assign randmly** remaining UAVs to existing groups;
4: **end**
5: **else if** `n_drones` = `n_sources` **then**
6: **Assign** each UAV to a unique group;
7: **Sort** the group indices to ensure orderly grouping;
8: **end**
9: **else**
10: **Assign orderly** UAVs up to `n_drones`;
11: **end**
12: **Set** the number of groups: `n_groups` $\leftarrow$ `max(group_indices)`;
13: **end**

Note that the function `sort_drones_in_groups` ensures the creation of sufficient groups for the localization of multiple sources only when the number of UAVs is equal to or greater than the number of sources.

Exploration Phase

At the start of the simulation, all UAVs are located at the origin of the search space. During the first Exploration Phase, the PSO algorithm is not yet employed. During this phase, the UAVs move according to predefined trajectories at maximum speed $v_{\max}$. The trajectories are determined based on the number of UAVs and follow a radial pattern. The UAVs follow these trajectories until they have covered a distance `travel_distance` equivalent to half the boundary of the search space. The goal each UAV needs to reach is computed using the following Algorithm 3:

Algorithm 3: `exploration_goal` (MATLAB function)

Input: `n_drones`: number of UAVs available for exploration.

Input: `bounds`: maximum limit of the search space.

Output: `goals`: array containing x,y positions of the exploration goals.

Function `exploration_goal(`*`n_drones, bounds`*`):` `/*` `*/`

Initialize an empty array `goals`.
Set `angle_step` $\leftarrow \frac{360}{\texttt{n\_drones}}$.
Set `travel_distance` $\leftarrow \frac{\texttt{bounds}}{2}$.
for drone $j = 1$ **to** m **do**
  Compute angle for the current UAV: `drone_angle` $\leftarrow j \cdot$ `angle_step`;
  Calculate `slope` $\leftarrow \tan\big($*deg2rad*$(\texttt{drone\_angle})\big)$;
  `/* 1st quadrant */`
  if `drone_angle` > 315 **or** `drone_angle` ≤ 45 **then**
    Set `goal` $\leftarrow [\texttt{travel\_distance}, \texttt{slope} \cdot \texttt{travel\_distance}]$
  `/* 2nd quadrant */`
  else if `drone_angle` > 45 **and** `drone_angle` ≤ 135 **then**
    Set `goal` $\leftarrow [\frac{\texttt{travel\_distance}}{\texttt{slope}}, \texttt{travel\_distance}]$
  end
  `/* 3rd quadrant */`
  else if `drone_angle` > 135 **and** `drone_angle` ≤ 225 **then**
    Set `goal` $\leftarrow [-\texttt{travel\_distance}, \texttt{slope} \cdot -\texttt{travel\_distance}]$
  end
  `/* 4th quadrant */`
  else if `drone_angle` > 225 **and** `drone_angle` ≤ 315 **then**
    Set `goal` $\leftarrow [-\frac{\texttt{travel\_distance}}{\texttt{slope}}, -\texttt{travel\_distance}]$
  end
  `goals` $\leftarrow$ `goals` + `[goal]`;
end
end

where `angle_step` is the angular increment to uniformly divide the space, `drone_angle`

represents the angle assigned to each UAV, while `m` is the slope of the trajectory calculated from the UAV's angle.

Exploitation Phase
This is the most important part of the simulation since it involves the localization of the victims through the deployment of our modified PSO algorithm with the PID control, described in Section **??**.

At each τ_s iteration step until `n_iterations`, each j-th UAV checks the neighboring UAVs with the *can_I_communicate_with()* function and with a positive response they share their exclusion zones with each other (if they have found a source). The communication radio (r_{comm}) is set to 10 m.

Then, it calls *check_if_in_exclusion_zone()* to verify if its position is inside the radius of any of the saved exclusion zones. In case of a positive response, the UAV is commanded to execute the *move_away_from_exclusion()* function. Then, its velocity is reset to zero, and its personal best position, `p_best`, is assigned to a random point far from the new location. Additionally, the $\mathbf{gbest}$ is reset to $-\infty$, and the UAV's `nss_best` is also reset to $-\infty$, ensuring the UAV restarts exploring with a fresh start. For simplicity reasons, this movement is considered "instantaneous", meaning that in the simulation, the UAV's journey to the goal is not displayed. Instead, in the next iteration, it will already be at the designated location. However, the number of iterations and the simulation time account for the actual time required for the UAV to reach the goal at maximum speed. In case of a negative response, the *evaluate_nss()* is called and the group best ($\mathbf{gbest}$) is updated if the received NSS signal is greater than the previous best.

Subsequently, the *update_velocity()* method computes the desired velocity command for the control loop and the desired position is obtained using the standard PSO formula in Eq **??**. At each time step τ_k until the end of the Δ, the *update_state()* function is called and the boundries are enforced as in **??**. Finally, we can present the complete modified PSO algorithm in Algorithm 4.
Note that the inertia of a UAV that has found a source is reduced ($\omega = 0.5$), the same way the maximum allowed velocity in the PSO is reduced to $v_{\max} = 0.5$ m/s, increasing convergence speed. The final output is the set of best estimates `g_best_values`, composed of the best-estimated position of a source by each group. Finally, we want to note that during the movement dynamics of the UAVs the exlusion zone mechanism and the NSS signal reception are both implemented, but not repeated in Alg.4 for brevity.

Algorithm 4: Particle Swarm Optimization for Multi-Source Localization

Initialize positions of UAVs to the center of the search space.
Initialize `g_best_values`.
Assign each UAV to a group using the function in Algorithm 2.

Exploration Phase:
Compute exploration goals using the function in Algorithm 3;
for drone $j = 1$ **to** m **do**
    Move to goal with `max_velocity`;
end

Exploitation Phase:
for $\tau_s = 1$ to `n_iterations` **do**
    for drone $j = 1$ **to** m **do**
        for each other drone $l \neq j$ **do**
            if *can_I_communicate_with()* **or** `victim_found_flag` = true **then**
                Share exclusion zones between j and l;
            end
        end
        if *check_if_in_exclusion_zone()* = true **then**
            if drone j is not alone in its group **then**
                Reassign drone j to a new group;
                Initialize new `g_best` and `nss_best`;
            end
            Move away from exclusion zones;
            Reset personal best `p_best`;
            Set drone j `inertia` to 0.5;
        end
        else
            Evaluate NSS at $\mathbf{p}_j(\tau_s)$;
            if `nss_value` $>$ `nss_best` **then**
                Update `p_best`;
            end
            if `nss_value` $>$ `g_best` **then**
                Update `g_best_values`;
            end
        end
        Compute desired velocity: *update_velocity()*;
        Apply persistence of excitation as in Eq.(**??**);
        Limit velocity to `max_velocity` as in Eq.(**??**);
        Desired position: as in Eq.(**??**);
        Enforce boundaries on desired position as in Eq.(**??**);
        Set `iteration_duration` to 1;
        for `t` $= 0 :$ `dt` $:$ `iteration_duration` **do**
            Control dynamics: *update_state()*;
        end
    end
end

1.2 Results

1.2.1 PID for Trajectory Tracking

The results of PID control using the full quadrotor dynamics model (**??**), applied to trajectory tracking during the first positioning (Exploration) phase of the algorithm, are presented in the following figures: Figure 1 illustrates the comparison between the actual and desired trajectory in 3D space, showcasing the effectiveness of the control strategy in aligning the UAV's path with the desired trajectory.

The desired trajectory, which has been chosen for this phase, is designed to provide smooth and easily controlled movement. The x and y components of the trajectory follow a linear motion with maximum velocity $v_{\text{max}} = 2.1$m/s. Since the UAVs need to travel on straight lines that radially distribute over the horizontal xy-plane, we define the `drone_angle` in Algorithm 3 as γ the orientation w.r.t. the x-axis of the trajectory. The altitude is modeled to rise smoothly using an exponential function, achieving a desired altitude z_d, with a smoothness factor k controlling the rate of ascent. The mathematical expressions for the desired trajectory and its derivatives are:

$$
\begin{aligned}
\mathbf{x_d}(t) &= \begin{bmatrix} x_d(t) \\ y_d(t) \\ z_d(t) \end{bmatrix} = \begin{bmatrix} v_{\text{max}}\, t \cos(\gamma) \\ v_{\text{max}}\, t \sin(\gamma) \\ z_d \left(1 - e^{-kt}\right) \end{bmatrix} \\
\dot{\mathbf{x}}_\mathbf{d}(t) &= \begin{bmatrix} \dot{x}_d(t) \\ \dot{y}_d(t) \\ \dot{z}_d(t) \end{bmatrix} = \begin{bmatrix} v_{\text{max}} \cos(\gamma) \\ v_{\text{max}} \sin(\gamma) \\ z_d k e^{-kt} \end{bmatrix} \\
\ddot{\mathbf{x}}_\mathbf{d}(t) &= \begin{bmatrix} \ddot{x}_d(t) \\ \ddot{y}_d(t) \\ \ddot{z}_d(t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -z_d k^2 e^{-kt} \end{bmatrix}
\end{aligned}
\tag{1}
$$

Figures 2, 3, and 4 provide a detailed breakdown of the trajectory tracking along the x, y, and z axes, respectively. These plots highlight the individual tracking performance in each dimension and confirm the controller's precision in following the desired positions.

Figure 5 shows the simulation over time, providing an overview of the control system's behavior during this phase. Additionally, the rotor velocities calculated using (**??**) are plotted in Figure 6.

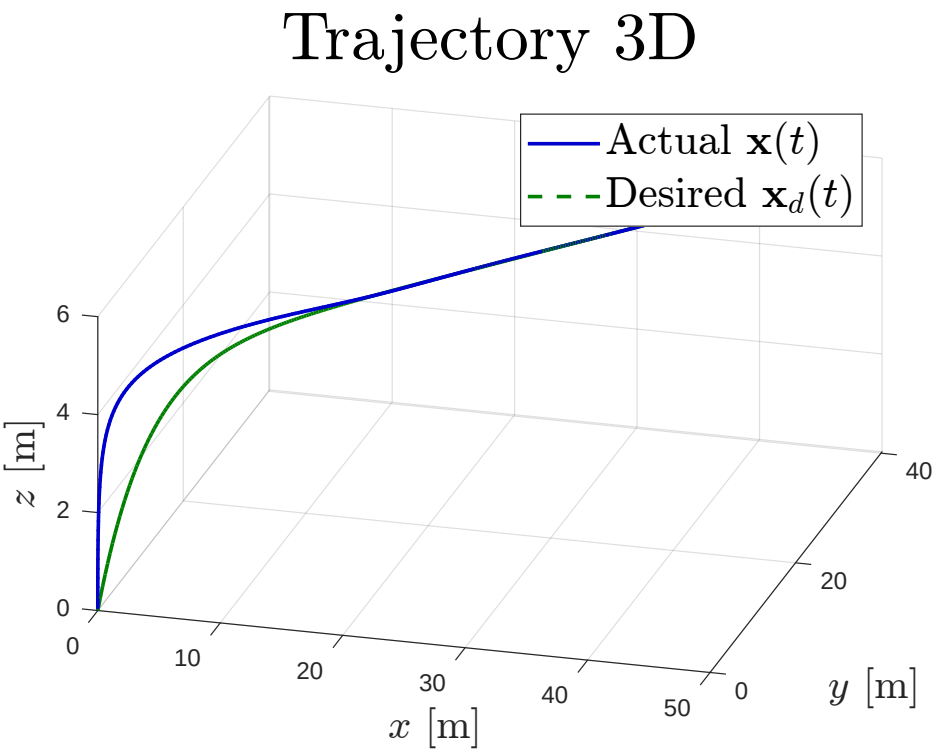

Figure 1: Comparison of the actual and desired trajectory in 3D space during the exploration phase.

The results demonstrate that after an initial overshooting the position controller can follow the desired trajectory. In particular, altitude control (hovering) is also quite important in our application and it is successful in our results. Furthermore, the rotor velocities exhibit regular and achievable values, meaning the actuation inputs can be followed. These results were obtained under a nominal scenario, where wind disturbances or other environmental factors are not considered, since the primary focus of this work is not on control robustness.

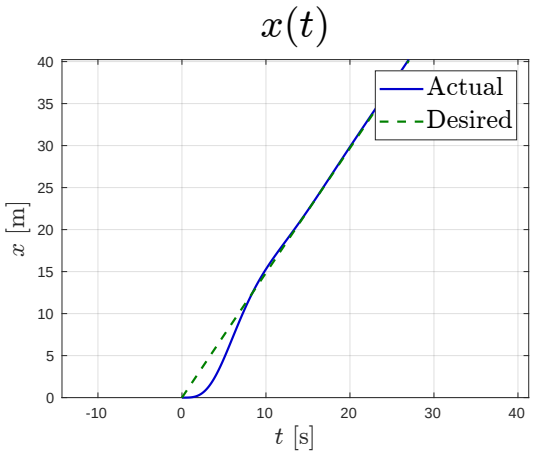

Figure 2: Actual and desired $x(t)$.

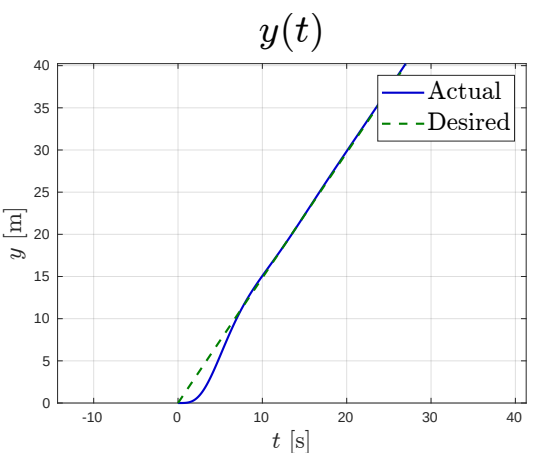

Figure 3: Actual and desired $y(t)$.

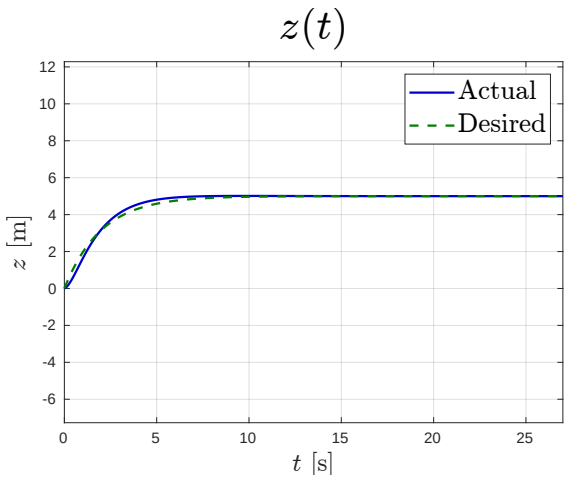

Figure 4: Actual and desired $z(t)$.

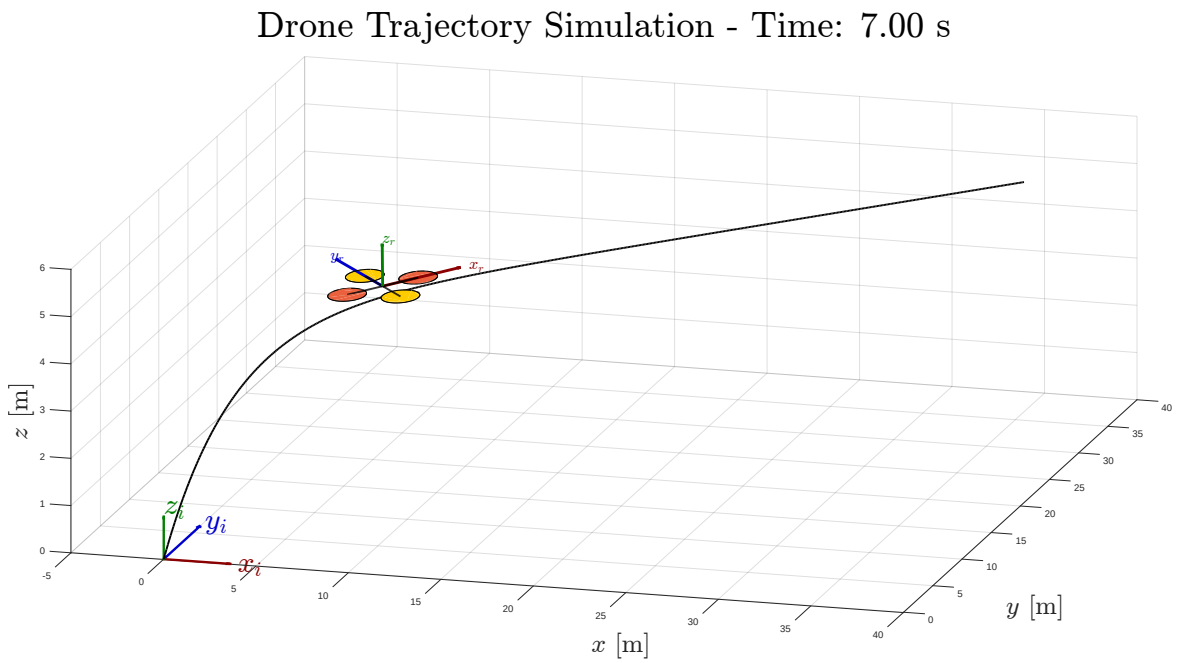

Figure 5: Time instant during the simulation illustrating the system's dynamic behavior.

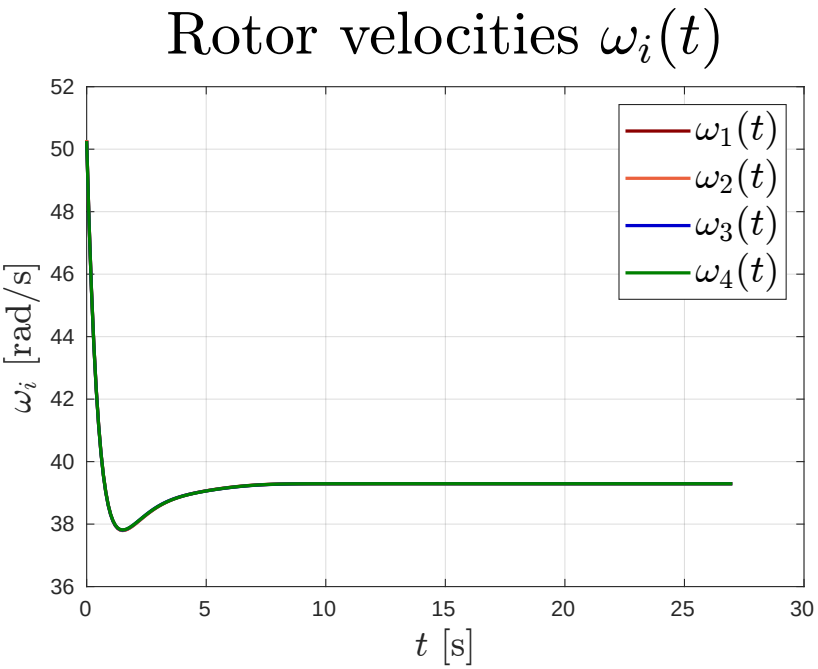

Figure 6: Rotor velocities over time as calculated using (**??**), the actuation required for trajectory tracking.

1.2.2 PSO with Control

In the context of search and rescue (S&R) missions, the number of available UAVs is typically limited (ranging from 1 to 10 [1]), and the search area usually spans only a few thousand square meters. To have faster simulations and more constrained values, we limit the search space to the hundreds of meters. However, the algorithm is scalable to bigger distances while maintaining the same performances by incrementing the UAVs' speed, or enhancing the capabilities of the ARTVA receiver, which is usually limited to our chosen limits.

Each UAV in the PSO control integrated framework is initialized with the hyperparameters described in the previous sections. The UAVs start their search with an initial position at the center of the search space, defined as $[0, 0]$. Their initial velocities are randomized using the velocity randomness factor capped at a maximum velocity of 1.5 m/s to prevent overshooting targets. The search space is constrained within bounds of $[-80, 80]$ m for both the x and y directions.

Each UAV's best-known position ($\mathbf{pbest}$) is initialized to its starting position, while its best NSS value is set to $-\infty$, ensuring that any detected signal improves its current state. The UAVs operate with an inertia weight of 1.05, balancing exploration and exploitation during the optimization process.

The UAVs are grouped based on a variable group index, and each group best-known position ($\mathbf{gbest}$) is also initialized to $-\infty$. Regarding the orientation and UAVs dynamics, the roll, pitch, and yaw angles are initialized to $[0, 0, 0]$. The initial velocity during the second phase of the algorithm is the same of the arriving velocity at the goal of the first phase. These parameters collectively define the initial state and behavior of the UAVs in the PSO

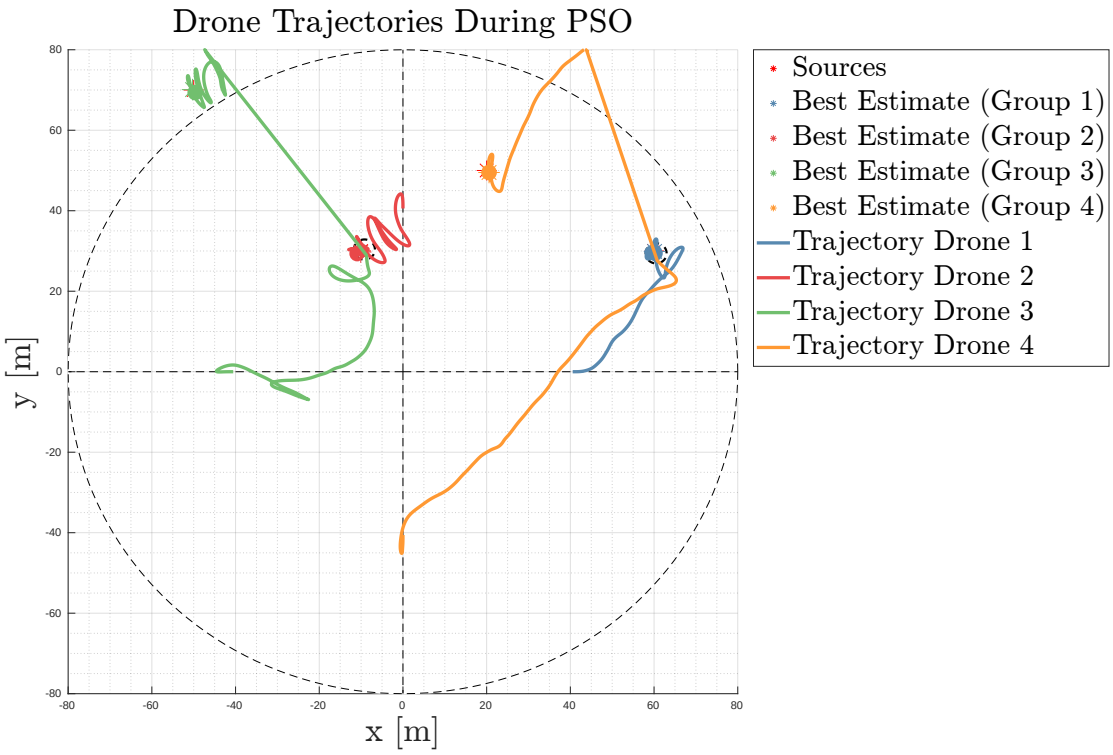

Figure 7: Results and Trajectories in Case 1.

framework. In the following sections, we report the results relative to 3 different scenarios regarding the number of UAVs employed and the positions of the victims in space, under nominal conditions both for the ARTVA signal and the PID controller.

Case 1

In the first case, the PSO algorithm needs to locate four sources positioned at $[20, 50]$ m, $[-50, 70]$ m, $[60, 30]$ m, and $[-10, 30]$ m, randomly distributed across the search space.

The time it took for the swarm to find the victims was 151 seconds (~ 2.52 minutes), during which the UAVs successfully converged on the sources, achieving precise estimates with an average error of 0.38 m. The estimate for the first source was just 0.756 m away, while the second, third, and fourth sources were located with errors of 0.295 m, 0.165 m, and 0.283 m, respectively.

In Figure 7 we can see the sources' locations and the UAVs' trajectories during the PSO optimization process, from which is evident that the exclusion zone mechanism worked in favor of UAVs 3 and 4 in successfully locating a source.

We also show the control inputs necessary to achieve the desired results during the Exploitation phase in Figure 8, as the ones during the Exploration phase have already been shown in the

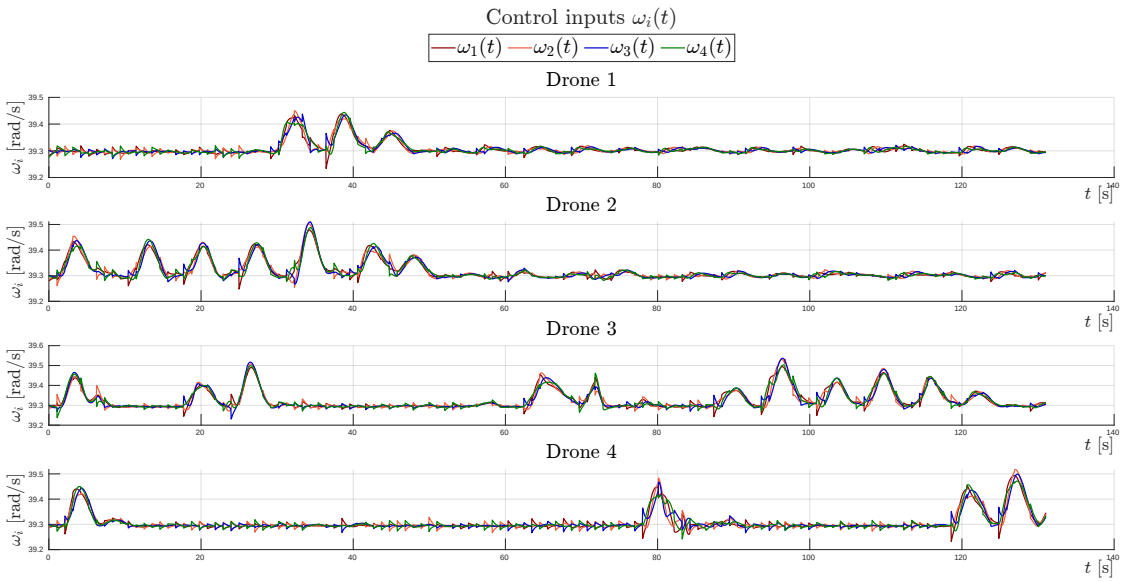

Figure 8: Rotor velocities for all UAVs during the Exploitation Phase in Case 1.

previous section.

Furthermore, the desired control inputs generated by the PSO algorithm are compared with the actual positions (Figures 11 and 12) and velocities (Figures 9 and 10) for each drone. Divided into the x and y components, the figures demonstrate that the controller performs adequately in the position control task (which is the primary focus of our interest), while showing slightly less accuracy in the velocity control, which has a more erratic behavior and therefore is more difficult to track.

Case 2

In the second case, the PSO algorithm is again tasked with locating four sources positioned at $[-70, -70]$ m, $[70, 70]$ m, $[-70, 70]$ m, and $[70, -70]$ m, distributed very far apart across the search space.

The time it took for the swarm to locate the sources was 519 seconds (~ 8.65 minutes). During this time, the UAVs successfully converged on the sources, achieving an average error of 0.39 m. The estimates for all the sources were: the first source was 1.025 m away, the second was 0.180 m away, the third was 0.147 m away, and the fourth was 0.208 m.

In Figure 13, the sources' locations and the UAVs' trajectories during the PSO optimization process are depicted. This case is one of the most complicated since the sources are near the borders of the search space and therefore the signal they send are less strong.
To evaluate the impact of increasing the number of UAVs, a second simulation was conducted by adding a UAV to the swarm. Naturally, we see improved performance in the total time needed by the UAVs to find all victims, which was just 233 seconds (~ 3.88 minutes). The

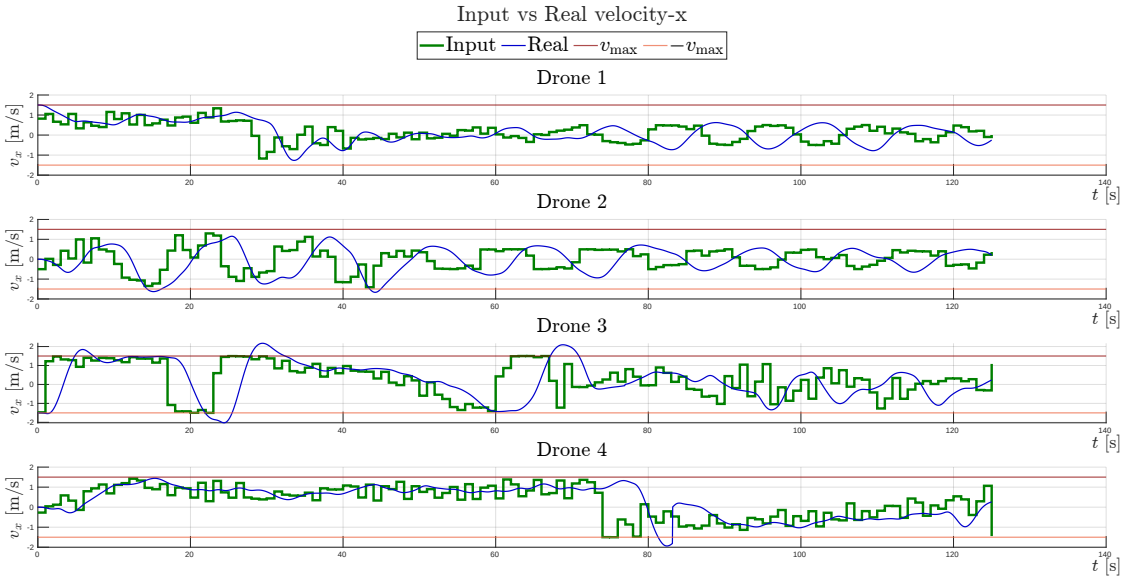

Figure 9: Comparison of the x-component of velocity between the input control and real UAV velocities during the PSO phase in Case 1.

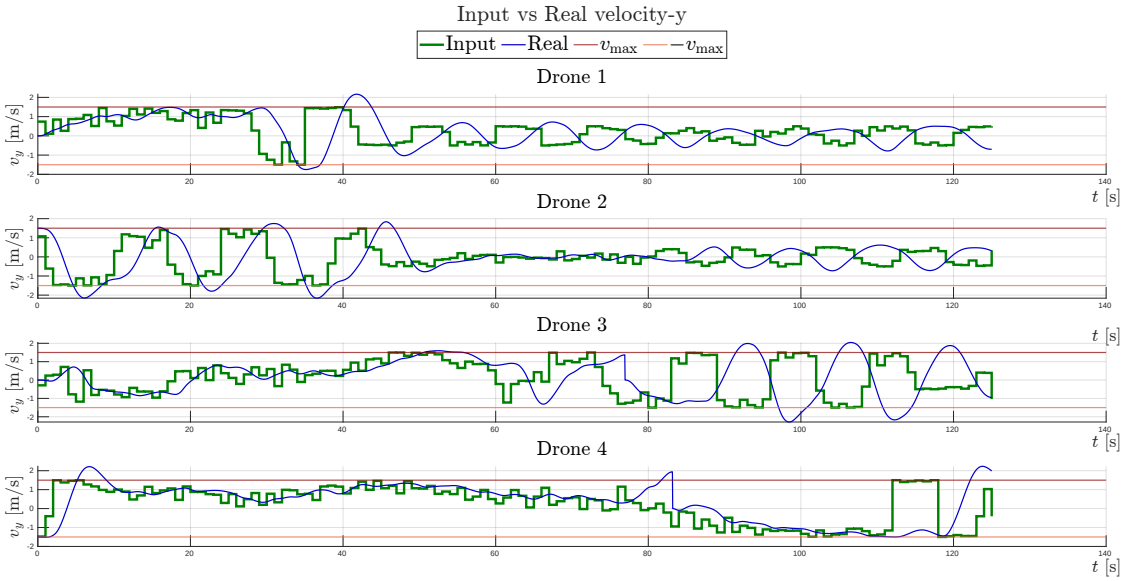

Figure 10: Comparison of the y-component of velocity between the input control and real UAV velocities during the PSO phase in Case 1.

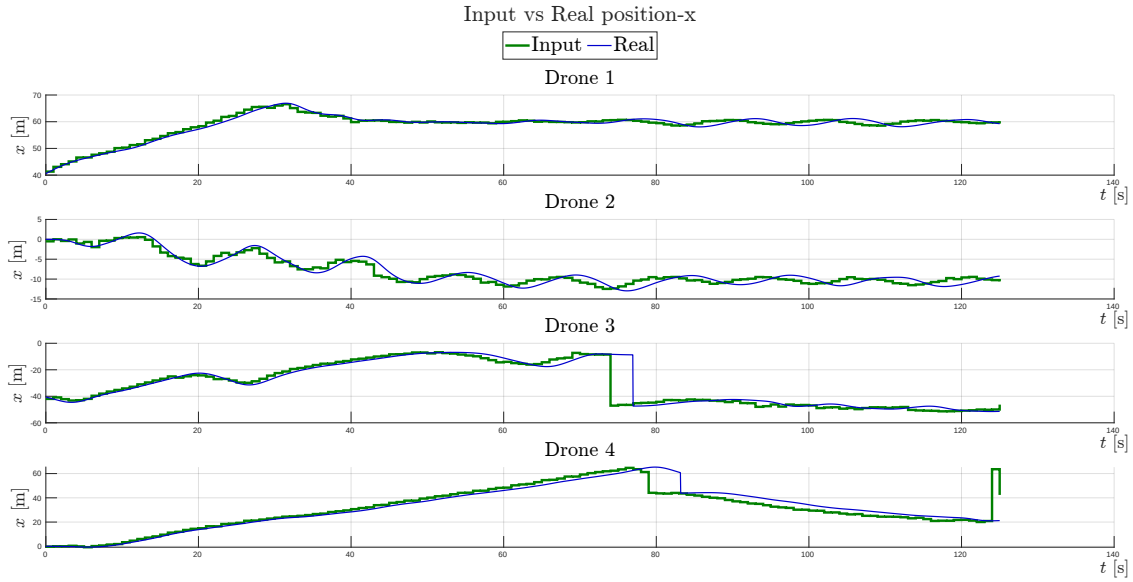

Figure 11: Comparison of the x-component of position between the input control and real UAV positions during the PSO phase in Case 1.

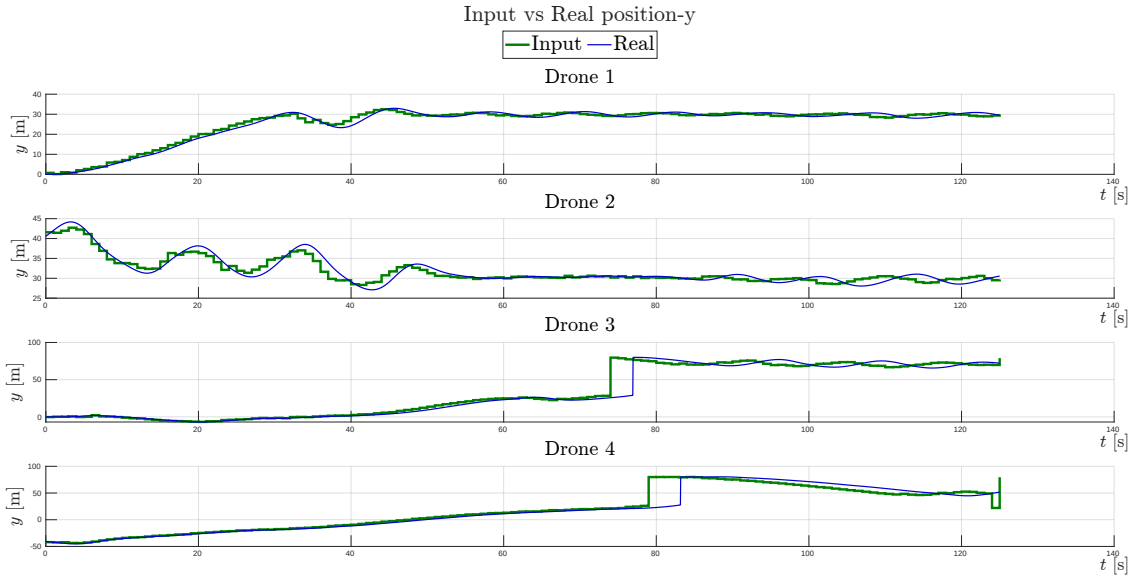

Figure 12: Comparison of the y-component of position between the input control and real UAV positions during the PSO phase in Case 1.

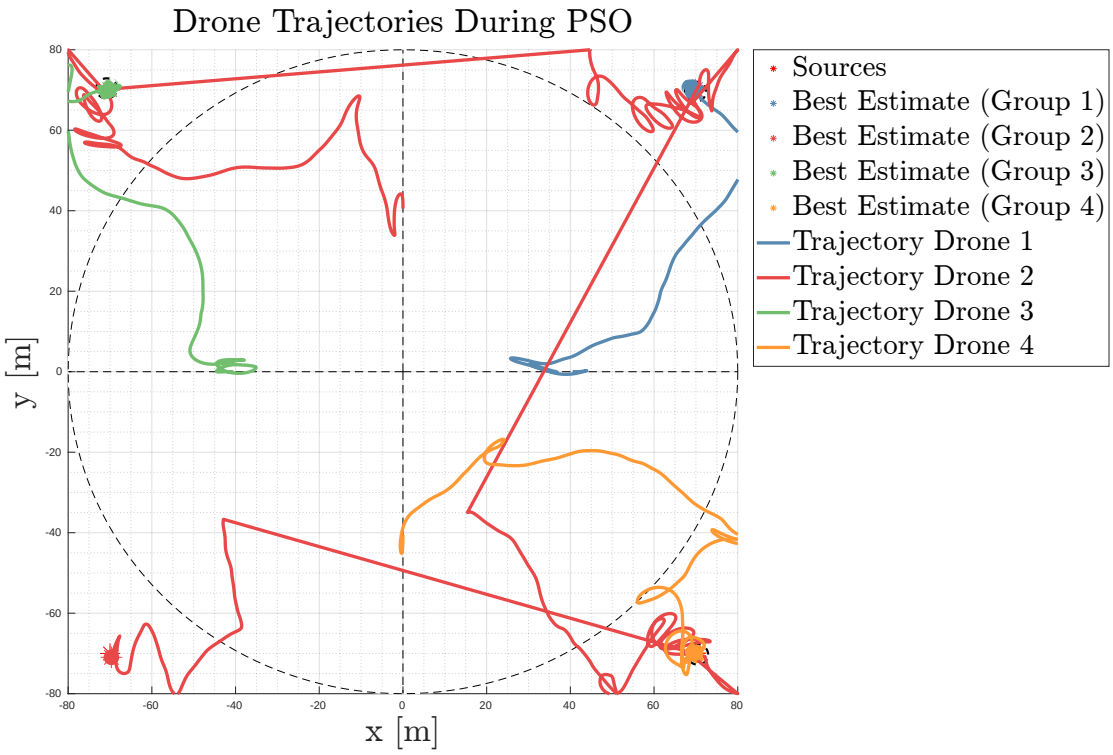

Figure 13: Results and Trajectories in Case 2.

average error across all sources slightly decreased to 0.37 m. The time improvement was due to some UAVs having a better trajectory during the exploration phase, which brought them nearer the sources. Figure 14 illustrates the results of this improved configuration.

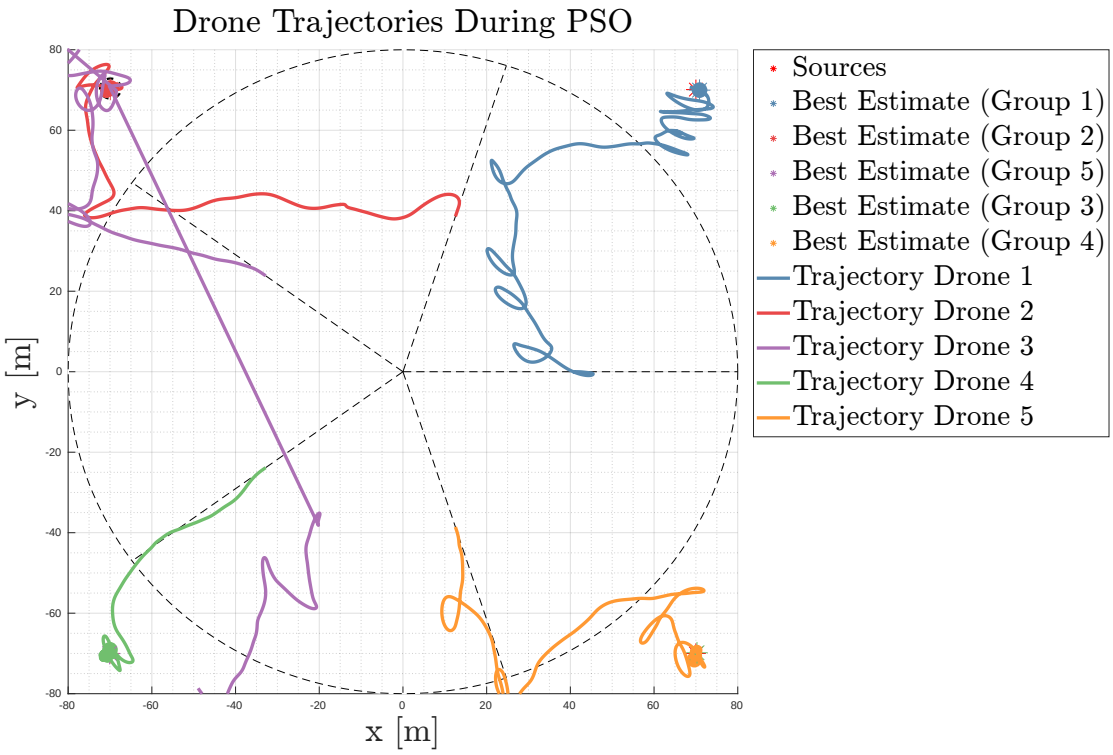

Figure 14: Results and Trajectories in Case 2 with an additional UAV.

Case 3

In the third case, the PSO algorithm faces a challenging scenario where the four sources are positioned very close to each other, only 8 m apart, at $[20, 50]$ m, $[28, 50]$ m, $[20, 58]$ m, and $[28, 58]$ m. This situation is particularly difficult for the algorithm due to the fact that we have an exclusion zone radius of 2.5 m and whenever one UAV gets inside the exclusion zone of another it has to go 120 m away. Therefore, when three UAVs have found their sources, the fourth has only a few possible directions to reach the fourth source.

Initially, with four UAVs, the algorithm fails to reliably locate all sources within the 600 seconds (10 min) allocated for the simulation, as seen in Figure 15. While increasing the simulation time could potentially improve performance, this approach is unsuitable for our time-sensitive application, where avalanche victims have a real chance of surviving only within 15 minutes after the time of the accident.

To address this issue, an additional UAV is introduced, increasing the swarm size to five UAVs. Now, the algorithm successfully locates all sources in 544 seconds (~ 9 minutes). The estimates for the sources are: the first source was 0.227 m away, the second was 0.925 m away, the third was 0.160 m away, and the fourth was 0.152 m away. The total error for all valid estimates was 0.37 m, demonstrating the effectiveness of increasing the swarm size in challenging scenarios. In Figure 16, the addition of a fifth UAV and the subsequent success

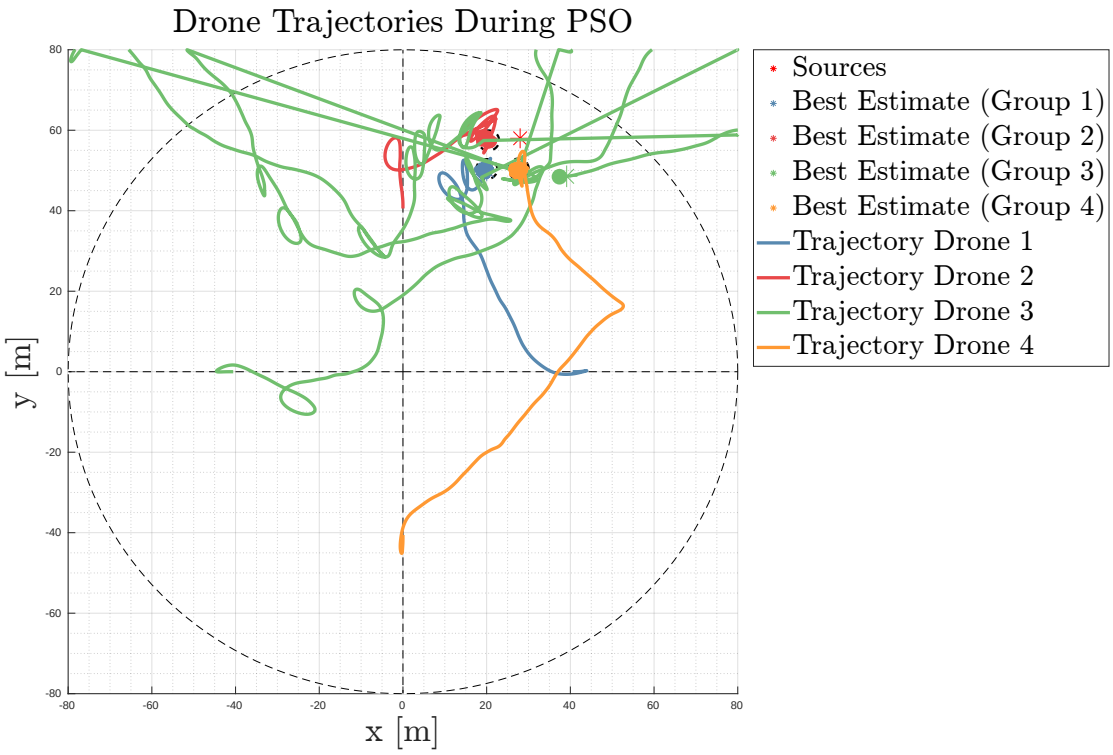

Figure 15: Results and Trajectories in Case 3 with fail.

in the localization can be seen clearly.

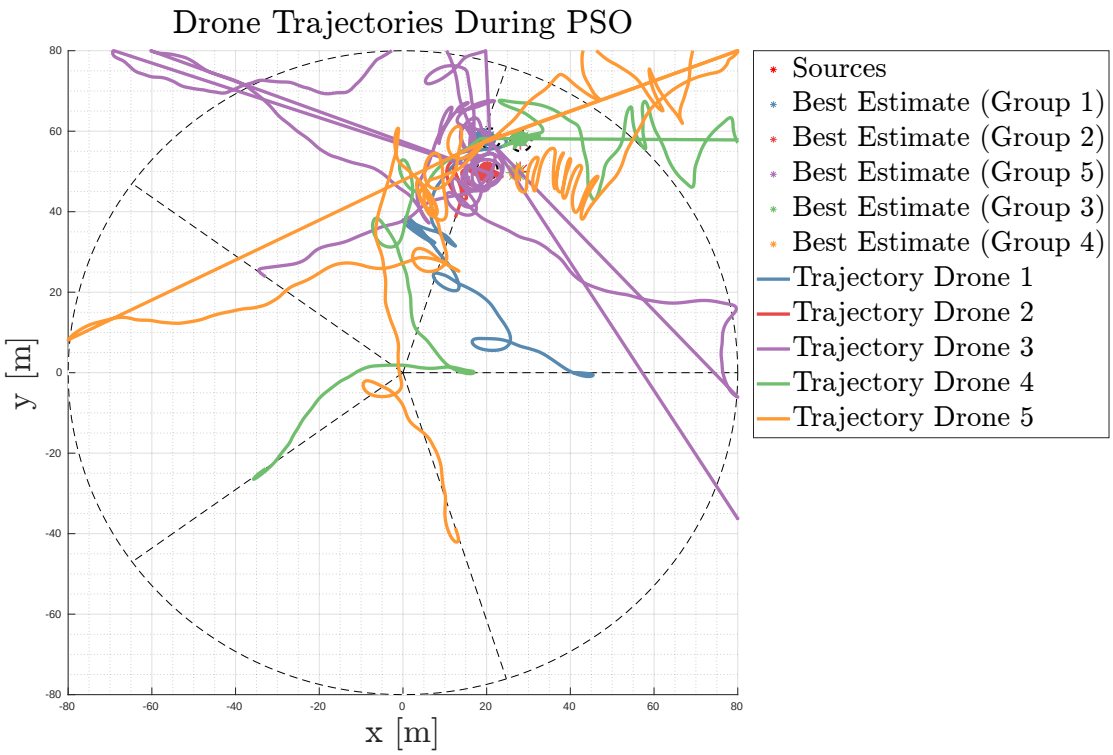

Figure 16: Results and UAVs Trajectories in Case 3 with an additional UAV.