DRONES Lab - Part 2: Flocking

Mathilde THEUNISSEN
mathilde.theunissen@ls2n.fr

December 16, 2024

1 Objectives

The aim of this lab is to implement and analyze Olfati-Saber's flocking algorithm for distributed multi-agent systems in a simplified 2D scenario.

In this lab, you will implement the **cohesion**, **separation** and **alignment** rules to create a flocking behavior. These rules ensure the drones stay connected, move cohesively toward a common goal, and avoid collisions.

Challenge the limits of the algorithm by varying the number of drones, choosing challenging trajectories and adding obstacles in unexpected positions.

This lab is strongly based on the work of Olfati-Saber, outlined in the paper: *R. Olfati-Saber, "Flocking for multi-agent dynamic systems: algorithms and theory," in IEEE Transactions on Automatic Control, vol. 51, no. 3, pp. 401-420, March 2006.* **Make sure to read this article before attending the lab session.**

Deliverables You will have to submit:

- a brief report in which you analyze the behavior of your flock.
- your code files.

2 Description of the lab

2.1 Run the lab

To begin the lab, you need to download the project files from GitHub. Use the following commands in your terminal to clone the repository and navigate to the project directory:

```
git clone https://github.com/mtheuniss/TP_flocking_student.git
cd TP_flocking_student
```

This will download the git repository. The lab is implemented entirely in Python. To run the program, execute the main.py file either using your preferred code editor or directly from the command line with:

```
python main.py
```

2.2 Lab files

Here are the files for this project:

```
TP_flocking_student
├── main.py
├── flock.py
├── drone.py
└── utils.py
```

You will have mainly to complete some functions in `flock.py` and in `drone.py`. The file `utils.py` provides useful functions that support the lab. This file also contains the flocking parameters. Before implementing any function, make sure it is not already included in this file.

3 Implementation

3.1 Create the topology of the flock

The flocking algorithm is a **distributed** control approach. Each drone only interacts with its immediate **neighbors**, determined by a maximum communication/sensing range r. This relationship among drones can be modeled as a weighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where:

- The vertices $\mathcal{V}$ represent the drones.
- The edges $\mathcal{E}$, representing proximity, connect neighboring drones.

To do: Implement the function `get_adjacency_matrix` in `flock.py`, which computes $A = [a_{i,j}]$, the spatial adjacency matrix of this graph. Each element $a_{i,j}$ is computed in the function `compute_aij` in the file `utils.py` which you should complete.
Hint: Refer to Equations 1 and 11 of Olfati-Saber's article.

To do: Implement the function `get_laplacian_matrix` in `flock.py` which computes the graph Laplacian.
Hint: The Laplacian is defined in Equation 17.

3.2 Indicators

The group of drones performs flocking if:

- the group is a **quasi-flock**. This means that the distance between neighboring drones is almost the desired distance d. The **deviation energy** $E = \frac{1}{1+|\mathcal{E}(q)|}\sum_{i=1}^{n}\sum_{j\in\mathcal{N}_i}(||q_j - q_i|| - d)^2$ measures the deformation of the group from the ideal configuration, where the distance between drones equals d. $|\mathcal{E}(q)|$ is the number of edges in the graph and q_i refers to the position of the drone i.
- the group remains **cohesive**. All drones are then connected and contained within a circle of finite radius. This property can be verified using the **relative connectivity index** $C = \frac{1}{n-1}\text{rank}(L)$. Since $\text{rank}(L)$ can reach a maximum value of $n-1$, the relative connectivity ranges from 0 to 1. A value of 1 indicates that the graph is connected, while a value of 0 signifies that no drones are connected. The cohesion can also be observed by computing the **cohesion radius** $R = \max_i ||q_i - q_c||$ which measures the maximum distance of a drone from the group's centroid q_c.
- the group reaches a common velocity. The **velocity mismatch** $K = \frac{1}{2}\sum_{i=1}^{n}||p_i - p_c||^2$ indicator computes the difference between the individual velocities p_i and the group velocities $p_c = \frac{1}{n}\sum_j p_j$ (average velocity).

To do: To analyze the future performances of your flock, implement in the file `flock.py`:

- the **relative connectivity** in the function `compute_connectivity()`
- the **cohesion radius** in the function `compute_cohesion_radius()`
- the **normalized deviation energy** $\widetilde{E} = \frac{E}{d^2}$ in the function `compute_deviation_energy()`
- the **normalized velocity mismatch** $\widetilde{K} = \frac{K}{n}$ in the function `compute_velocity_mismatch()`

Hint: The different indicators are given paragraph IX.A, page 18 of Olfati-Saber's article.

3.3 Drone dynamics

Drone dynamics are simplified as double integrators. In `drone.py`, complete the function `update_state`.

Hint: Look at the equations of motion in Equation 2.

3.4 Free-flocking

3.4.1 Reynolds Rules

Let's now compute the Reynolds Rules. Each robot i receives as control input an acceleration proportional to an attractive/repulsive force u_i^α such that:

$$u_i^\alpha = c_1^\alpha \sum_{j \in \mathcal{N}_i} \phi_\alpha(||q_i - q_j||_\sigma)\mathbf{n}_{ij} + c_2^\alpha \sum_{j \in \mathcal{N}_i} a_{ij} \cdot (p_j - p_i)$$

The second term represents the velocity matching term while the first term corresponds to flock separation and alignment.

To do: In `drone.py`, implement u_i^α in the function `u_alpha()`. Then complete the function `update_cmd` in order to update the drone control command at each time step.
Hint: This exercise corresponds to the Algorithm 1 presented in Equation 23.

To do: Modify the initial size of the area where the drones are initialized by changing the following line in `main.py` (line 18):

```
INIT_AREA_FLOCK = np.array([20,20])
```

Experiment with different box sizes to observe the impact on flocking behavior.

Question: Test your control law. Is the flock behavior what you expected? Are the Reynolds Rules sufficient to create a flocking behavior? Justify your answer using the 4 indicators.

3.4.2 Navigation feedback

To finish the flocking protocol, we add a navigation feedback term u_i^γ such that:

$$u_i^\gamma = -c_1^\gamma(q_i - q_r) - c_2^\gamma(p_i - p_r)$$

with q_r and p_r respectively a reference position and velocity that is shared with all drones.

To do: In `drone.py`, implement u_i^γ in the function `u_gamma()`. Then complete the function `update_cmd` with this second attractive force.
Hint: This exercise corresponds to the Algorithm 2 presented in Equation 24.

To do: Test your flock with different numbers of robots (parameter in `main.py`) and tune the flocking parameters (in `utils.py`).

Question: Test your control law. Is the flocking working? Why?

3.5 Flocking with obstacle avoidance

Let's make our flock reactive to avoid **convex obstacles**. The key idea is to represent nearby obstacles as virtual kinematic drones positioned at the point on the boundary of the obstacle that is the closest to the nearby drone. A drone and an obstacle are considered as neighbors if their distance is smaller than a threshold r_{obs}.
Computation of the neighboring drones and the closest point on the obstacle to a drone is provided. You can check this in the function `get_pos_vel_virtual_agent()` in `obstacle.py` and in the function `update_flock` in `flock.py`.

An additional repulsive u_i^β force is added to each drone to prevent it from colliding with obstacles.

$$u_i^\beta = c_1^\beta \sum_{j \in \mathcal{N}_i^{obs}} \phi_\beta(||q_i - \widehat{q}_j||_\sigma)\mathbf{n}_{ij} + c_2^\beta \sum_{j \in \mathcal{N}_i^{obs}} a_{ij} \cdot (p_i - \widehat{p}_j)$$

To do: In `drone.py`, implement u_i^β in the function `u_beta()`. Complete then the function `update_cmd` with this repulsive force.
Hint: The final algorithm is presented in Equation 59.

To do: In `main.py`, uncomment the **line 10** to add some obstacles:

```
OBSTACLES = [
    Circle(np.array([15,2]), 2.),
    Circle(np.array([30,7]), 6.),
    Wall(np.array([45,15]), 6,20),
    Wall(np.array([45,-15]), 6,20),
]
```

You may change the position of the obstacles and the navigation trajectory by changing in line 16 of `main.py`:

```
INIT_POSE_FLOCK = np.array([-10,0])
FINAL_POSE_FLOCK = np.array([70,0])
```

Question: Analyze the behavior of the flock using the 4 indicators you have computed.