ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Versatile Simulator for a Swarm of Quadcopters

LIS - Semester Project

Fall Semester 2018-19

Author:
Victor Delafontaine

Supervisor:
Enrica Soria

January 9, 2019

Contents

Versatile Simulator for a Swarm of Quadcopters

Summary

This Matlab simulator was created with versatility in mind. Its goal was to be able to simulate multiple situations depending on the user requirements: for example a single one or a swarm of 100 drones. This project extended a previously existing simulator to simulate multiple drone instances. Based on this, it became possible to test swarm dynamics with algorithms such as Olfati-Saber or Vicsek.
An example of that is shown in figure 1 where we can see three drones initialized with random positions flocking together after 46 seconds of simulation.

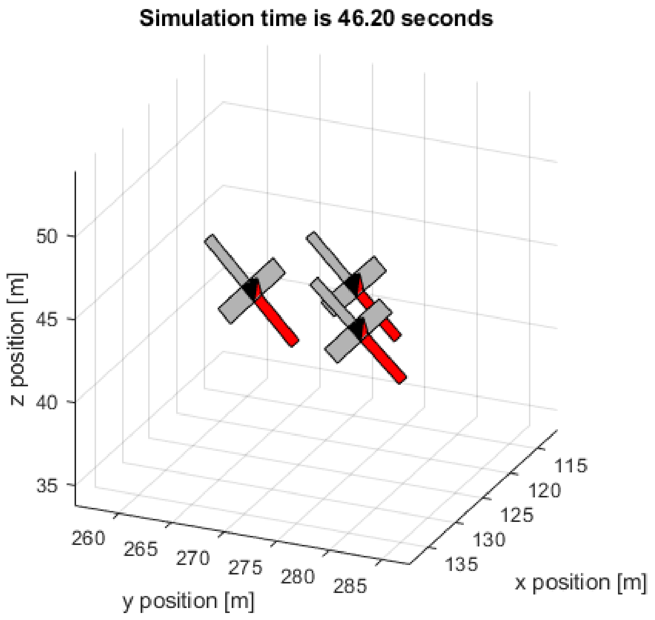


Figure 1: Three drones forming a small flock

In addition to that, this project also brought many improvements to the simulator, such as a graphical user interface for ease of use or two separate simulation modes depending on the simulator's user requirements (realism or speed).

1 Introduction

I chose this project as a semester project during my third semester of master in microengineering.

The goals of the project are multiple. Overall, the final result should be a versatile simulator supporting multiple drones.
The first and main one is to generalize the simulator to multiple quadcopters. This will make it possible to simulate a small fleet of drones flying at the same time. For this, multiple instances of the simulation will need to be run simultaneously. Using object-oriented programming will be essential for this step. The second one is to enhance the simulator as a whole to add ease of use and realism. This will be done by adding different functionalities such as direct parameter change during the simulation using a graphical user interface (GUI) or a simple wind model. The last goal is adding decentralized swarming algorithms to the simulator. The two algorithms used here are Vicsek and Olfati-Saber.

I first did a state of the art study with three different papers relative to the above goals. This gave me a base knowledge on which to start working.

The report is structured as follows. First I will describe the studied articles. Then will come a description of the work done to convert the previous `Simulink` simulator into one using object-oriented programming. The fourth and principal section is the documentation of the new simulator. It describes the work done on this project as well as explains how to use it for future projects.

The resulting simulator can be found on Github at address `https://github.com/lis-epfl/uav_simulator/tree/feature/quadcopter`[4]. All the code was developed and tested in Matlab version 2018a.

2 State of the art

Before beginning this project, a study of the state of the art was done in order to assess the current situation. Three main documents were studied:

1. *Small unmanned aircraft*, by Beard and McLain [1];
2. *Flocking for multi-agents dynamic systems*, by Olfati-Saber [2];
3. *Modeling and control of a quadcopter in a wind field*, by Massé et al. [3].

These three articles are on three subjects that will be studied during this project. In order they are: a general introduction to drone and path planning, the theory behind an example of swarming algorithm and finally an advanced control method for drone autopilot in a disturbed environment.

2.1 Beard and McLain: *Small unmanned aircraft*

The first document studied is published in 2012 by two professors at University of Princeton, Randal W. Beard and Timothy W. McLain. It treats on the subject of drones, more specifically fixed-wing. This book was already used to create the simulator, which take a lot from concepts described there. Understanding this book and its structure is important to understand the current simulations.

The first chapter is on the drone's basic coordinate frames. In this section, the authors describe the Euler angle representation, NED (north-east-down) referential as well as airspeed model. These are important to master as the drone orientation will vary in the different frames, so knowing how to represent the drone is essential.

The second chapter describes the drone kinematics and dynamics. The drone has a total of six degrees of freedom. Here, the drone state is represented with four main triplets. This same state will be used throughout the project. It is represented as follows:

$$x = [p_n\ p_e\ p_d\ u\ v\ w\ \phi\ \theta\ \psi\ p\ q\ r]$$

In order, the four triplets are position in NED frame, velocity in body frame, attitude (roll, pitch and yaw) and attitude rate.
The kinematic equations used to obtain new position based on the previous one as well as the forces and moments applied on the drone are also introduced. The external forces are the gravity and drag, and the drone creates control moments from the control surfaces (elevator, rudder and ailerons) and propeller thrust.

The next chapter is on the subject of autopilot. It describe the structure of the autopilot with lateral-directional and longitudinal autopilots. Note that this is only described for a fixed-wing UAV. The autopilot structure will be very different for a quadcopter, as their control and structure are. The autopilot described there uses different PID control loops. For example a loop is created for the airspeed hold using throttle.

The authors then describe a sensor and state estimation model. The sensor used are GPS, accelerometers, gyros and pressure sensors. A Kalman filtering is used to have a precise state estimation. While this was implemented in the previous simulator, it was not incorporated into the span of this project.

The following chapters are on the guidance model. This is the main interest of the project. Once again it is only presented in the case of a fixed-wing and will vary for a quadcopter. The guidance model is staged onto three layers. First is a straight-line and orbit following. Once the drone is able to follow a given path, the path manager is added to create the path based on given waypoints. The final layer is the creation of said waypoints by the path manager. These will be explained in more details later in this project in section 4.3.

Overall this book was a great resource as it enabled me to understand the simulator structure as the latter was based on the book.

2.2 Olfati-Saber: *Flocking for multi-agents dynamic systems*

This 20 pages article was published in *IEEE Transactions on automatic control* in 2006 by Reza Olfati-Saber.
It describes the "framework for design and analysis of distributed flocking algorithms" [2].

The author begins by defining flocking algorithms by the rules enabling the flock to follow the Reynolds rules:

- flock centering: stay close to flockmates
- collision avoidance: while keeping a safety distance
- velocity matching: and flying at the same general speed.

Following a general state of the art on different flocking algorithms, he explains the concepts of proximity nets. These are a set of nodes and connections between agents. Different examples are shown in figure 2 below.

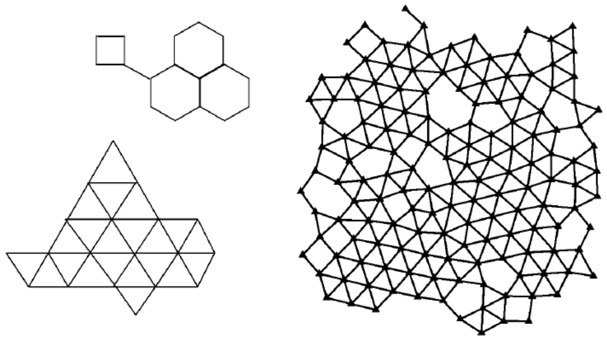


Figure 2: Proximity nets

One goal of a flocking algorithm is to update this net based on the agents position and movement. A swarm can "break" in the case of a net linked at some parts with few links.

Olfati-Saber then proceeds to build the framework for testing the stability of a flock in free flight or disturbed by obstacles.

This article is essential to understand the basics of flocking and how to apply them to a drone swarm.

2.3 Massé et al.: *Modeling and control of a quadcopter in a wind field*

This article describes a robust control technique. It is supposed to be more precise and efficient than a PID controller for an UAV flying in a strong wind field. The article was written in 2018 and published at the *International Conference on Unmanned Aircraft Systems* in Dallas, Texas.

The technique used there is H_∞ and the article shows the guidelines for the synthesis of a controller. It uses the `Robust Control` toolbox of Matlab. The paper describes the state space representation of the dynamic system as follows:

$$\dot{x} = Ax + Bu \quad and \quad y = Cx + Du \tag{1}$$

with x is the drone's current state: $x = [u\ v\ w\ p\ q\ r\ x\ y\ z\ \phi\ \theta\ \psi]^T$ in order linear velocities, rotational velocities, position and orientation, u is the system input represented by the four motors: $u = [\omega_1\ \omega_2\ \omega_3\ \omega_4]^T$ and y is the output, which we chose as the NED velocities and ψ yaw angle (all four are independent): $y = [v_n\ v_e\ v_d\ \psi]^T$.
They then use the toolbox's *loopsyn* function with different response requirements to obtain a transfer functions to go from previous state to new state.

This method can be implemented in the simulator in order to improve the drone handling in high wind scenarios. As the wind model had to be added to the project, we decided that implementing the control model wasn't a priority. However, the article also described a wind model which proved useful for our implementation.

3 Conversion of previous simulator

The previous simulator was only able to model one single drone. This needed to change as we will need to be able to simulate swarms later on.

Changing the simulator done in `Simulink`, to accept running parallel instances was estimated to be difficult. It was chosen that the simulator would be changed using object oriented programming (OOP) instead of `Simulink`.

The idea is to create a *Drone* object (=class) which will have properties and methods. Its properties correspond to the different drone variables, such as coordinates (position, speed, attitude...), motor states, etc. Inside the class' methods will be stored the functions directly relative to the drone in itself, for example the function computing the drone dynamics.
In addition to the *Drone* class, we also created a *Swarm* class. This second class only contains a vector of *Drone*s as well as the number of drones in the swarm. In addition to that, it will receive later into the project the properties and methods for swarm navigation.
Instead of `Simulink` model, the new simulator now take the form of a Matlab script creating a *Swarm* object. It then periodically update the states of the *Drone*s contained inside the *Swarm*.

The main difficulties we addressed when converting the simulator from `Simulink` to OOP are as follows:

- `Simulink` has an optimized tool used for the drone dynamics (S-functions[1]), which we will need to replicate using the ordinary differential equation solver functions (such as *ode45(...)*);
- the code previously used a lot of persistent variables, which worked for one drone but won't for the swarm, these variable will need to be added as properties of the *Drone* class;
- the functions needed to be converted to accept a *Drone* (or *Swarm*) as input. They also needed to be moved inside their relative class, which will also clean the code.

[1]`https://www.mathworks.com/help/simulink/sfg/s-function-concepts.html`

3.1 Simulator structure

The existing simulator structure follows a waterfall described in *Beard and McLain*[1]. It is shown in figure 3 below. This structure is valid for a single drone. For multiple, the guidance model (path planner, manager and follower) will need to be replaced by a swarm version. The velocity commands will come from the swarm dynamic based on the drone position.

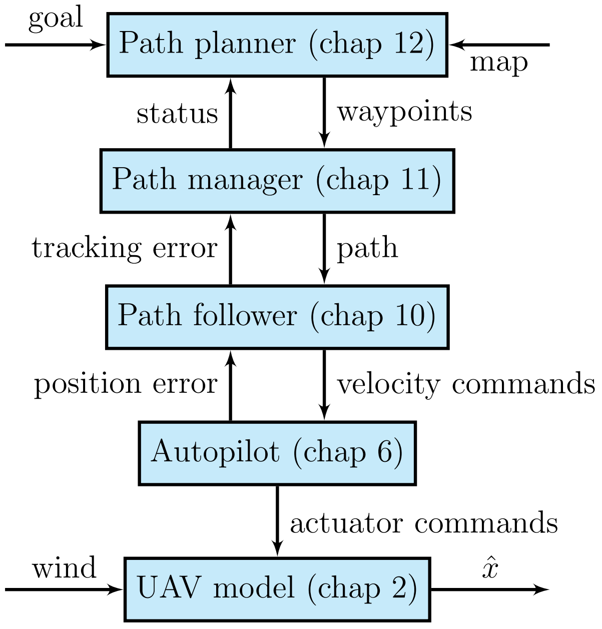


Figure 3: Simulator waterfall structure

The book was also used as a base on which to build the simulator. Some of the functions' structure presented in the `Simulink` simulator were taken directly from the book. In figure 3 these are shown with the relative chapter next to their names.

The waterfall structure enabled the possibility to build the simulator from the ground up, adding functions as the project progressed. Before staring this project, the simulator had all steps implemented for a fixed-wing and only up to the autopilot (no guidance model) for a quadcopter. The first step was to convert the simulator up to the autopilot to OOP. When this was done, we gradually changed the path planning for a fixed-wing into OOP, then finally added the guidance model for a quadcopter.

3.2 Conversion to object oriented programming

The first step to address is the creation of the *Drone* and *swarm* classes. We first focused on the *Drone*, then when it was functional we added the *swarm* function. At this stage of the project, the simulator was tested with a single drone inside the swarm. With this reasoning, we could already create the class, but focus on the swarming functionalities only later on.

All the functions relative to the drone or swarm also had to be moved inside their respective classes. For example, the function presented in the `Simulink` simulator *forces_moments.m* was moved inside the *Drone* class as the method *forcesMoments*. Indeed, this method uses the state of the drone and its computed air data (containing movement and wind velocities) to compute the forces and moments applied to the drone. All input and output values are stored inside the *Drone* class.

Moving the functions inside the class made the code cleaner as all functions and variables relative to the drone or swarm are now stored directly in their respective class.

One of the important challenge in the conversion was to replicate the S-function used to update the drone state. It receives as input the current state as well as the forces and moments applied to the drone. This gives us an ordinary differential equation system which is solved by the solver function.

As we couldn't use the S-function block outside of `Simulink`, we used the ordinary differential equation solver functions (e.g. *ode23*) of vanilla Matlab to solve our equations. The resulting method is shown below in listing 1. The function *kinematicsOde_f* sets the drone's equation system and can be found in annex A.

```
input         = [obj.forces; obj.moments];
tspan         = [time time+obj.P.Ts];
[~,x_ode]     = ode23(@(t, x) kinematicsOde_f(time, x, obj.
    getState(), input, obj.P), tspan, obj.getState());
```

```

x_new = x_ode(end,:);
obj.pos_ned  = x_new(1:3)';
obj.vel_xyz  = x_new(4:6)';
obj.attitude = x_new(7:9)';
obj.rates    = x_new(10:12)';
```

Listing 1: *computeKinematics* method in class *Drone*

3.3 Results

In the end we achieved a working simulation up to the controller stage. Further development concerning the guidance model will be described later on in section 4.

Our OOP simulation was initially slower than the `Simulink` simulation by a factor three. A solution we decided to implement is to separate the simulation into two modes. The first one is time-efficient and plots only what is required: the drone is shown as a dot instead of a complex object with faces and lattices. It also compute the states based on Euler forward method. The second is more precise and plots the whole drone, but takes more time to simulate.
These modes will be described in more details in section 4.6.

4 Documentation

This section will explain the functioning of the simulator and how to use it. The main objects of interest are as follows:

1. the *Drone* and *Swarm* classes: what they contain and how to use them;
2. the autopilot for both a fixed-wing or a quadcopter: input, outputs and function;
3. the guidance model used in the simulation, obtained from *Beard and McLain*[1];
4. the different parameter files and how to modify them in the code;

5. the *main* functions: usage and utility;
6. the simulation modes: lightweight and fast or realistic but slower;
7. the graphical user interface (GUI): how to use it to run the simulation and change parameters;
8. the implemented wind model.

4.1 Swarm and Drone class

These two classes are born from the will to move from a `Simulink` simulator to object-oriented programming (OOP). They contain all properties (variables) as well as all the methods (functions) concerning their relative use.

These are the main properties of the drone class:

- Properties concerning the drone configuration and parameters:
  - *uav_config*: 0 for fixed-wing and 1 for quadcopter
  - *autopilot_version*: only for a quadcopter (fixed at -1 for a fixed-wing), 1, 2 and 3 for respectively attitude, speed and acceleration controller: this will be discussed in section 4.2
  - *P*: set of general parameters P defined by the parameter file relative to the current simulation (see section 4.4)
  - *B*: set of battery parameters B defined by *param_battery.m*: unused in the current state of the simulator
  - *map*: set of map parameters defined by *param_city.m*
- State of the drone:
  - *pos_ned*: (vector 3x1) contains the position coordinates in the inertial frame (North, East and Down)
  - *vel_xyz*: (vector 3x1) contains the velocity in body frame (u, v and w)
  - *attitude*: (vector 3x1) contains the attitude of the drone in Euler angles (ϕ, θ and ψ)

  - *rates*: (vector 3x1) contains angular rates $\dot{\phi}$, $\dot{\theta}$ and $\dot{\psi}$

- Variables for guidance model:
  - *path*: (vector 33x1) contains the current path returned by the path manager: $path = [flag\ Va_d\ r\ q\ c\ \rho\ \lambda\ state\ flag\_need\_new\_wpts]$. r and q are the inertial position and direction of the path, used for straight lines. c is the center of a circle of radius ρ in direction λ (+1 for CW, -1 for CCW) for an orbit follow.
  - *nb_waypoints*: the number of waypoints computed by the path planner. The path manager can demand new waypoints by setting the $flag\_need\_new\_wpts$ to one
  - *waypoints*: (vector 5*Tx1, with T contained inside structure P as P.size_waypoint_array) contains the waypoints, each defined by 5 variables: $[p_n\ p_e\ p_d\ \chi\ Va_d]$, stored as a line vector (need to reshape before using). The "empty" waypoints with an index above *nb_waypoints* defined above are set to $-[9999\ 9999\ 9999\ 9999\ 9999]$

- Miscellaneous variables:
  - *command*: (vector 4x1) contains the command inputs. It depends on the autopilot version (see section 4.2 for more information)
  - *airdata*: (vector 6x1) contains the UAV air speed V_a, angle of attack α, side-slip angle β and wind speed in NED frame w_n, w_e, w_d
  - *forces*: (vector 3x1) contains the forces acting on the drone
  - *moments*: (vector 3x1) contains the moments acting on the drone
  - *x_hat*: (vector 22x1) the estimated state: $\hat{x} = [p_n\ p_e\ h\ v_x\ v_y\ v_z\ V_a\ \alpha\ \beta\ \phi\ \theta\ \psi\ \hat{\chi}\ p\ q\ r\ \hat{V_g}\ \hat{w_n}\ \hat{w_e}\ \hat{b_x}\ \hat{b_y}\ \hat{b_z}]$
  - *delta*: (vector 4x1) contains the normalized angular velocities commanded to the 4 motors in radian per second
  - *full_command*: (vector 19x1) contains the full command state vector, used in function *plot_uav_state_variable.m*: $x_{command} = [p_{n,c}\ p_{e,c}\ h_c\ v_{n,c}\ v_{e,c}\ v_{d,c}\ a_{n,c}\ a_{e,c}\ a_{d,c}\ V_{a,c}\ \ \alpha_c\ \beta_c\ \phi_c\ \theta_c\ \psi_c\ \chi_c\ p_c\ q_c\ r_c]$. In most case, most of the variables are empty. For example for an attitude autopilot, only $h_c, \phi_c, \theta_c, \psi_c$ will be different to 0

- Variable for plot and figure handles

- Parameters for autopilot PIDs (e.g. *P_roll_torque* or *P_pe_roll*)
- Battery state (capacity, voltage and current)

For the swarm the list is much shorter:

- *drones*: a vector of class *Drone* containing all the drones of the swarm
- *nb_drones*: the number of drones contained inside the previous vector
- *equivalentDrone*: the equivalent drone for the swarm, namely a virtual drone with position at the barycenter of the swarm. It is used in the guidance model

You can find the different methods and their usage directly in the code.

The creation of a *Drone* or *Swarm* object can be done with the lines shown in respectively listing 2 and 3.

```
% Drone initialization
drone = Drone(uav_config, autopilot_version, P, B, map);
```

Listing 2: *Drone* object creation

```
% Swarm initialization
swarm = Swarm();
for i=1: nb_drones
    swarm.addDrone(Drone(uav_config, autopilot_version, ...
                                P, B, map));
end
```

Listing 3: *Swarm* object creation

In order to put the drones of a swarm in random positions inside the map, you can run the lines shown below in listing 4. This sets all drones in a cube of size *map.width* (default as 200). The sixth line can be used to fix the altitude as a negative value between 0 and 120 for visibility in the plots.

```
1 % Set swarm position
2 seed = floor(rand()*10000);    % random for random positions
3 %seed = 5;                     % fixed seed to avoid randomness
4 rng(seed);
5 pos0 = repmat(map.width, 1, nb_drones) .* rand(3,nb_drones);
```

```
%pos0(3,:) = -pos0(3,:)*0.6;
swarm.setPos(pos0);
```

Listing 4: *Swarm* position initialization

4.2 Autopilot versions

The input of the autopilot function is a command (for example "move at 2m/s in North coordinate") and its output is a actuator command (for example "set quadcopter motor command to [0.2 0.2 1 1]").

The autopilot is distinct for fixed-wing and quadcopter. The functions are respectively *autopilotWing.m* and *autopilotQuad.m*.

For the fixed-wing, the input commands are $command = [V_{a,c}\ h_c\ \chi_c\ \phi_{ff}]$. For a quadcopter, four different autopilot modes are available. The input commands will vary depending on the mode. These are as follow:

1. autopilot in attitude, $command = [h_c\ \phi_c\ \theta_c\ \psi_c]$
2. autopilot in velocity, $command = [\psi_c\ v_{n,c}\ v_{e,c}\ v_{d,c}]$
3. autopilot in acceleration, $command = [\psi_c\ a_{n,c}\ a_{e,c}\ a_{d,c}]$
4. autopilot in position, $command = [\psi_c\ p_{n,c}\ p_{e,c}\ p_{d,c}]$

Each of these functions will call "hold" sub-functions that uses Matlab's *pidloop* function to obtain the actuator commands. For a quadcopter the output is the rotational speed of each of the four motors (normalized at 1). For a fixed-wing, it is the actuation levels of elevator, ailerons, rudder and throttle.

By default, when commanding a quadcopter the velocity autopilot is used. This can be changed only when running the controller main (*main_controller_6.m*) by changing the value of *autopilot_version*. The other *main* files receive commands from the guidance model or the swarming algorithms and the autopilot type can't be changed. Attitude, velocity, acceleration and position autopilot corresponds to values of respectively 1 up to 4.

4.3 Guidance model

The simulator's guidance model for a single drone is composed of three functions. They are the path planner, manager and follower. In the case of a quadcopter, most of the work to convert the previous `Simulink` simulator was done on this stage. Indeed, the controller was already implemented, but not the guidance for a quadcopter. The guidance for a fixed-wing was implemented but needed to be converted nevertheless.

The three functions were already created for a fixed-wing. However some differences required to do changes to adapt the functions to a quadcopter. The main difference is that a fixed-wing needs to do circular turns and cannot turn directly in yaw without also moving forward. For this reason the path had two options: straight lines or circular turns. In the case of a quadcopter, these two modes are not needed as the drone can rotate freely in yaw. Hence the quadcopter is always given straight lines path, with rotations in between. Another modification is the cruising speed of the drone depending on drone type as a fixed-wing is typically faster than any given quadcopter. The two speeds were fixed at 35m/s and 5m/s.

The three functions described above are staged onto three levels. These levels are also shown in figure 4 below.

1. The **path follower** is the lowest level of the three. It has a fixed path as input and its output is a command for the drone.

2. The **path manager** uses the list of waypoints to extract a path that the drone needs to follow to go to the next waypoint. The waypoints comes from a simplified path planner and are fixed between calls. The created path can then be given to the path follower.

3. The **path planner** is at the highest level. Its input is the map and a goal from which it computes the list of intermediary waypoints in between. These waypoints will be the path manager's input.

We created different *main* functions to test these functions. They were created in order of level, beginning by the path follower and finishing with the path planner. These can be found later on in section 4.5 and can be run separately to test the different stages.

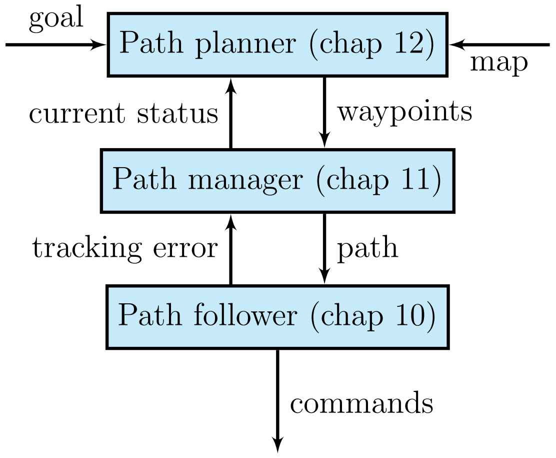


Figure 4: Guidance model structure

4.4 Parameter files

The different parameters used for the simulation are stored inside structures created by the call of the required *param* function.

The main simulation parameters functions have names such as *param_4_forces.m*. In this case, it corresponds to the fourth chapter in *Beard and McLain* [1] on the subject of forces. These files go up to 12 for the path planner parameters and creates the structure P. Each file calls the previous one so only the call of the last required file is needed in the main script. When looking for a parameter to change, you need to understand at which level of the simulation it is first used. For example the PID gains for the controller will be found in *param_6_controller.m*.

In addition to structure P, two additional structures are created. These are B and *map*, created by respectively *param_battery.m* and *param_city.m*. They contains the drone's battery parameters and the city landscape. The call of these functions needs to be done at the start of each *main* functions described in section 4.5. For example the lines shown below in listing 5 are at the start of *main_path_planner_12*.

```
param_12_path_planner;          % creates P
param_battery;                  % creates B
```

```
param_city;                        % creates map
```

Listing 5: Call of parameter functions

The swarming parameters are contained in three different files which create the stucture *S*. *param_flock.m* has the principal parameters useful regardless of the algorithm used. For example, these are the migration direction and speed or the maximum number of neighbors. The parameters specific to a certain algorithm are located in another file, respectively *param_vicsek.m* and *param_olfatisaber.m* for the two algorithms.

4.5 Different *main* functions

The simulator has different *main* functions for different uses. They are as listed below:

1. *main_controller_6*
2. *main_path_follower_10*
3. *main_path_manager_11*
4. *main_path_planner_12*
5. *main_flocking*
6. *main_GUI*

The first four files corresponds to respectively chapters 6, 10, 11 and 12 in *Beard and McLain*'s book [1]. Each main adds functionality onto the previous one. The GUI main is mainly used to test the graphical user interface functionality and doesn't add any functionalities over the other scripts.

We will describe each of the listed *main* functions below.

4.5.1 *main_controller_6*

This main tests the drone controller. In this use case, no path is involved. It also simulate a unique drone.

A command is created by the function *generateCommand* and the drone follows it. This function takes as an input drone type (quadcopter or fixed wing) as well as autopilot mode in the case of a quadcopter (autopilot in attitude, velocity or acceleration). You may need to enter these two values manually when prompted at the start of the simulation.

For example in the case of a quadcopter controlled with velocities, the given command alternates with a period a 6 seconds between fixed values of 6 and -6 m/s in east velocity.
It is possible to change the commands given in *generateCommand* to see the drone comportment.

The output is a graph of the moving drone, centered on its center of mass. An example with a simulation end time of 20 seconds is shown in figure 5.

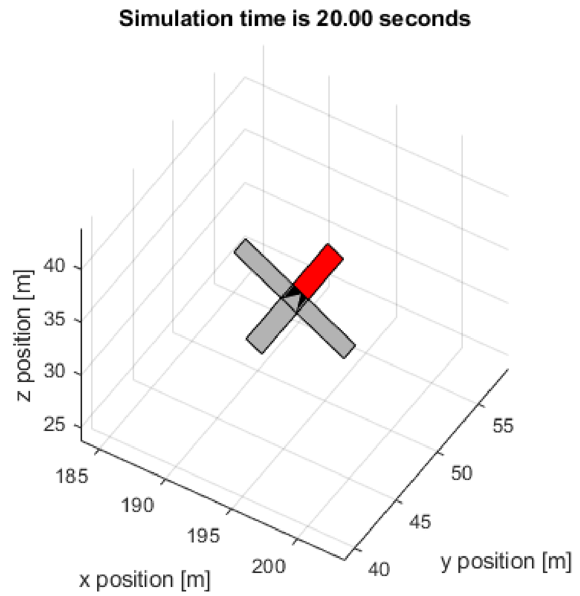


Figure 5: Drone representation

In all *main* scripts, the function *plot_uav_state_variables* can also be called to plot the state values relative to time, as shown in figure 6. However this second plot slows the simulation execution time so it is preferable to use it for debugging only. Note that here in figure 6 only the tab "Velocity" is shown, but the figure also contains tabs for the position, acceleration, air-data, attitude, angle rates and actuators. For concerned variables, it plots the real value in blue as well as the commanded one in red and the one read by the sensor model in green.

The *debug_plot* check-box in the user interface (see section 4.7) can be used to use this function. It can be deactivated during the simulation and restarted later one. However, the simulation needs to be first launched with the check-box active to work.

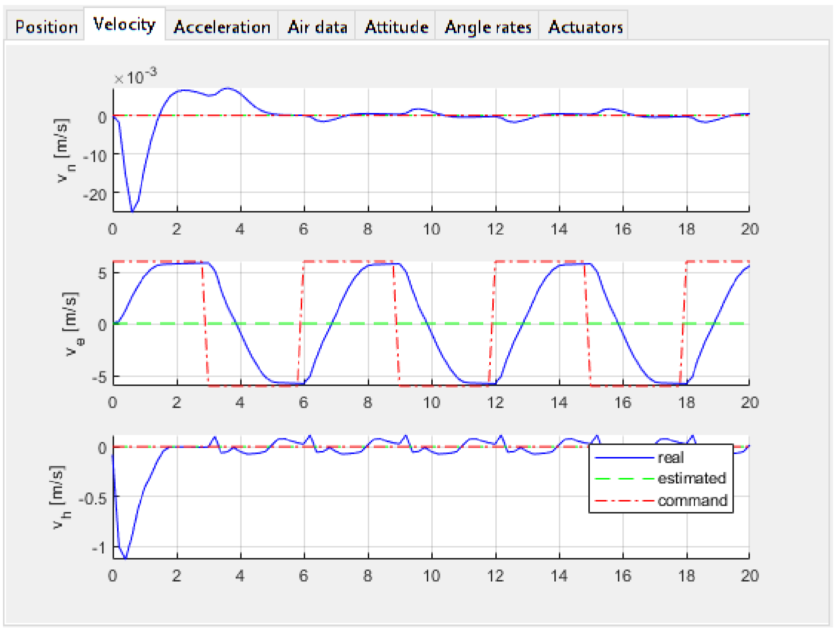


Figure 6: State plot

These graphs are useful when testing a new controller for response time, percent overshoot and other criteria. As it can be used with the different autopilot versions described in 4.2, this main can be very useful in the design of a controller.

We can for example see that our drone takes almost two seconds to change from -6 to 6 m/s. It also has a behavior we would want to limit (vertical speed of 1m/s) during the first second of simulation. This controller could be improved to reduce this.

4.5.2 *main_path_follower_10*

This main is the first to add a path to the drone. The path is created using the function *pathManagerQuadChap10*. This function was created for the `Simulink` simulator for the simulation of the book chapter 10, corresponding to this *main*. It created a simple heading which doesn't change as the drone moves.

The second difference is in the plotting functions. This time the external environment can be active. This is set by a boolean variable: *is_env_active*. If this boolean is set to *true*, the function *drawBuildings* will be called. As a result, the view won't be centered on the drone, but will be a fixed view of the city's building and how the drone moves inside. This is useful to see the drone's generated path as a whole.
You can see the path generated by function *pathManagerQuadChap10* in red. This path is only a straight line from the initial position in direction [2;1;0]. Note that the waypoints are already defined and plotted (in blue) but are not used for now.

This *main* can be used to test if the drone manage to follow its path without too much error. Adding a measure of deviation could be useful to determine if the path following algorithm is optimal.

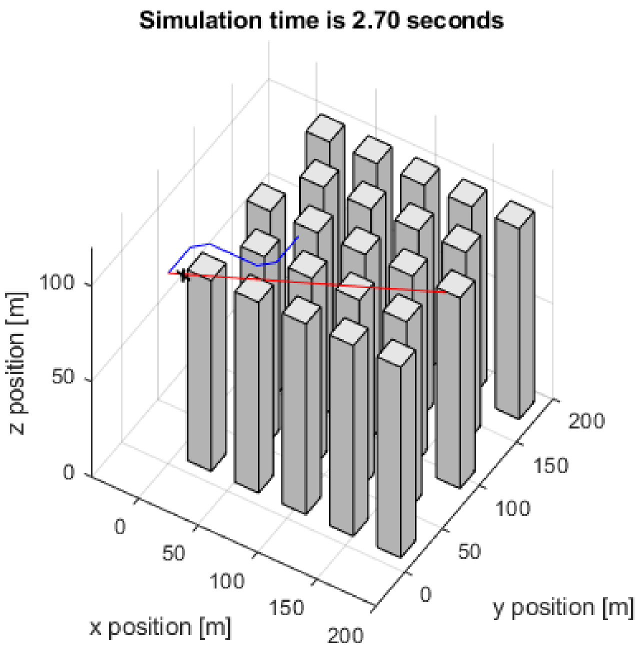


Figure 7: *main_path_follower_10* with a path in direction [2;1;0]

4.5.3 ***main_path_manager_11***

The path manager is added here. It creates a path from given waypoints set by a unique call to the path planner. Called with a *path_type* of 1 or 2, the path planner returns a list of waypoints corresponding to respectively a Fillet or Dubins path.

For example, calling it with a path type of 2 (Dubins path) will return the waypoint list shown in listing 6 and figure 8. The waypoints structure is in order, north position, east position, altitude (negative for above ground), yaw angle at waypoint and velocity at waypoint. Both the number of waypoints and waypoints are stored inside the *Drone* class and are specific to each drone.

```
nb_wpts = 6;
wpt_list = [0,  0,   -100, 0,          P.Va0;...
            30, 0,   -100, 45*pi/180,  P.Va0;...
            40, 10,  -100, 90*pi/180,  P.Va0;...
            40, 50,  -100, 45*pi/180,  P.Va0;...
            50, 60, -100, 0*pi/180,  P.Va0;...
            80, 60, -100, 45*pi/180, P.Va0];
```

Listing 6: Path returned by a call of *path_planner.m* with path type 2

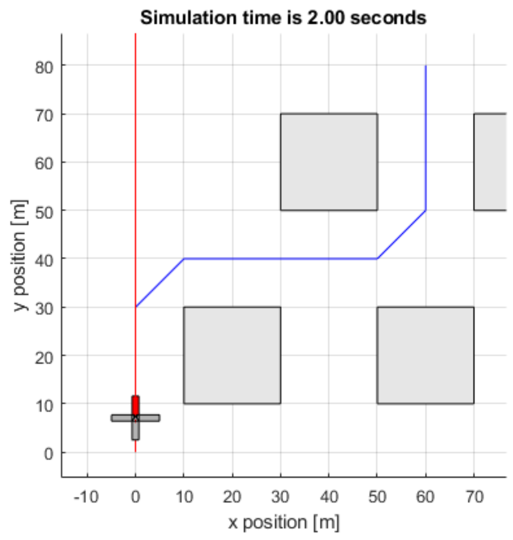


Figure 8: Path set by code lines of listing 6, seen from above

The *main* will then call the manager to decide which waypoint to follow based on current progress. This will return two waypoints and a path will be created in order to link these. The path follower will then be called to create the actuators' commands.
Please note that when the drone reaches the last waypoint it won't stop and will follow its last heading until infinity.

4.5.4 *main_path_planner_12*

The path planner adds automatic waypoint generation to the previous *main*. It enables the drone to create its waypoints based on a map and on a goal point. The map contains building that the drone needs to avoid.

A first example of the algorithm used is shown below in figure 9.
In this case, both the starting and ending points are above the buildings

(height of 110m).
The algorithm used is the "rapidly-exploring random tree", or RRT. This algorithm consists of growing a tree-like structured rooted on the staring point. For each iteration, one branch grows in a random direction. The length is fixed based on the number of iteration (smaller branches for the last iteration). If the branch is feasible, in this case if it doesn't interest any building, it is drawn. Otherwise the algorithm switches to the next branch. In the graph of figure 9, we can see then main branches in red and the sub-branches in green.

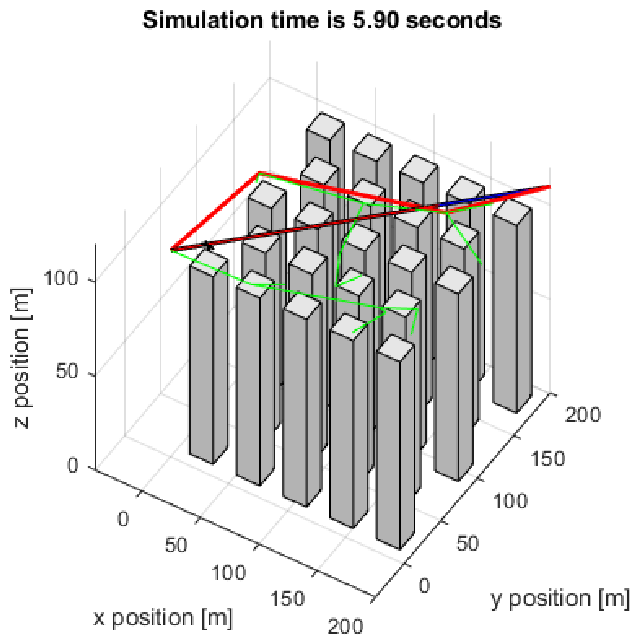


Figure 9: Output of function *main_path_planner_12*

When the algorithm reaches the end point, it checks if it could have done the path in a shorter way. If that's the case, it uses this alternative path. On the graph, this is the final path shown in blue, as a direct line doesn't intersect any buildings in this case.

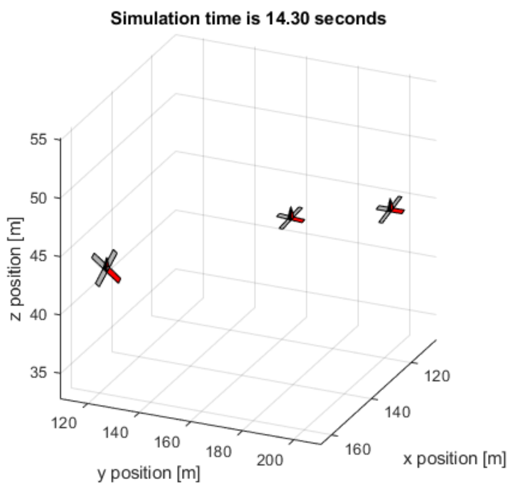
Unfortunately, the algorithm doesn't manage to link the two waypoints when their height is under 100m (inside the buildings). Some parameter tuning such as changing the branches length could probably help.

4.5.5 *main_flocking*

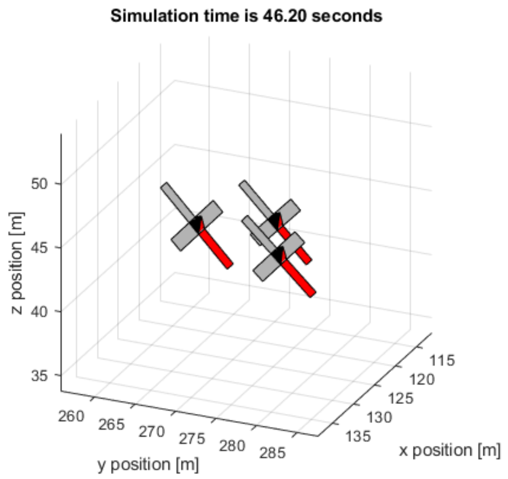
This *main* simulate the comportment of a swarm. It can be with both Vicsek or Olfati-Saber [2] algorithms. This can be changed by setting the parameter *flocking_algo* to 0 (Vicsek) or 1 (Olfati-Saber).

In this *main*, the actuators' commands are computed using the flocking algorithm. For a quadcopter, the velocity autopilot is used.
In addition to that, obstacles can be added. The goal of these obstacles ("spheres") is be to "squeeze" the flock through two obstacles. Doing this, we could check the collisions both between the drones themselves and with the obstacles.

First if we run with three drones and no obstacles (see figure 10), we see that they align with a common "going forward" goal. They also keep a fixed separation distance between themselves by aligning on a proximity net as described in *Olfati-Saber* [2].



(a) Before flock formation



(b) After flock formation

Figure 10: Execution of *main_swarming* with three drones

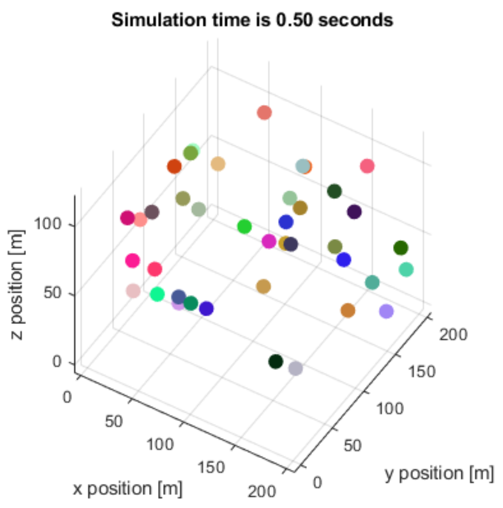
We can also launch the simulation with more drones. To limit the execu-

tion time, they are shown here in limited mode (see section 4.6).
Doing so enables us to observe the three Reynolds rules described in *Olfati-Saber* [2]:

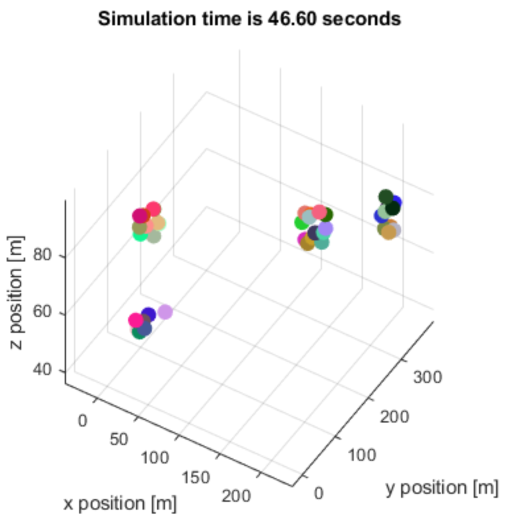
1. flock centering: the drones form small flocks, four in the example of figure 11. In each of these flocks the drones try to get close to each other

2. collision avoidance: no collisions happen inside the flock as each of the drone pair has a repulsion force separating them

3. velocity matching: the common motion of ”going forward” is seen by the global movement on the y axis, mainly the two swarms on the right moved from y between 120 and 200 to y between 250 and 350



(a) Before flock formation



(b) After flock formation

Figure 11: Execution of *main_swarming* with 40 drones

At a maximum speed of 5m/s and with a runtime of 45 seconds as shown in figure 11b, a drone randomly initialized in [0;200] would be in [225;425]. The drones shown in figure 11b are located between 100 and 350, which shows a moving swarm speed of at least 2m/s. As the 45 seconds include the time while the swarms form themselves, these values are expected.

4.5.6 *main_GUI*

This *main* is used to test the GUI functionalities during development. The simulated drones take command the same way as *main_controller_6*. However a swarm can be simulated, which was not possible there. As a result it is possible to simulate a flock all moving according to the same commands. In addition to that, it can show the environment or not which was not possible in the controller *main*.

4.6 Simulation modes

Two simulations modes are available for a quadcopter with velocity-based autopilot. This is because the simulation is slow with multiple drones and in some occurrence it is not needed to have a very precise simulator. In these cases, the simulation can be run with limited precision in less time.

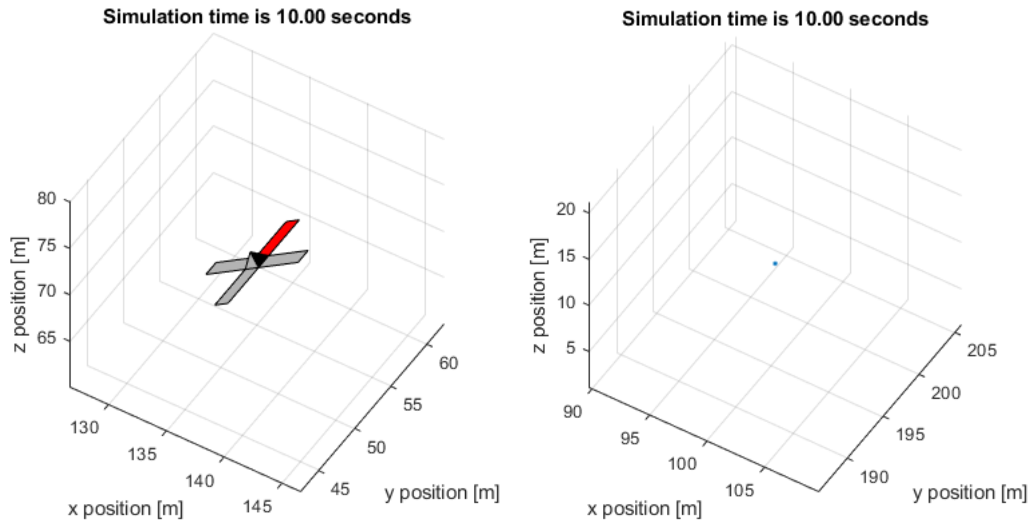
The simulation mode is set by the boolean defined in the main: *realistic_sim*. The changes in the *main* files are shown in listing 7 below.

```
if realistic_sim   % realistic version
    drone.updateState(wind, time);
    drone.drawAircraft(time, period_fig, fig_handle, axes_lim);
else               % time efficient version
    drone.updateStateEulerForward();
    drone.drawBoid(time, period_fig, fig_handle, axes_lim);
end
```

Listing 7: Changes set by boolean *realistic_sim*

The state update function is different. In the case of a time efficient simulation, the state is updated using forward Euler method. The new position is obtained with $pos = old\_pos + command * T$. As the command is in velocity, we consider that the drone will immediately react to a new command by moving at the commanded speeds.
In addition to that the wind is not used in the case of a limited representation. We consider that the drone autopilot will perfectly compensate for any perturbations.

The plotting functions also change with this boolean. The different outputs are shown in figure 12 below. We can see that the representation of figure 12b is limited as it can't display current attitude or heading. However, in most case only the position of the drone is needed and the heading can be understood based on the current swarm movement. Note that the plot shown in figure 12b is not really useful in the case of a single drone. However for a swarm it makes more sense. The drone's initial position is set randomly. The two functions called are named *drawAircraft* and *drawBoid* and are methods of the *Drone* class. The same functions are available in the *Swarm* class, namely *drawAircrafts* and *drawBoids*.



(a) Accurate representation (aircraft) (b) Simplified representation (boid)

Figure 12: Output of functions *drawAircraft* and *drawBoids*

For one drone, running the function *main_controller_6* with the boolean set to *true* (accurate simulation) takes 2.7[2] seconds to simulate 10 seconds of flight. When the boolean is switched to *false*, it takes 2.1 seconds. This represents a reduction of approximately 20% in simulation time.
When running a more complex simulation (up to the path manager, chapter 11), the time goes from 8 seconds to 5.8 seconds to simulate 20 seconds (time required to go to the end of the waypoints described in figure 8). This is a decrease of 27.5%.

[2]All measures were done on a computer equipped with a Intel Pentium G4560 and a graphical card GTX 1050Ti.

For one drone and a short simulation, these are small reductions. However if the simulation needs to run many parallel instances, it can make an important difference.

We can test this difference by running *main_GUI* with 50 drones with the environment active. The program takes an average of 11 seconds to run the simulation for one second in realistic mode. In limited mode, it takes less than one second. In this scenario, the time reduction was more than 90%!

As the simulation is faster than real time in limited mode, the use of a remote-controller becomes possible. This adds many possibilities to the simulator. It could for example be possible to control the swarm direction in real time.

4.7 Graphical User Interface

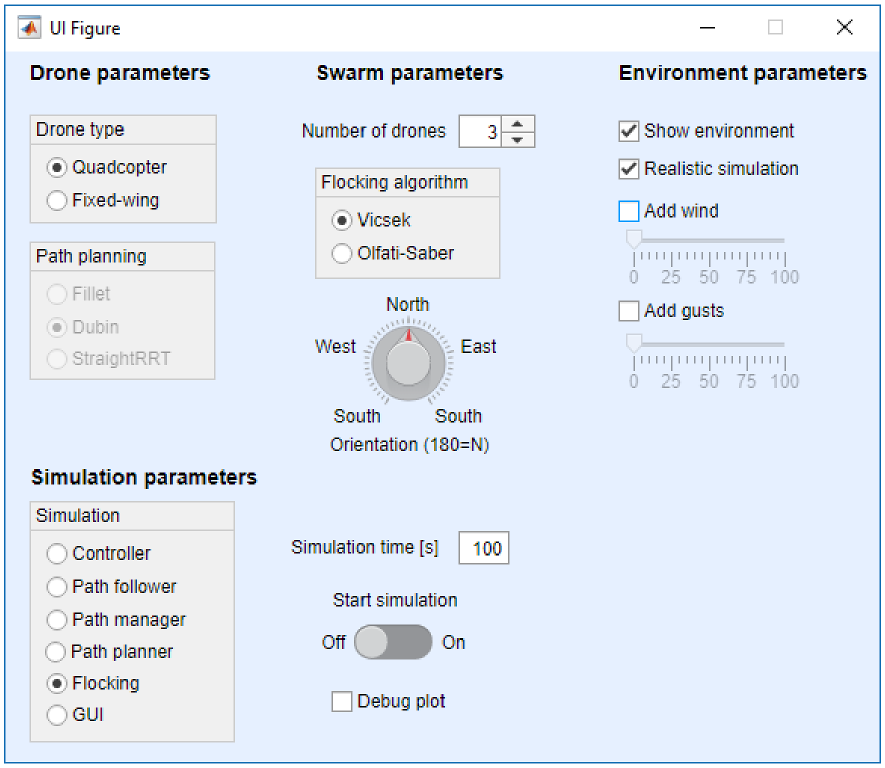
The goal of the creation of a graphical user interface (GUI) is to enable us to change different parameters easily. Some parameters can be changed during the simulation (e.g. wind) while some others need to be set before launching it.
The GUI was created using Matlab `App Designer` toolbox, producing a *.mlapp* file.

The different variables we decided to add to the GUI are as follows:

- Variables that need to be set before launching the simulation
  - Which simulation to run: decide between the different *main* presented in 4.5
  - Number of drones, only for flocking simulation
  - Type of drones: quadcopter or fixed-wing
  - Type of path, for path manager and planner only
  - Flocking algorithm used, Vicsek or Olfati-Saber
  - Environment parameters: realistic simulation or not (see section 4.6), draw environment or not

- Variables that can be set during the simulation
  - Wind parameters: level of steady wind and gusts
  - Swarm migration orientation

The produced GUI is shown below in figure 13.



Drone type: ● Quadcopter ○ Fixed-wing

Path planning: ○ Fillet ● Dubin ○ StraightRRT

Number of drones: 3

Flocking algorithm: ● Vicsek ○ Olfati-Saber

Orientation (180=N)

☑ Show environment
☑ Realistic simulation
☐ Add wind
☐ Add gusts

Simulation: ○ Controller ○ Path follower ○ Path manager ○ Path planner ● Flocking ○ GUI

Simulation time [s]: 100

Off ● On

☐ Debug plot

Figure 13: Created GUI

We can see that the parameters are separated in four categories: drone, swarm, environment and simulation. Some parameter won't be possible to change in some simulation modes. For example the orientation dial controls the swarm migration, and will be active only when the chosen simulation is the flocking one.

The Matlab script corresponding to the chosen simulation is launched directly from the GUI. Each time a parameter changes inside of it, a callback function is called. This callback sets a parameter inside the workspace variable *app*. At each point of time inside the simulation, the code checks if any parameter changed. If any did, the corresponding variable contained in the script is changed to its new value.

4.8 Wind model

We tried to implement the Dryden wind model using the block found in the `Aerospace` toolbox[3]. This block takes as input the drone altitude, speed norm and DCM (rotation) matrix. It returns two 3x1 vector for the linear and rotational speeds.
The block itself is shown in figure 14.

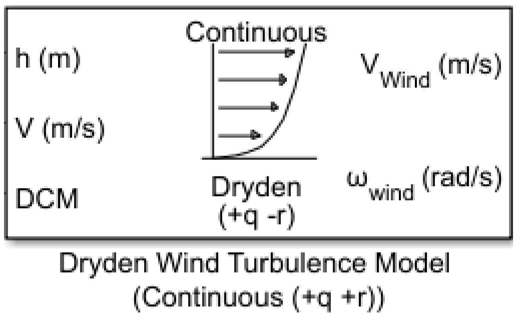


Figure 14: Dryden block

The issue we encountered is that the wind needs to be computed for each drone at each point of time. Coupled with the fact that a `Simulink` model needs to run parallel to the main simulation, this implementation was too heavy on computation which was not acceptable.

The solution we found was to use a turbulent wind model found on the Mathworks community website[4]. This model was created to use in wind turbines by user *PEF*.
Its main input are as follows:

[3]`https://ch.mathworks.com/help/aeroblks/drydenwindturbulencemodelcontinuous.html`

[4]`https://www.mathworks.com/matlabcentral/fileexchange/54491-3d-turbulent-wind-generation`

- average wind speed [m/s]
- turbulence intensity [%]
- size of X-Y grid [m]
- time step length [s] and simulation time [s]

It returns a wind field of dimension 4. The first dimension is for each time step in the simulation, the second and third are the X and Y coordinates, and the last is a 3x1 vector corresponding to the wind at each coordinate and time step. The wind returned is in order z-x-y.
We decided to use this model with an average wind speed of 5 m/s.

For the gust generation, we used a random values generated using a normal distribution. The mean is 0 is all x-y-z directions and we used a standard deviation of 0.3. The values were then multiplied by a maximum value of 2 m/s. As a result, the gusts will be most of the time between -0.6 and 0.6 m/s. These values would need to be tuned to match more closely a real life situation.

Both the steady wind and gusts are then modified using the variables *wind_level* and *wind_gust_level* found in the GUI. They act as a modification in percentage of the obtained value. A *wind_level* of 50 will reduce the obtained values by a factor 2.

It is also possible to completely deactivate the wind or gusts separately from the GUI.

This wind model could still be improved. For now it doesn't consider the drone altitude. We could simply add a factor depending on its altitude: the wind is the computed value at 100m, then decreases gradually to reach 0 m/s on ground level. The gusts also are not dependent on position, only on random values. As a result, two drones side by side could have gusts in opposite direction.

Another drawback of this model is that it takes time to compute the field when launching the simulation. For one drone, it took between 3 and 6 seconds. Even with these flaws, this wind model enables us to test the drone comportment in a wind-field. This brings it closer to a realistic simulation.

5 Discussion

Overall, this project was both interesting and returned a consequent addition on the previous simulator. Quite some time and effort was spend on features that were not fully implemented in the end, such as a first draft of the new control method or the Dryden wind model that would still be helpful for future improvements. However, the result was still satisfactory.

Future work on the simulator could include some of the following points:

- implementing a new control method using the H_∞ method described in Massé et al. [3] to improve the robustness against perturbations such as wind
- implementing a better wind model, which could be done using a look-up table with the drone coordinates, the wind would need to be defined for the entire map beforehand
- changing the path follower to a more precise one in order to reduce the difference between the path and the drone position
- adding the sensor model that was present in the `Simulink` simulator
- optimizing the RRT waypoint generation to be able to find a path through the city
- adding the compatibility of a remote controller to control the swarm

6 Conclusion

In the end we have a versatile simulator capable of working in multiple scenarios. It can be used to test low-level control methods as well as higher-level flocking algorithms.
As the files to launch the different simulations are separate, it makes it possible to test features relative to these different levels separately.

The addition of a graphical interface makes it simple to use and adds the possibility to change variables easily directly during the simulation execution. Choosing between the two different modes enable the simulator's user to decide whether they prefer to use a realistic simulation with the limitation of the number of drones or if they prefer the ability to simulate dozens of drones in a more limited view. With this the simulator becomes versatile and can be used by both people studying drone control where a realistic simulation is needed, or swarm dynamics where the priority is to simulate multiple drones.

References

[1] Randal W. Beard and Tomothy W. McLain. *Small unmanned aircraft: theory and practice.* 2012.

[2] Reza Olfati-Saber. *Flocking for multi-agent dynamic systems: algorithms and theory.* IEEE transactions on automatic control. 2006.

[3] Catherine Massé, Olivier Gougeon, Duc-Tien Nguyen and David Saussié. *Modeling and control of a quadcopter flying in a wind field: a comparison between LQR and structured H_∞ control techniques.*

[4] LIS-EPFL. *Versatile simulator for a Swarm of Quadcopter.* 2018.

A Drone kinematics equations

The function *kinematicsOde_f* below sets the equation system to solve for the drone kinematics.

```
function dxdt = kinematicsOde_f(t, x, xx, uu, P)
% Defines the equation system for the drone kinematics

% Speed and position
vx    = x(4);
vy    = x(5);
vz    = x(6);
p     = x(10);
q     = x(11);
r     = x(12);

% Forces and moments
fx    = uu(1);
fy    = uu(2);
fz    = uu(3);
l     = uu(4);
m     = uu(5);
n     = uu(6);

% Orientation
phi0   = xx(7);
theta0 = xx(8);
psi0   = xx(9);

% Trigonometry
cr = cos(phi0);
cp = cos(theta0);
sr = sin(phi0);
tp = tan(theta0);

% Rotation matrix from inertial frame to body frame
Rbi = Rb2i(phi0, theta0, psi0);

% Velocity in inertial frame
pos_dot = Rbi* [vx vy vz]';
pndot = pos_dot(1);
pedot = pos_dot(2);
pddot = pos_dot(3);

```

```
% Acceleration in body frame
vxdot = r*vy - q*vz + fx/P.mass;
vydot = p*vz - r*vx + fy/P.mass;
vzdot = q*vx - p*vy + fz/P.mass;

% Rotation matrix for rotation rate
Si_b = [1, sr*tp, cr*tp;    ...
        0, cr,    -sr;      ...
        0, sr/cp,  cr/cp];

% Rotation rate in bodyframe
phidot   = Si_b(1,:) * [p q r]';
thetadot = Si_b(2,:) * [p q r]';
psidot   = Si_b(3,:) * [p q r]';

% Rotation acceleration in body frame
pdot = P.gamma1*p*q - P.gamma2*q*r + P.gamma3*l + P.gamma4*n;
qdot = P.gamma5*p*r - P.gamma6*(p^2-r^2) + m/P.Jy;
rdot = P.gamma7*p*q - P.gamma1*q*r + P.gamma4*l + P.gamma8*n;

% Output
dxdt = [pndot, pedot, pddot, vxdot, vydot, vzdot, ...
          phidot, thetadot, psidot, pdot, qdot, rdot]';

end
```

Listing 8: Drone kinematics equations