A Versatile Simulator for a Swarm of Drones

Author:
Enrica Soria
Contact:
enrica.soria@epfl.ch

Credits to:
Prof. Randal W. Beard
Prof. Timothy W. McLain
Victor Delafontaine
Anthony De Bortoli
Prof. Dario Floreano

June 25, 2019

Contents

A Versatile Simulator for a Swarm of Drones

Abstract

This Matlab simulator was created with versatility in mind. Its goal is to be able to simulate multiple scenarios depending on the user requirements: either a single drone or a swarm of hundreds of drones. This project extends the work provided by Beard and McLain in [1], focused on reproduction of the flight of a single fiexed-wing drone. Building on it, this simulator provides a useful tool to test drone and swarm dynamics. An example is shown in figure 1 where three drones fly together after 46 seconds of simulation.

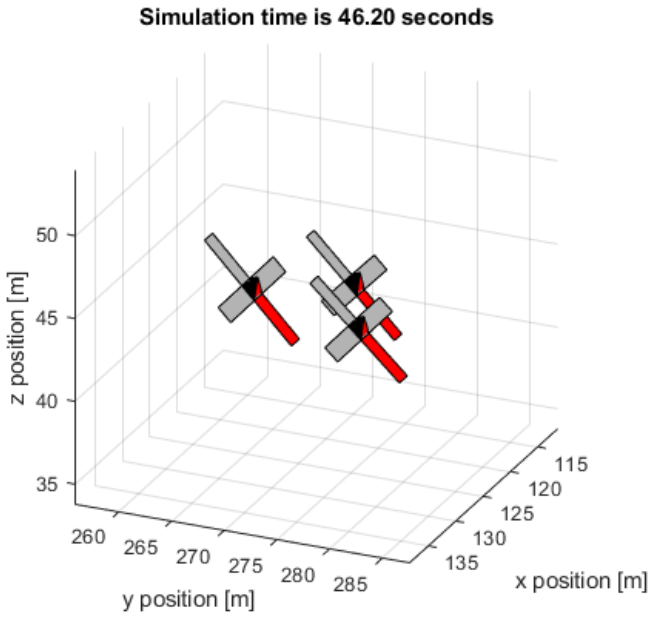


Figure 1: Three drones flying together

This simulator includes a graphical user interface (GUI) for ease of use and the possibility to choose between two different simulation modes, depending on the user requirements (realism or speed).

1 State of the art

1.1 Randal Beard and Timothy McLain: *Small unmanned aircraft: theory and practice*

This book was first published in 2012 by two professors at University of Princeton, Randal W. Beard and Timothy W. McLain. It describes in detail the structure of the simulation of a drone, more specifically a fixed-wing. With the book, the authors released a set of templates that allow the reader to practice all the topics explained in the book. These are available at:

https://github.com/randybeard/mavsim˙template˙files

The first chapter is on the drone's basic coordinate frames. In this section, the authors describe the Euler angle representation, the NED (north-east-down) reference frame as well as the airspeed model. These basics allow to know how to represent the drone in the three dimensional space.

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 this simulator. It is represented as follows:

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

The four triplets are position in NED frame, velocity in body frame, attitude (roll, pitch and yaw) and attitude rate, in the order.
The kinematic equations are essential to get the evolution of the drone state 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 focused on the description of the autopilot, which is divided in two parts: lateral and longitudinal autopilots. Note that this is only described for a fixed-wing UAV. The autopilot structure will be differ for a quadcopter, as its control and structure are. The autopilot described in the book 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. The Kalman filtering is used for state estimation. While templates for this are provided at the link before, it is not incorporated into our simulation.

The following chapters are on the guidance model. Once again, this is mainly focused on fixed-wings and varies for a quadcopter. The guidance model is staged onto three layers. The first is a straight-line and orbit follower. 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 the waypoints by the path manager. These will be explained in more details later in this project in section 3.3.

2 Simulator structure

The simulator structure follows the waterfall structure described in *Beard and McLain*[1]. It is shown in figure 2 below. This structure is valid for a single drone. For multiple drones, the guidance model (path planner, manager and follower) will be replaced by a swarm version. The input commands for the drones will come from the swarming model based on the drones' positions and velocities and on the obstacles.

The book was used as a base to build our simulator. Some of the functions' structure presented in the `Simulink` simulator were taken directly from the open-source templates suggested by the authors. In figure 2 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 templates were implemented for a fixed-wing drone. The first step was to convert the simulator up to the autopilot to Object Oriented Programming (OOP). Then, equivalent blocks for the quadcopter were added. Finally, the swarm object was created to allow the simultaneous simulation of multiple drones.

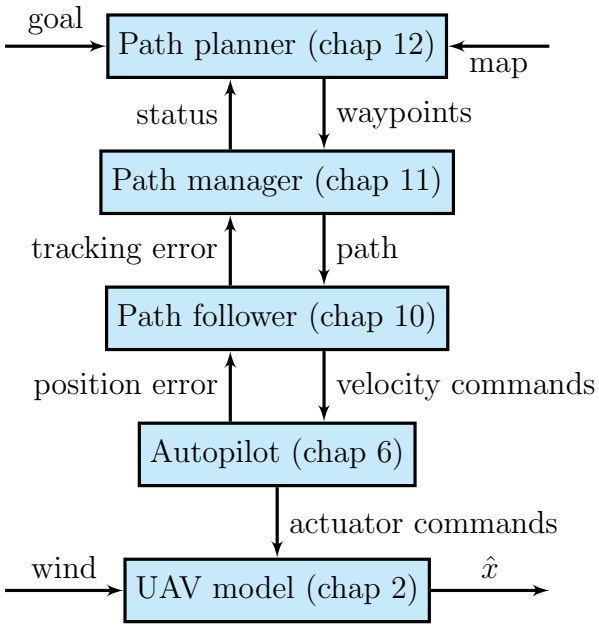


Figure 2: Simulation of a single drone: waterfall structure

3 Documentation

This section explains the functioning of the simulator and how to use it. The main objects of interest are the followings:

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 functions;
3. the guidance model used in the simulation, as explained by *Beard and McLain* [1];
4. the different parameter files and how to modify them;
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.

3.1 Drone and Swarm 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_type*: 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 3.2
  - *P*: set of general parameters P defined by the parameter file relative to the current simulation (see section 3.4)

  - 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 3.2 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 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
- *equivalent_drone*: 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
- *swarming_algo*: the type of swarming algorithm used to command the agent of the swarm

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 1 and 2.

```
% Drone initialization
drone = Drone(DRONE_TYPE, AUTOPILOT_VERSION, REALISTIC_SIM, P, B
    , map);
```

Listing 1: *Drone* object creation

```
% Swarm initialization
swarm = Swarm(SWARMING_ALGO);
for i = 1 : S.nb_agents
swarm.add_drone(Drone(DRONE_TYPE, AUTOPILOT_VERSION,
    REALISTIC_SIM, P, B, map));
end
```

Listing 2: *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 3. 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.

```
% Set swarm position
seed = 5; % fixed seed to avoid randomness
rng(seed);
pos0 = repmat(map.width, 1, S.nb_agents) .* rand(3,S.nb_agents);
swarm.setPos(pos0);
%pos0(3,:) = -pos0(3,:)*0.6;
```

Listing 3: *Swarm* position initialization

3.2 Autopilot versions

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

The autopilot is distinct for fixed-wings and quadcopters. The functions are respectively *autopilot_wing.m* and *autopilot_quad.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 *pid_loop* 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.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 needed is the velocity one. Attitude, velocity, acceleration and position autopilot corresponds to values of respectively 1 up to 4.

3.3 Guidance model

The simulator's guidance model for a single drone is composed of three functions. They are the path planner, manager and follower. Some differences distinguish a quadcopter from a fixed-wing. The main is that a fixed-wing drone needs to perform circular turns and cannot change yaw without moving forward at the same time. For this reason, the path has two options: straight line or circular turn. 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 35 m/s and 5 m/s.

The three functions described above are staged onto three levels. These levels are also shown in figure 3 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. Please, refer to chapter 10 of the reference book for details on the theory.

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. Please, refer to chapter 11 of the reference book.

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. Please, refer to chapter 12 of the reference book.

We created different *main* scripts to test these functions. These can be found later on in section 3.5 and can be run separately to test the different stages.

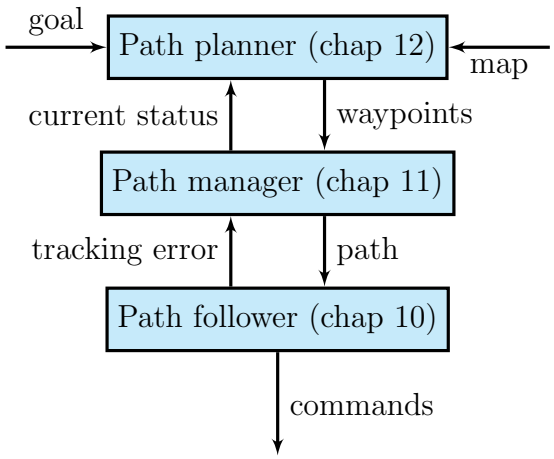


Figure 3: Guidance model structure

3.4 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 3.5. For example the lines shown below in listing 4 are at the start of *main_path_planner_12*.

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

Listing 4: Call of parameter functions

The swarming parameters are contained in a different file which create the stucture S. *param_swarm.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.

3.5 Different *main* functions

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

1. *main_controller*
2. *main_path_follower*
3. *main_path_manager*
4. *main_path_planner*
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 functionality over the other scripts.

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

3.5.1 *main_controller*

This main tests the drone controller. It simulates a unique drone and, in this use case, no path is involved. A command is created by the function *generate_command* and the drone follows it. This function takes as an input drone type (quadcopter or fixed-wing drone) as well as autopilot mode in the case of a quadcopter (autopilot in attitude, velocity or acceleration). You may need to enter the first input 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 *generate_command* 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 4.

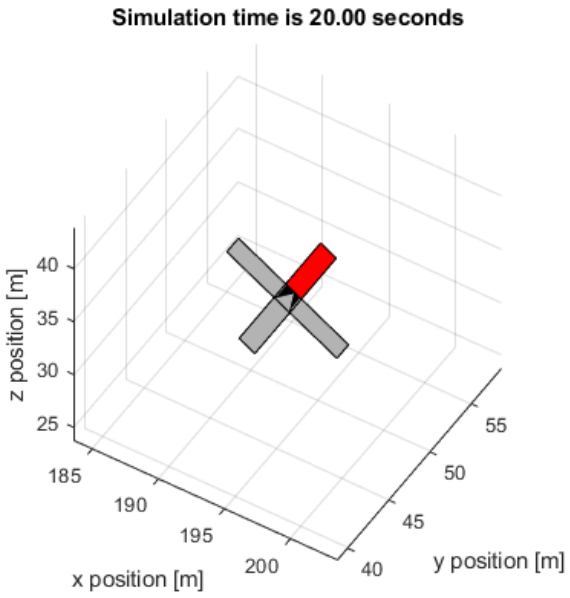


Figure 4: Drone representation

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 5. However this second plot slows the simulation execution time so it is preferable to use it for debugging only. Note that here in figure 5 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 estimation from the sensor values in green. The computation of this last value has still to be coded.

The *debug_plot* check-box in the user interface (see section 3.7) 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.

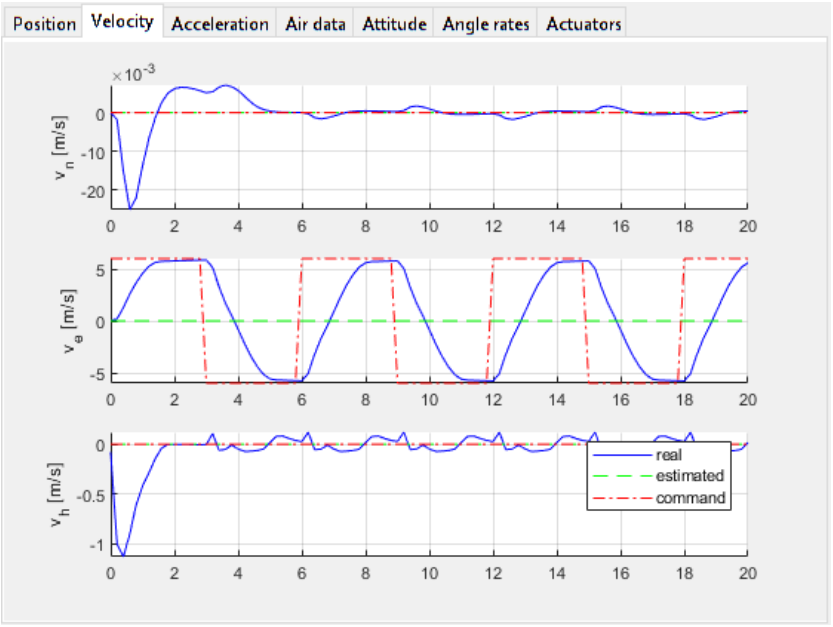


Figure 5: State plot

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 3.2, 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.

3.5.2 *main_path_follower*

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 *path_manager_quad_chap10* 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.

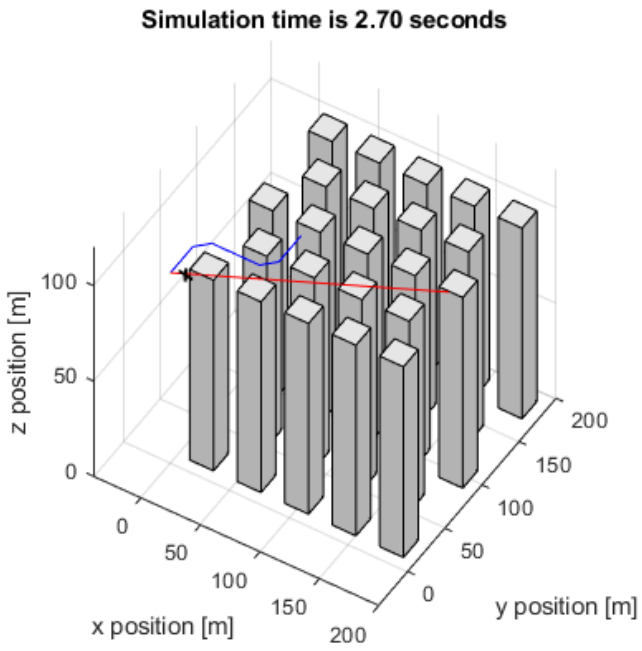


Figure 6: *main_path_follower_10* with a path in direction [2;1;0]

3.5.3 *main_path_manager*

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 5 and figure 7. 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.

```
1 nb_wpts = 6;
2 wpt_list = [0, 0,   -100, 0,          P.Va0;...
3             30, 0,  -100, 45*pi/180, P.Va0;...
4             40, 10, -100, 90*pi/180, P.Va0;...
5             40, 50, -100, 45*pi/180, P.Va0;...
```

```
6            50, 60, -100, 0*pi/180,  P.Va0;...
7            80, 60, -100, 45*pi/180, P.Va0];
```

Listing 5: Path returned by a call of *path_planner.m* with path type 2

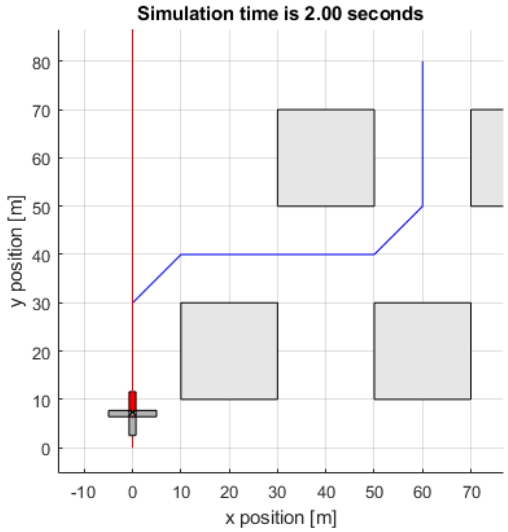


Figure 7: Path set by code lines of listing 5, seen from above

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.

3.5.4 *main_path_planner*

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 8. 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 (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 8, we can see then main branches in red and the sub-branches in green.

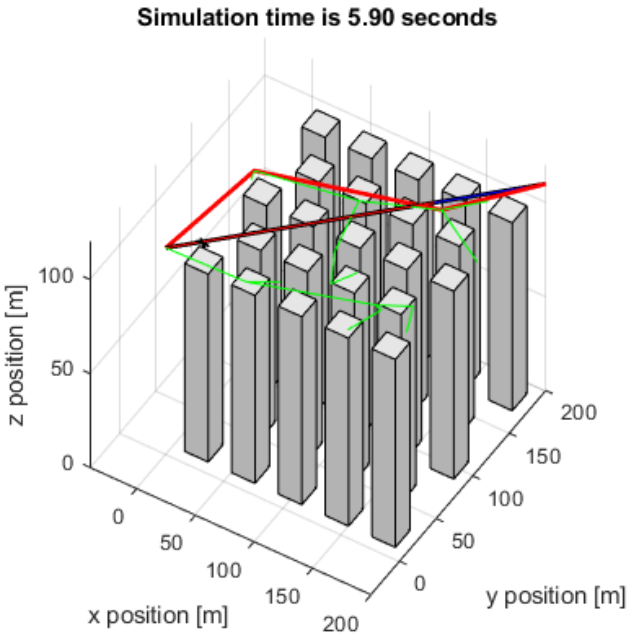


Figure 8: Output of function *main_path_planner_12*

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.

3.5.5 *main_swarming*

This *main* simulates the comportment of a swarm. It is currently in development.

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 swarm through two obstacles. Doing this, we could check the collisions both between the drones themselves and with the obstacles.

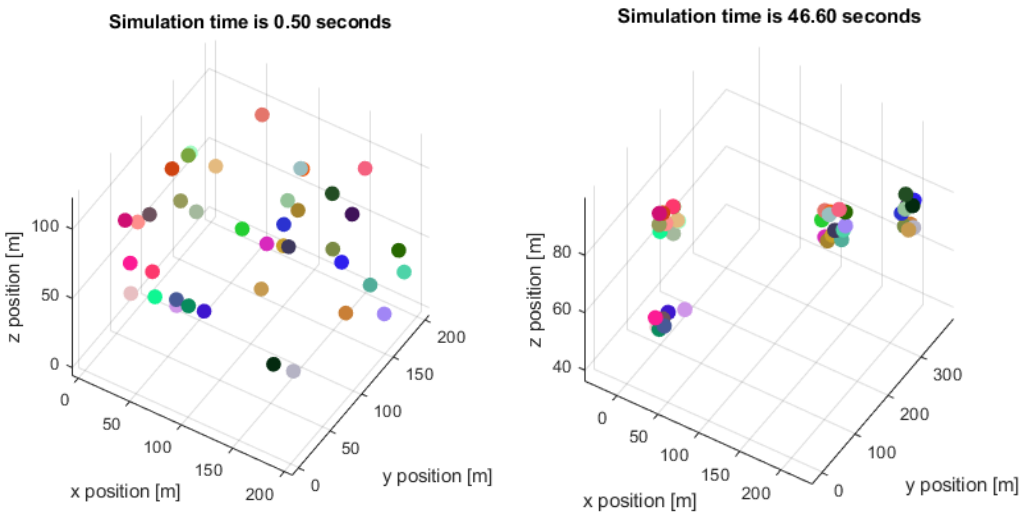
First if we run with three drones and no obstacles (see figure 9), we see that they align with a common "going forward" goal. They also keep a fixed separation distance between themselves by aligning on a proximity net.

(a) Before swarm formation (b) After swarm formation

Figure 9: Execution of *main_swarming* with three drones

1. centering: the drones form small flocks, four in the example of figure 10. In each of these swarms the drones try to get close to each other
2. collision avoidance: no collisions happen inside the swarm 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 10: Execution of *main_swarming* with 40 drones

At a maximum speed of 5 m/s and with a runtime of 45 seconds as shown in figure 10b, a drone randomly initialized in [0;200] would be in [225;425]. The drones shown in figure 10b 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.

3.5.6 *main_GUI*

This *main* is used to test the GUI functionalities during development. The simulated drones take command the same way as *main_controller*. 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.

3.6 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 dynamic simulation. 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*. This parameter affects the *update_state* function in the *Drone* class. The piece of code managing it is shown in the listing 6 below.

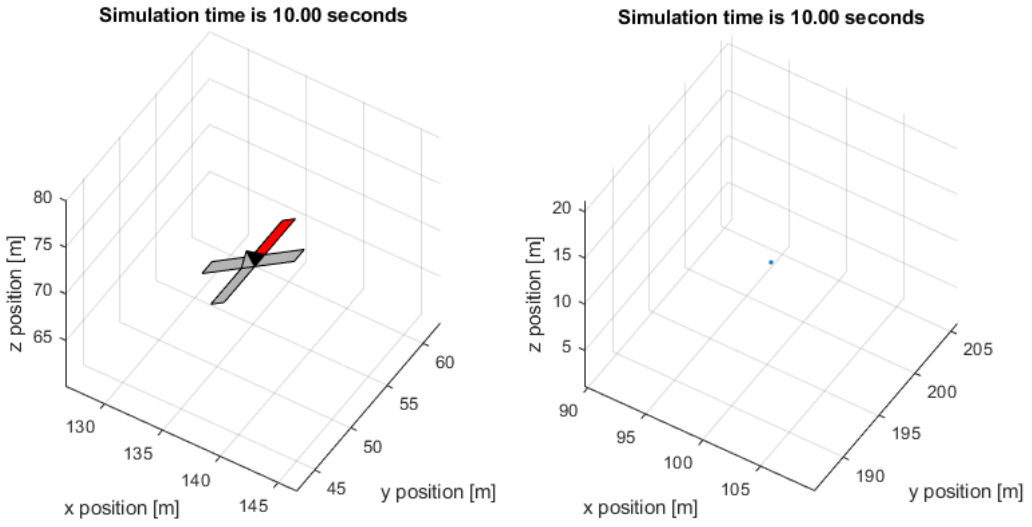
```
if obj.is_realistic == true
    % Choose the autopilot
    if obj.drone_type == 0% fixed-wing
        temp3 = autopilot_wing(obj, 0, time);
    else % quadcopter
        temp3 = autopilot_quad(obj, time);
    end

    obj.delta = temp3(1:4);
    obj.full_command = temp3(5:end);
    % Update true state
    obj.compute_dynamics(wind, time);
    obj.compute_kinematics(time);
    obj.update_battery();
else
    % Compute the new drone position with Euler forward method.
    % This method do not take the attitude into account.
    % We suppose that the attitude is always (0,0,0), so the
    % velocity in the body frame correponds to the velocity in
    the
    % inertial frame.
    obj.vel_xyz = obj.command(2:4);
    obj.pos_ned = obj.pos_ned + obj.vel_xyz * obj.P.dt;
    obj.attitude(3) = obj.command(1); % to plot drone psi angle
end
```

Listing 6: Changes set by boolean *realistic_sim*

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 speed. In addition to that the wind is not used in the case of a limited rep-

resentation. We consider that the drone autopilot will perfectly compensate for any perturbations.

The plotting functions can also change with a parameter *SWARM_VIEWER_TYPE*. The different outputs are shown in figure 11 below. We can see that the representation of figure 11b 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. The drone's initial position is set randomly.



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

Figure 11: Output of functions *drawAircraft* and *drawBoids*

For one drone, running the function *main_controller* with the boolean set to *true* (accurate simulation) takes 2.7[1] 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 7). This is a decrease of

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

27.5%. 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.

3.7 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 parameters in the GUI are:

- Variables that need to be set before launching the simulation
  - Which simulation to run: decide between the different *main* presented in 3.5
  - Number of drones, only for swarming simulations
  - Type of drone: quadcopter or fixed-wing
  - Type of path, for path manager and planner only
  - Swarming algorithm used: **more on this topic is coming out soon!**
  - Environment parameters: realistic simulation or not (see section 3.6), 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 12.

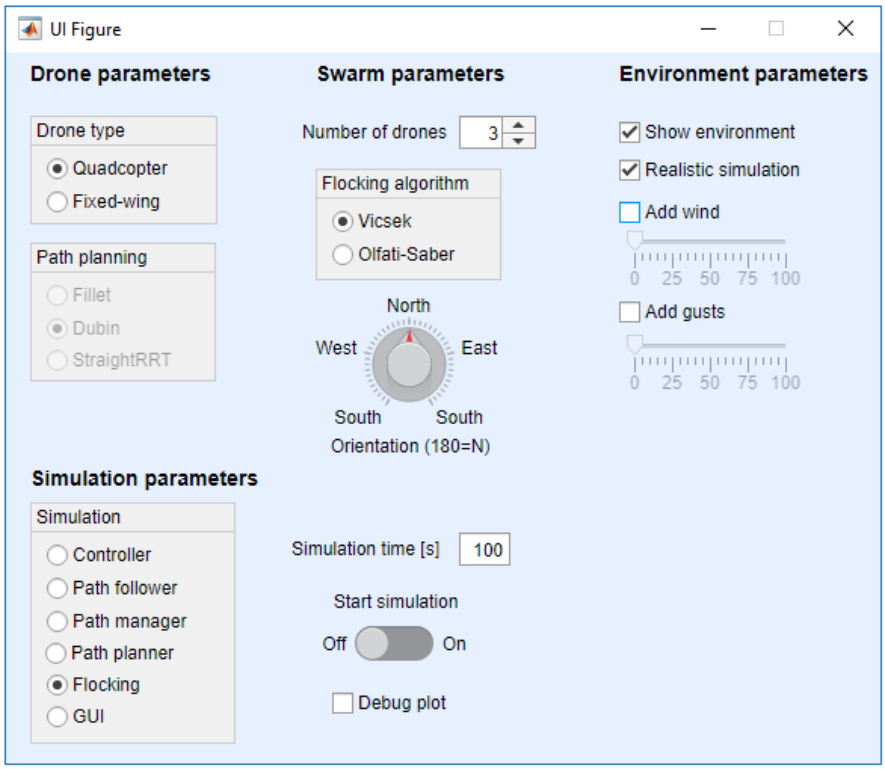


Figure 12: Created GUI

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 swarming 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.

3.8 Wind model

We use a turbulent wind model found on the Mathworks community website [2]. This model was created to use in wind turbines by user *PEF*.

Its main input are:

- 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 use this model with an average wind speed of 5 m/s.

For the gust generation, we used 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

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

wind or gusts separately from the GUI.

This wind model can still be improved. For now, it doesn't consider the drone altitude. The gusts also do not dependent on position, but they take 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.

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 7: Drone kinematics equations