NOMMON	User Manual	Ref	UM-USE-21001
		Version	3
		Date	11/01/2022

Title

BlueSky user manual

Keywords

USEPE, drone, simulator, BlueSky, D2-C2, UAM, separation method,

Summary

This document is intended to serve as a user manual for the simulation tool developed in USEPE project

Prepared by	Miguel Baena, Jerónimo Bueno	11/01/2022
Revised by		
Approved by		

Distribution

USEPE consortium

Record of revisions

Version	Date	Revision Description	Sections affected
1	29/10/2021	Initial version	N/A
2	29/11/2021	Add section describing the strategic deconfliction module Add section describing the wind pre-processing	2.1.1, 2.1.2, 2.3, 3
3	11/01/2022	Add section describing the performance model Add section describing the conflict detection module Add section describing the conflict resolution	2.2.1, 2.2.2, 2.2.3, 3

1. Objectives and scope

The tool presented here is developed as part of the USEPE project. All the developments try to add to BLueSky the functionalities associated with the D2-C2 separation method explained in USEPE ConOps.

This document is intended to serve as a user manual for the simulation tool.

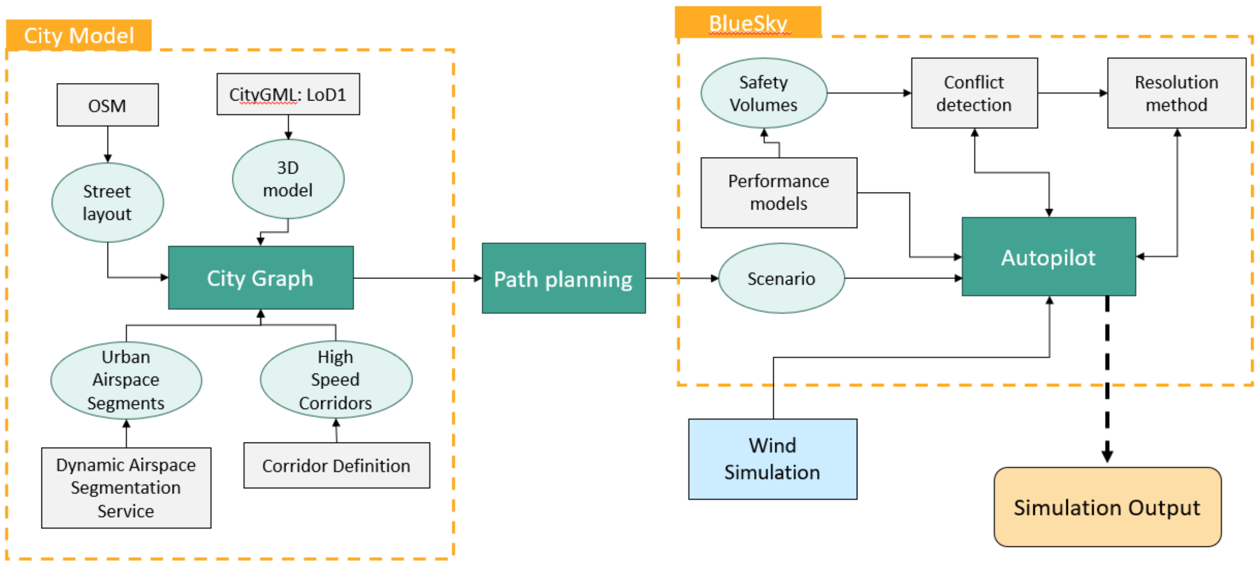
2. Detailed description of the solution

BlueSky is an open-source and open-data air traffic simulation tool. Even though it was originally created for manned aircraft and as a tool to perform research on Air Traffic Management, additions have been made for urban UAV simulation. It is developed using a high-level programming language with a simple structure, to facilitate expandability of the code. In this way, new functionalities can be easily added to BlueSky by plugins.

Even though BlueSky has many useful functionalities for USEPE objectives, there are some aspects that are not covered by this tool. Therefore, some new functionalities are developed.

The simulator here presented is divided in three modules:

- City Model. First, we have to create a model to represent the city. This is done by means of a graph, which is created using the street layout and the building information obtained from OSM and CityGML LoD1 model respectively. The graph is created according to the D2-C2 method, so it considers high speed corridors and a dynamic airspace segmentation.
- Path planning. Once we have the graph, optimal trajectories are calculated. These trajectories have to be transformed into BlueSky commands which are understood by the simulator.
- BlueSky. This tool is used to simulate air traffic. Some additions have to be made to implement all the functionalities considered by the D2-C2 method: i) performance models of the drones we are interested in; ii) a conflict detection method based on the safety volumes and the urban airspace segments; and iii) resolution method. In addition, BlueSky receives as input the wind data. The simulation results will be stored in a log file.



2.1. City model and path planning

Regarding the code, the developments related with the city model and the path planning are included in the folder "nommon". Using the script "main.py" we can generate the scenarios to be used as input for BlueSky.

2.1.1. Input

From the point of view of the algorithm, the only input passed to the component will be a configuration file "settings.cfg". It is divided into eight sections: "City", "Layers", "Corridors", "Segments", "BuildingData", "Options", "Outputs" and "Strategic_Deconfliction". The "City" section contains information about the city or zone where drones will fly:

- **zone_mode:** string indicating the type of zone considered. Possible values: "square" or "rectangle". If "square" the module will use the parameters "hannover_lon", "hannover_lat" and "zone_size", while if its value is "rectangle", "hannover_lat_min", "hannover_lat_max", "hannover_lon_min" and "hannover_lon_max" parameters are required.
- **hannover_lat:** float representing the latitude of the square fly zone center.
- **hannover_lon:** float representing the longitude of the square fly zone center.
- **zone_size:** integer representing the size of the square centered on the point given by "hannover_lat" and "hannover_lon". It is expressed in meters.
- **hannover_lat_min:** float representing the minimum latitude of the fly zone.

- **hannover_lat_max:** float representing the maximum latitude of the fly zone.
- **hannover_lon_min:** float representing the minimum longitude of the fly zone.
- **hannover_lon_max:** float representing the maximum longitude of the fly zone.
- **import:** boolean variable indicating if the graph is imported or not. “True” if the module is configured to import a previously created graph; “False” if the graph is created.
- **imported_graph_path:** string indicating the path to the graph.

On the other hand, the section “Layers” contains information related with the layers:

- **number_of_layers:** integer indicating the number of layers into which the urban airspace will be divided.
- **layer_widht:** integer indicating the distance between layers expressed in meters.

The section “Corridors” contains the following parameters:

- **corridors:** string of numbers indicating the active corridors.
- **altitude:** integer indicating the altitude of the corridors expressed in meters.
- **delta_z:** integer representing the altitude gap between a corridor and the one above it with the opposite direction. It is expressed in meters.
- **speed:** float representing the velocity of the corridors.
- **acceleration_length:** integer representing the length of the acceleration length expressed in meters.
- **file_path_corridors:** string indicating the path to the csv file containing the points of the corridors.

Moreover, the section “Segments” includes options abouts the segmentation:

- **import:** boolean variable indicating if the segmentation is imported or not. “True” if the module is configured to import a segmentation; “False”, otherwise.
- **path:** string indicating the path to the npy file containing the segment information.

In addition, the information about the buildings is indicated in the section “BuildingData”:

- **lat_min:** float representing the minimum latitude of the region where building altitude is considered.

- **lat_max:** float representing the maximum latitude of the region where building altitude is considered.
- **lon_min:** float representing the minimum longitude of the region where building altitude is considered.
- **lon_max:** float representing the maximum longitude of the region where building altitude is considered.
- **divisions:** integer representing the number of horizontal and vertical airspace divisions. Each sector has associated the altitude of the tallest building.
- **directory_hannover:** string indicating the path to the directory where the GML data is stored.

The section "Options" contains some general options when creating the graph:

- **one_way**: boolean variable indicating weather in each layer drones can fly only in one direction. If "True" drones only fly in one direction; "False", otherwise.
- **simplify**: boolean variable indicating if the graph has to be simplified. If "True" the number of nodes is reduced; "False", otherwise.
- **simplification_distance**: float representing the distance below which two nodes will be merged into one. It applies if "simplify=True".

The section "Outputs" contain information about the outputs of the tool:

- **graph_path:** string indicating the path where the graph should be stored.

Finally, the section "Strategic_Deconfliction" includes options related to the strategic conflict resolution:

- **ratio**: maximum ratio between the travel time of the trajectory without exceeding the segment capacity and the optimal trajectory.
- **delay**: number of seconds a flight is delayed if we cannot find a trajectory without exceeding either the segment capacity or the above mentioned ratio.

An example of the configuration file is shown below:

```
[City]
mode = rectangle
hannover_lat = 52.376
hannover_lon = 9.76
zone_size = 1000
hannover_lat_min = 52.35
hannover_lat_max = 52.4
hannover_lon_min = 9.72
hannover_lon_max = 9.78
```

```
import = True
imported_graph_path =
C:\workspace3\bluesky\nommon\city_model\data\hannover_segments.graphml

[Layers]
number_of_layers = 9
layer_width = 25

[Corridors]
corridors = 1 2 3 4 5
altitude = 250
delta_z = 25
speed = 100
acceleration_length = 50
file_path_corridors =
C:/workspace3/bluesky/nommon/city_model/data/corridor_coordinates_nommon.csv

[BuildingData]
lat_min = 52.35
lat_max = 52.4
lon_min = 9.72
lon_max = 9.78
divisions = 8
directory_hannover =
C:\Users\jbueno\Desktop\Stadtmodell_Hannover_CityGML_LoD1\LoD1_Graph

[Options]
one_way = False
simplify = False
simplification_distance = 12

[Outputs]
graph_path = ./data/hannover.graphml

[Strategic_Deconfliction]
ratio = 3
delay = 60
```

2.1.2. Algorithm

The code is divided in some modules:

- **city_graph.py**. Script to create a graph that represents the city. It imports the street information from OSM and the building information from GML files. The output is a 3D graph (MultiDiGrpah3D class) that can be used to compute drone trajectories between and above buildings.
- **multi_di_graph_3D.py**. Subclass of MultiDiGraph. It is created to code the particularities of the graph used in the USEPE project. Some class methods are implemented that allow an easy implementation of some graph attributes.

- **building_height.py**. A script to import and process the building heights from gml files. It is used by "city_graph.py".
- **city_structure.py**. A module to define the sectors into which the city is divided to consider the building height. It is used by "city_graph.py".
- **dynamic_segments.py.** A module to update the segment information in the graph.
- **path_planning.py.** A module to compute an optimal route from origin to destination.
- **scenario_definition.py.** A module to compute scenarios which can be loaded by BlueSky.
- **corridors_implementation.py**. A module to create the desired corridors defined in a .csv or .geojson file. The parameters that define the corridors can be tuned in the configuration file.
- **no_fly_Zones.py**. A module to create no-fly zones by assigning zero velocity to the segments intersecting the restricted region.
- **strategic_deconfliction.py.** A module responsible for strategic conflict resolution. The user can generate flight plans without exceeding segments' capacity. These flight plans are transformed to BlueSky scenarios.
- **utils.py.** Additional functions which can be useful; e.g. "read_my_graphml" to read a previously computed graph.

In addition, there is a main script which can be used to control all the previous modules. It is called "main.py". This script can be adapted based on the execution needs. However, the general steps that are contained in this master script are explained below.

1. Configuration file

The first step is to read the configuration file. We have to modify the variable **config_path** depending on where the configuration file is located.

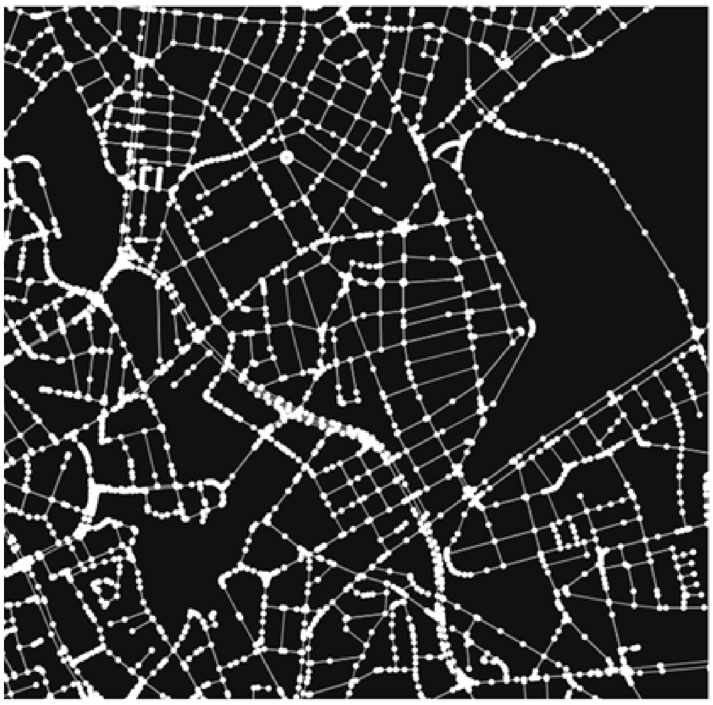
2. City graph

The second step is to obtain the graph that represents the city. This step depends on the parameters specified in the sections "City" and "Layer" of the configuration file, and there are two possibilities: i) create a new graph; ii) import a previously computed graph.

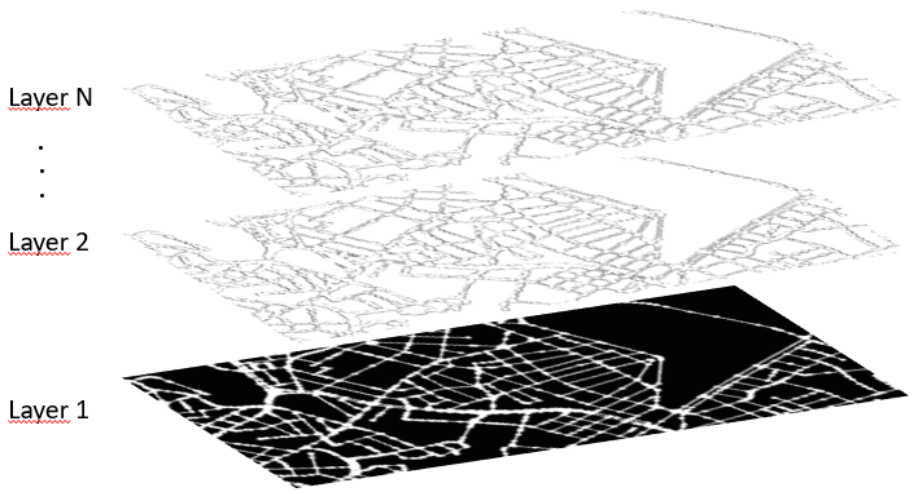
i) New graph

A new graph is created if the parameter "import" in the configuration file is "False". If so, the code calls the module "city_graph.py" and the graph is obtained following the next steps:

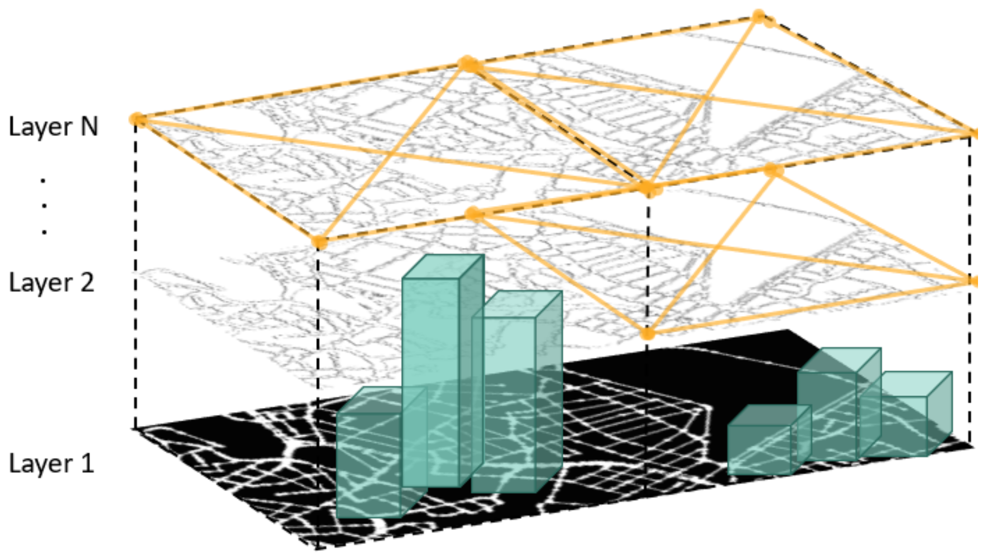
1. Obtain the graph from OSM. For that, the python library "osmnx" is used.



2. We transform the 2D graph into a 3D graph (MultiDiGraph3D class). This process consists in repeating the 2D graph at different altitudes and connecting each node with the one below and the one above, so each layer represents a flight level. This allows us to compute trajectories between buildings at different altitudes.



3. Modify the graph to also consider the movement above buildings. For that, the city is divided in some sectors. Then, we create edges connecting the corners of the sector in all the layers above the tallest building in the sector.



4. Save the graph in the path specified by the parameter "graph_path" in the section "Outputs" of the configuration file.

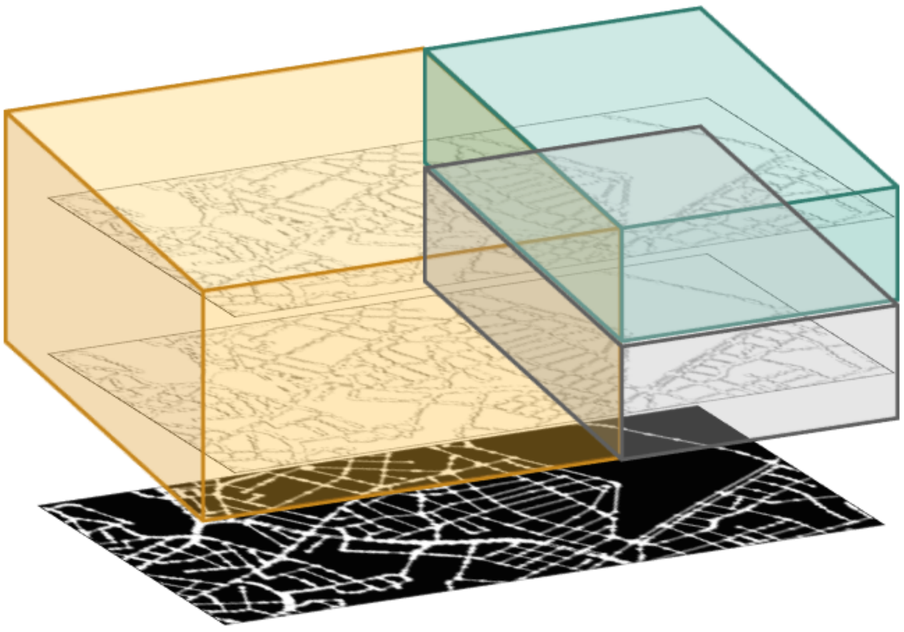
ii) Imported graph

If we have previously created a graph, we can import it by defining two parameters in the section "City" of the configuration file:

- Import = True
- imported_graph_path = {path}

3. Segments

Airspace is divided in rectangular segments. Each segment has its own attributes: velocity limit, capacity, performance requirements, safety volumes, etc., and they are common for every drone flight. Therefore, each graph node or edge has a segment associated. Changing the segment parameters, the airspace configuration is modified.



The user can dynamically update the airspace by using the function “dynamicSegments” in the module “dynamic_segments.py”. The code is divided in three parts:

1. Assign segments to edges. A parameter indicating the segment is given to each edge. An edge belongs to a segment if its origin node is inside the segment.
2. Update velocity. The speed associated with the edges is modified according to the segmentation.
3. Add travel time. Based on the speed and length of the edges, a travel time is computed for all the edges.

There are two possibilities depending on the configuration:

1. Import the segment information.
2. If there is no segment information, the function will create a default segmentation.

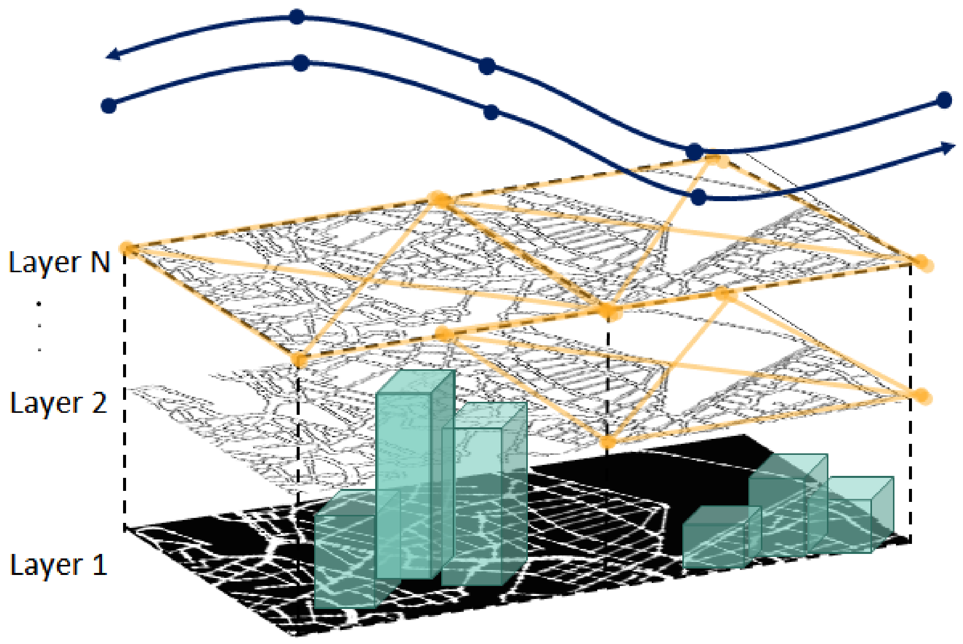
Regardless of this, the segment information is structured as a dictionary. The keys are the name of the segments and the associated values are dictionaries with the following parameters:

- lon_min (float): minimum longitude of the segment.
- lon_max (float): maximum longitude of the segment.
- lat_min (float): minimum latitude of the segment.
- lat_max (float): maximum latitude of the segment.
- z_min (float): minimum altitude of the segment.
- z_max (float): maximum altitude of the segment.
- speed (float): maximum speed of the segment.
- capacity (integer): capacity of the segment in terms of number of drones.
- new (boolean): If "True" the segment is new; "False" otherwise.
- updated (boolean): If "True" some segment property is modified (typically the speed will be updated); "False" otherwise.

The last two parameters are useful for the code to reduce the computational time. When assigning segments to edges, the algorithm will focus only on the segments categorised as "new". On the other hand, when updating the velocities of the edges, the algorithm will consider only the nodes belonging to a segment categorised as "updated".

4. Corridors

The corridors are placed above the layers created by the city graph. The projection of the corridors on the ground is defined in a .csv or .geojson file, whose path is stored in the configuration file. The altitude of the corridors is also defined in the configuration file. Traffic must be allowed in both directions. To avoid possible collisions, additional corridors in the opposite direction are defined above the previous ones, at a distance 'delta_z' defined in the configuration file.



The user can select the corridors to be included in the simulation with the variable 'active_corridors' in the configuration file. The module corridors_implementation.py first reads from the configuration file which are the corridors to be used, defined with a numerical id.

Then, corridors are created one by one. For each corridor:

1. Gets its coordinates from the data file (.csv or .geojson). If the corridor id defined does not exist, the script raises a message and skips to the next corridor defined in the configuration file
2. The corridor in one direction is created. This process includes:
    a. To create the nodes defining the corridor and the edges that join each node. The edges have associated the velocity (km/h) defined in the configuration file. In case that the first and last nodes are the same, the corridor is defined as a closed path.
    b. To define the entry paths to the corridor. Additional nodes and edges are created to ascend from the city graph level to the height of the corridor and

accelerate up to the corresponding velocity of the corridor. The length (in meters) of the acceleration lane is a parameter that can be modified in the configuration file.

3. The corridor in the opposite direction is created following the same process.

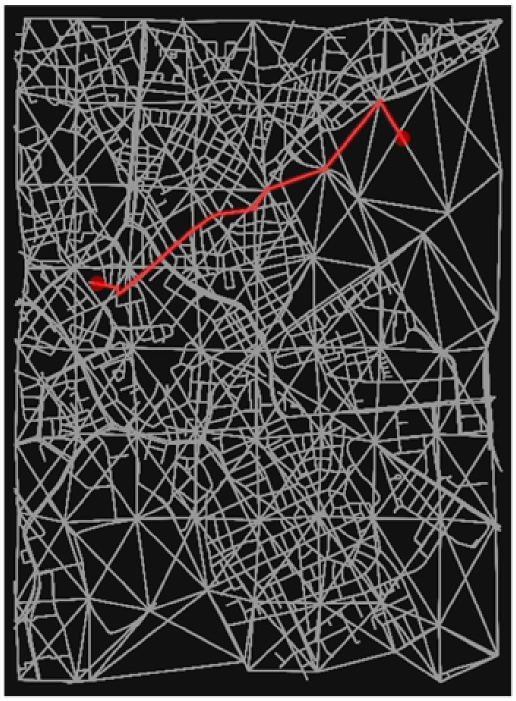
5. No-fly zones

It is possible to create restricted no-fly zones within the city. The restricted area is defined by a polygon and it will affect every segment that intersects with the restricted area at any height.

The module no_fly_zones.py analyses each segment and in case it intersects with the restricted area, its fly speed is updated to 0.

6. Path planning

The user can compute an optimal trajectory between two points. The default objective function is to minimize the travel time, but the user can also minimize the length. By using the function "trajectoryCalculation" in the module "path_planning.py" the user can calculate the optimal trajectory.



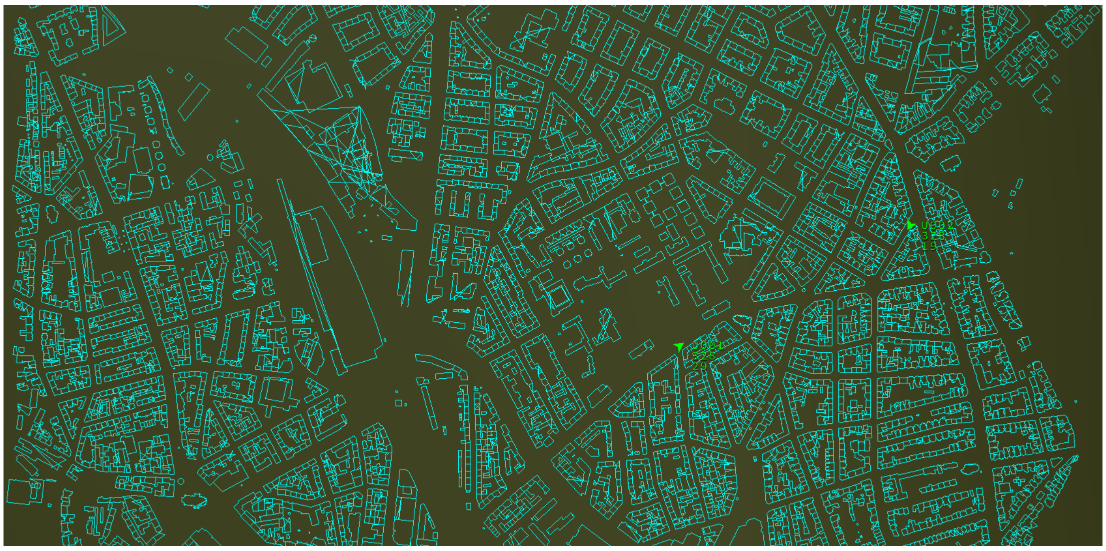
7. Scenario definition

Using the module "scenario_definition.py", the user can generate BlueSky scenarios.

The fundamental function is "createInstructionVX", which transforms two consecutive waypoints in the commands needed by BlueSky to follow the trajectory. This function determines when a drone has to decelerate, climb, turn, etc.

In addition, there are three high level functions:

1. "createFlightPlan". This function creates a flight plan for a drone. It successively calls "createInstructionVX" and writes all the commands in a text file.
2. "automaticFlightPlan". It generates many flight plans. Origins and destinations of a defined number of trajectories are randomly generated. We create a BlueSky scenario for each drone and a general BlueSky scenario which calls all the drone flight plans. Loading this general path in BlueSKy, we can simulate all these drones flying at the same time.
3. "drawBuildings". It generates the scenarios needed for printing the buildings in BlueSky. First, it loads the building data. Then creates several BlueSky scenarios. Each scenario prints the footprints of 10000 buildings. The reason for grouping them in batches of 10000 is BlueSky's memory. We can have issues with giving many commands in a row, so they are divided in batches of 10000. Once we have loaded the buildings of one scenario, we can continue with the next one. For loading a second scenario we can use the BlueSky command "PCALL". After importing the buildings, some flight plans can be loaded.



8. Strategic deconfliction

This module enables strategic conflict resolution. As explained before, the airspace is divided into segments, each of them with a maximum capacity associated (in terms of number of drones contained in the segment). The system computes how each new user populates the segments (based on their flight plan). In accordance with the principle of 'First Planned - First Served', if a new flight plan results in an overpopulated segment, the requested operation will be rescheduled or rerouted.

The module is composed of some basic functions:

- "initialPopulation". Create an initial data structure with the information of how the segments are populated. The information is stored as a dictionary, where the keys are the segment names. The information is stored as a list for each segment: dict[j] = [x(t1), x(t2), x(t3),..., x(tn)], where x(t) represents the number of drones in the segment j during the second t. Therefore, strategic deconfliction takes place at the level of seconds. Using this function we initialise all values to zeros.
- "droneAirspaceUsage". Computes how the new user populates the segments. It returns the information of how the segments are populated including the tentative flight plan of the new drone.
- "checkOverpopulatedSegment". Check if any segment is overpopulated.
- "deconflictedPathPlanning". Computes an optimal flight plan without exceeding the segment capacity limit. The procedure consist in:
    1. Compute the optimal path from origin to destination without considering the segment capacity limits.
    2. While the new drone exceeds the segment capacity limit:

        2.1. A sub-optimal trajectory is computed without using the overpopulated segment.

        2.2. If the travel time of the sub-optimal trajectory divided by the optimal travel time is higher than a configurable threshold, e.g., 3 times optimal travel time:

            2.2.1. The flight is delayed by a configurable value, e.g., 60 seconds.

            2.2.2. Repeat step 2 with the new departure time.

    3. It returns the flight plan, the departure time and the new information about how the segments are populated.
- "deconflcitedScenario". The user can generate a BlueSky scenario of a strategic deconflicted trajectory from origin to destination.

2.1.3. Output and relation with other components

This component allows the user to generate some text files with the extension '.scn' that can be loaded by BlueSky.

A scenario can be loaded by using the command 'IC' followed by the scenario path. However, if we want to load a second scenario without deleting the first one, we have to use the command 'PCALL' followed by the scenario path.

2.2. BlueSky

2.2.1. Performance model

OpenAP is one of the performance models available in Bluesky. The OpenAP performance model allows for using different types of aircraft by defining the following parameters:

- "name": "aircraft full name",
- "n_engines": "number of engines",
- "engine_type": "engine typer: TF, TP",
- "mtow": "max take-of weight (kg)",
- "oew": "operating empty weight (kg)",
- "mfc": "max fuel capacity (L)",
- "engines": "[(engine: power (kW))]",
- "envelop": {
    - "v_min": "minimum speed (m/s)",
    - "v_max": "maximum speed (m/s)",
    - "vs_min": "minimum vertical speed (m/s)",
    - "vs_max": "maximum vertical speed (m/s)",
    - "h_max": "maximum altitude (m)",
    - "d_range_max": "maximum flight range (km)" }

Some types of rotor aircraft are defined in BlueSky:

1. EC35: Eurocopter EC135
2. M600: DJI Matrice 600
3. Amzn: Amazon octocopter
4. Mnet: Matternet quadcopter
5. Phan4: DJI Phantom 4
6. M100: DJI Matrice 100
7. M200: DJI Matrice 200
8. Mavic: DJI Mavic pro
9. Horsefly: Horsefly Gen 5.3

For the implementation of one of the USEPE validation exercises, an additional drone is defined:

```
"W178"
: {
            "name": "Wingcopter 178 Heavy Lift",
            "n engines": 4,
            "engine type": "TS",
            "mtow": 18,
            "oew": 12,
            "mfc": 0,
                 "engines":  [["SII-4035-560KV",  3.50],  ["SII-4035-560KV",  3.50],
         ["SII-4035-560KV", 3.50], ["SII-4035-560KV", 3.50]],
            "envelop": {
              "v min": -14,
              "v max": 41,
              "vs min": -5,
              "vs max": 6,
              "h max": 5000,
              "d range max": 120
            }
```

2.2.2. Conflict detection

First, some concepts are clarified:

- A Near Mid-Air Collision (NMAC) volume around the ownship is defined. This can be identical to the physical extent of the aircraft or it can be a little wider.
- A loss of separation takes place when the NMAC of two drones intersect.
- Two aircraft are in conflict if you can expect them to enter each other's NMAC within a defined time.

For the USEPE project, an NMAC is always considered as a cylinder (e.g., at a distance <3.75 m horizontal and <10 m vertical) as it only models navigational inaccuracies to provide a safety margin against collisions. A different NMAC size can be defined for each drone.

We can activate or deactivate the conflict detection by using the following command:

- ASAS ON/OFF

and we can select the conflict detection method with:

- CDMETHOD {method name}

At this moment the only method available is "StateBased". This method calculates if within a look ahead time the NMAC of two drones will intersect based on the current state. The size of the NMAC volumes can be modified by the following commands:

- ZONER radius acid
- ZONEDH height acid

where ZONER command set the radius of the NMAC of the drone called "acid" (aircraft id) as the "radius" value in nautical miles, and the ZONEDH command set the vertical distance of the NMAC of the drone called "acid" as the "height" value in feet.

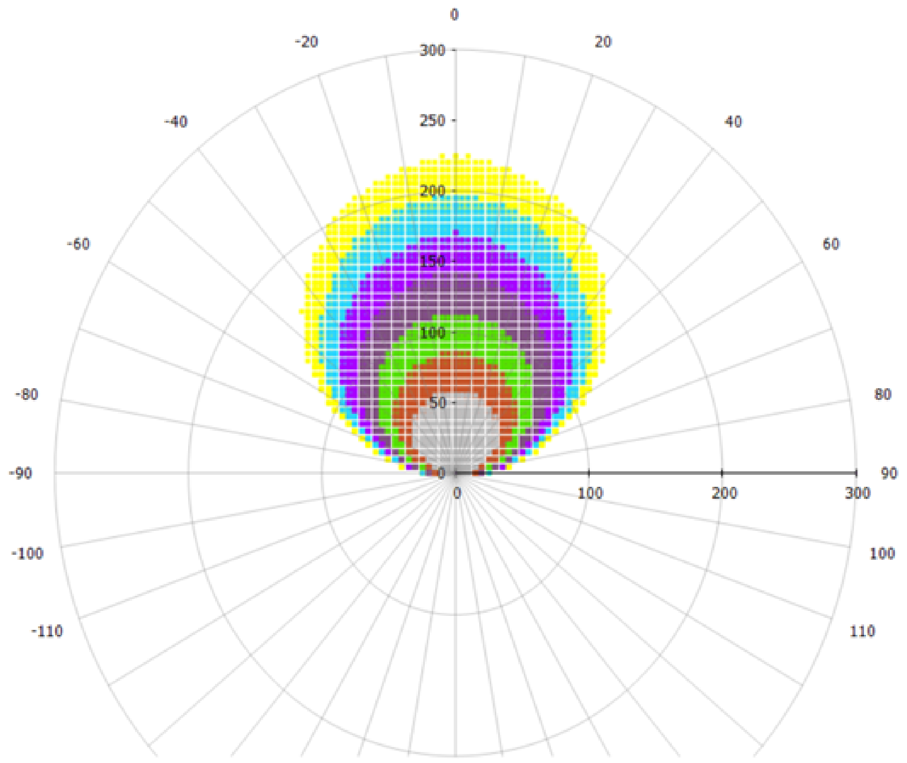
However, we are working on an improved conflict detection method which combines this "StateBased" method with the safety volumes.

2.2.2.1. Safety volumes

To further improve the method and make it performance based we include the capabilities of the ownship: If we know about the principal geometry of the ownship avoid trajectories, we can pre-simulate the avoidance maneuver for each possible encounter. For example, using the following assumptions:

- An NMAC at a distance <3.75 m horizontal or <10 m vertical
- Both, ownship's and intruder's absolute speed is 13.88 m/s (50 km/h)
- The intruder can avoid with a bank angle of max. 30°
- Avoidance is done horizontally only without changing speed
- Avoidance may be done to any side (left or right) up to a relative course of +-120°

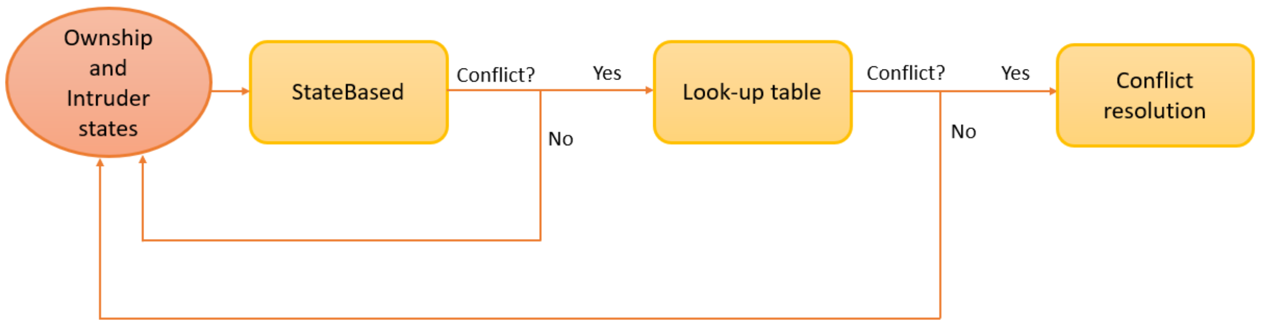
we compute a look-up table that can be represented as a safety volume indicating the time-to-react levels accumulated over all intruder courses. This is represented in the following image, where yellow means seven seconds to react while gray means 1 seconds to react.



Using these look-up tables we can detect the conflicts with more accuracy. However, since we have to compute these tables for each pair of drones, velocities and resolution maneuvers, this method will be very CPU intensive. Therefore, we use the following approach:

1. BlueSky will perform a preliminary check using the "StateBased" method.
2. Once possible conflicts have been detected, they will be checked again with the look-up tables.
3. Compute avoidance maneuvers.

This is represented in the following scheme:



2.2.3. Conflict resolution

Once a conflict is detected, the next step is to compute the resolution maneuver. We can active/deactive the conflict resolution using the following command:

- RESO {resolution method}/OFF

We can create our own resolution method in a plugin, but in this first of the simulator we will use the MVP resolution method defined in the source code of BlueSky. Therefore, to activate this resolution method we should introduce:

- RESO MVP

This method can avoid conflicts by controlling horizontal speed, heading or vertical speed. This can be selected by using two commands: RMETHH and RMETHV. The first command allows the user to activate the horizontal resolution which can be configured to control speed, heading or both:

- RMETHH BOTH/SPD/HDG/OFF

On the other hand, the RMETHV command allows the user to activate the vertical resolution which avoids conflicts by changing the vertical speed:

- RMETHV VS/OFF

Both horizontal and vertical resolutions cannot be active at the same time. Also, two different resolution methods cannot be selected for two different drones. Since we are simulating drones in an urban environment, the user should be careful with the resolution method used (e.g., if a drone is flying between buildings, a heading change may generate a crash with a building). We recommend using the horizontal speed resolution which can be used above and between buildings. In next versions of the simulator we will include the option of using a resolution method adapted to each part of the city: between buildings, above buildings or corridors.

2.3. External tools

2.3.1. Wind pre-processing

PALM generates turbulent wind field data (consisting of the three Cartesian velocity components) with temporal and spatial resolution as required for the validation exercises, that will be used as input for the BlueSky simulator. Data is in netCDF format.

A python module has been developed to perform a pre-processing of the data. This module is called **"wind_preprocess.py",** which consists of several functions. Using the high level function "main()", the user can generate a BlueSky scenario with the necessary commands to import the wind data into the BlueSky simulation.

PALM simulation generates the data with a high spatial resolution. Using such a resolution in the simulator implies an enormous computational cost. Therefore, this module allows us to interpolate the data in a grid with configurable grid spacing.

It is important to note that BlueSky has limitations in relation to how the wind affects the drone performance. The actual BlueSky version only allows the wind data to be imported as a direction in the horizontal plane (0° - 360°) and a speed at different points in the space (latitude, longitude, altitude). Consequently, the vertical component is not considered in the drone simulations.

3. Limitations and future improvements

- Corridor entry acceleration lanes are defined with an angle of 30 degrees with respect to the line defined by the entry point and the next point of the corridor. In some cases, this leads to a small angle of the entry lane with respect to the corridor before the entry point. We will take into account not only the next point but also the previous one to define the path of the entry lane.
- Corridor exit deceleration lanes are to be defined. So far, entries are also exits.
- Define the procedure when a no-fly zone is created for drones that
    - depart/arrive at a spot within the restricted zone
    - are flying in the region at the moment of creation of the restriction (terminate flight, hover, continue).
- An initial version of the Dynamic Airspace Segmentation Service has been developed. It needs to be integrated into the simulator
- Conflict detection and resolution methods based on the D2-C2 method need to be developed as BlueSky plugins. They will be included in the next version.
- The strategic conflict resolution estimates where the drone will be at each timestamp using the graph velocities. We do not take into account speed limitations associated with turns and altitude changes.

- BlueSky does not consider the effect of the vertical component of the wind in the drone performance.