Getting Started with the ELSA Air Traffic Simulator

Prerequisites

In the following, we will assume that that the user has:

- basic knowledge of Python for the strategic layer,
- no knowledge of C OR no knowledge of Python for the tactical layer, because one can use either the executable compiled from C or the Python interface.
- some basic knowledge of UNIX commands.

1 Setup

To install the code, first create a directory where you see fit to host the code. Then download the code either using git (preferred method) or directly downloading the zip from `https://github.com/ELSA-project/ELSA-ABM`.

1.1 Using git

git is already installed in several flavours of Ubuntu, Linux Mint, and Fedora. If git is not installed on your system, you can install it for instance using:

```
sudo apt-get install git
```

on a Debian-like system like Ubuntu[1]. Then run:

```
git clone https://github.com/ELSA-project/ELSA-ABM.git
```

in the folder you have created to download the code. git is especially handy if you want to keep the repository updated with new releases and if you want to contribute to the code.

1.2 Downloading directly

You can also just download the source code by clicking on "download zip" on the main page at `https://github.com/ELSA-project/ELSA-ABM`, then extract it wherever you want.

[1]In the following, we will assume that that user has this kind of OS. The code has not been tested with Windows or MacOS, although. We are pretty sure that the code will be broken for the former (because of paths issues), but it may run on the latter. The final release should be compatible with all OS.

1.3 Installing dependencies

The code is mainly written in Python 2 so you should install it before anything else, using:

```
sudo apt-get install python
```

Then you should make sure that all the dependencies required are installed. The following python modules should be installed:

- scipy
- shapely
- networkx
- basemap
- MySQLdb
- matplotlib
- descartes
- scikit-learn

On Debian-like system like Ubuntu, you can use the following command to install them all:

```
sudo apt-get install python-scipy python-numpy python-shapely
python-networkx python-mpltoolkits.basemap python-matplotlib
python-sklearn python-mysqldb
```

The package ‘descartes’ has to be installed separately using for instance pip:

```
sudo pip install descartes
```

If pip is not installed on your system, you can install it like this:

```
sudo apt-get install python-pip
```

You will also need the swig package to do the interface between C and Python, which can be installed this way:

```
sudo apt-get install swig
```

1.4 Make the setup

Once the source code has been downloaded and dependencies are satisfied, you can run the setup.py in the main folder in command line:

```
./setup.py
```

It will prompt you for path of the directory you want for the results generated by the code. It then compiles the C code, makes the python wrapper for the tactical level and generates the html documentation describing the Python code. Hopefully, the script should not yield any error. If it does, it will most likely be due to some broken dependencies. In any case, you can open a ticket on GitHub to ask for help.

After that you can run some tests with:

```
tests/run_all_tests.py
```

All tests should be positive. If it is not the case, you can once again open a ticket on GitHub.

1.5 For MAC OS

The installation has only been tested marginally on MAC OS. Everything should be easy for the tactical layer, because there are no dependencies. For the strategic, most of the modules are easily installable, using a package manager. The only exception is basemap, which is used by the code to draw some maps. The issue seems to be in the installation of a dependency of basemap, GEOS, but solutions exist online. The alternative is to comment all the import of basemap in the code, which are only used for post-processing. The rest of the setup should be the same.

2 Organization of the code

The code is organized as follows:

- the `abm_strategic` folder contains an ABM simulating the strategic phase, including a network builder, airlines and flights objects, and a network manager allocating the airspace to the airlines.
- the `abm_tactical` folder contains a tactical model including a ”mega-controller” with different rules of conflict resolution as well as a ”shock generator”.
- the `interface` folder contains very basic functions which allows to easily control the tactical model from Python.
- the `libs` folder gathers several hand-made libraries.
- the `scripts` folder contains some handy scripts which can be used as examples to produce results with the model
- the `tests` folder contains several scripts for unitary and manual tests.

- the `doc` folder gathers the documentation, in the form of html files describing the python code and pdf files for small tutorials.

The Strategic model and the Tactical one are fairly independent. One can use them for his/her own purpose by feeding them with the right input. They can also be used conjointly, even if this feature is quite basic at the moment.

3 Using the network generator

Included in the strategic agent-based model is a tool to build airspaces. The relevant function is called `prepare_hybrid_network` which is present in the file `prepare_navpoint_network.p`
In the following, we call an "hybrid network" a Net object (the class is defined in the file simAirSpaceO.py) which has an attribute which is a NavpointNet object (also defined in simAirSpaceO.py). The Net object, let us call it `G`, is the support for all the information concerning the sectors, whereas the NavpointNet object – `G_nav` – includes all information related to the navigation point network. The latter can be accessed using `G.G_nav`.

The `prepare_hybrid_network` function is a constructor which returns such an hybrid network. It builds it using some parameters gathered in the dictionary `paras_G`. This dictionary can be built using a file like the template `paras_G.py`. Some explanations are available in the template file concerning the possible options. In particular, one can:

- Give a networkx Graph which will be taken as input for the construction of the sector network. All data from the original graph should be conserved. The nodes should have a key 'coord' which give their coordinates (centroids of the sectors). If no graph is given in input, the builder can build a artificial network based on a Delaunay triangulation of some random points distributed in the $[0, 1] \times [0, 1]$.
- Give the boundaries of every sectors in the form of shapely Polygons.
- Give the capacities of the sectors with a dictionary or give some traffic data to infer the capacities (maximum number of flights in the area in the time-window).
- Give the time of crossing between navpoints, or provide some data to infer it.
- Give the list of entry/exit in the area. At the moment, the user cannot use the traffic data to infer it, but this feature will be added in the future. If the user does not provide the entry/exits, it can be picked at random.
- Give a networkx Graph which will be the base of the NavpointNet object. All data are conserved, unless stated otherwise (if the user decides to generate some new times of crossing for instance). The times of crossing should be stored as keys 'weight' on each edge if the user wants to use them. The positions of the navpoints should be should be stored with the key 'coord' on each node.

The user can directly use prepare_navpoint_network.py as a script to generate a network by changing the name of the file from which the parameter dictionary is built:

```
from paras_G import paras_G
```

and then run it:

```
./prepare_navpoint_network.py
```

As an example, the user can directly use the template file paras_G.py to generate an artificial network. Running directly prepare_navpoint_network.py without modification is enough for this.

The output consists in three files:

- The network itself in a pickled file[2], called for instance Example.pic,
- an image of the network, called for instance Example.png
- some very basic statistics on the network (this file could be expanded in the future), called for instance Example_basic_stats_net.txt

One can copy the template file and modify it to generate different kinds of network. The inputs needed are all described in the template file.

4 Using the strategic ABM

The strategic layer aims at producing sensible trajectories planned by the air companies in different environments with different behaviors from the airlines. The output is essentially the "M1 files", which are the equivalent of the DDR data about planned trajectories.

The main functions concerning the ABM strategic can be loaded via[3]:

```
from abm_strategic import *
```

The main functions are:

- `do_standard`,
- `generate_traffic`,
- `iter_sim`,

[2]pickle is a python module able to serialize any python object.

[3]Note that this is not likely to override any other loaded functions given the specific names of the functions. The list of the functions loaded can be viewed in the __init__ function in abm_strategic/

The `prepare_hybrid_network` is also loaded, as well as other functions concerning the rectification of the trajectories (see section 5).

The first one is a low level function which is essentially doing one simulation of the strategic ABM. The third one is able to sweep some parameters of the ABM, averaging the results of several instances with the same values of parameters. These two functions are made for systematic studies of the strategic ABM and are not suited to the interface with the tactical layer.

On the other hand, the generate_traffic function provides a better interface and a higher level of control of the ABM, as well as a simplified setup with standard default parameters. It is suited as a generic trajectory generator, usable in particular with the tactical layer. In this tutorial, we only describe this function which includes everything needs to do simple simulations[4].

The function needs an hybrid network to run, that you can build following section 3. It needs also a parameter file, a template of which you can find in `abm_strategic/paras.py`. The template serves also as example and is used if no parameter dictionary is given as entry. Hence one can very simply use the function like this:

```
generate_traffic(G)
```

with `G` an hybrid network. The file `tests/example/generate_traffic_example.py` can be used as a way of exploring this function. One of the most interactive features is the ability for the user to directly control some parameters in a transparent way. For instance:

```
generate_traffic(G, ACtot=1000, days=48*60)
```

overwrites the number of flights required by the parameter dictionary, setting it to 1000, and simulates two days of data.

The parameters which are not specified has keyword arguments are taken from a configuration file of which a template can be found in `abm_strategic/paras.py`. The setup makes a copy of it as `abm_strategic/my_paras.py`. This also the default path on which the function 'generate_traffic' looks if the path of the config file is not specified (with keyword argument 'paras_file'). So in summary, the user can fully control the function via a configuration file, or use a configuration just as a template and overwrite the parameters via keyword arguments in 'generate_traffic'. The last option is particularly useful and one wants to iterate on different values for a given parameter.

Another important option for the user is using the option `coordinates`. If set to True, the function returns some coordinate-based trajectories instead of some label-based ones. This is in particular important for the tactical ABM which takes as input the former.

The function returns the simulated trajectories but also some statistics about what happened during the strategic allocation so as to keep track of them.

[4]Or more complex ones, since a lot of optional capabilities are included there.

5 Rectification of the trajectories

5.1 Description

The abm_strategic folder also include some tools to study some new concepts of SESAR, among which the free-route trajectories. In the model, trajectories can be "rectified" by moving some points of the trajectories in order to "straighten" them. The relevant function is also available by calling:

```
from abm_strategic import *
```

and is named `rectificate_trajectories_network_with_time`. The inputs to this function is a list of trajectories to be modified, the navpoint network on which the trajectories are defined and an efficiency target. The efficiency E_S of a set S of flights is defined by:

$$E_S = \frac{\sum_S l_b}{\sum_S l_t}$$

where l_t are the current lengths of the trajectories and l_b are the best ones (straight lines or great circle). This efficiency is always smaller than 1 and is usually around 0.95 in reality, meaning roughly that the trajectories could be shortened by 5%.

The function returns the straightened trajectories, and can do it in two ways:

- Either by removing some nodes on the trajectories, keeping always the first and the last points. Afterwards the points are resampled on each trajectory, hence keeping the number of points per trajectory constant (but not the total number of points).

- Or by moving some points slowly towards the middle of the segment defined by the point before and the point after it.

The first procedure is much faster and should be preferred, but uniformize the distribution of points along the trajectories. It is triggered by choosing `True` with the `remove_nodes` keyword.

The function also recomputes automatically the altitudes and times of crossing of the points using a linear interpolation between the point before and the point after.

This function also features some more advanced capabilities, like the possibility of fixing different probabilities of moving different groups of points. This allows to smoothly go from the current scenario to a SESAR scenario where all (planned) trajectories are straight. One can refer to the documentation of the lower level function `rectificate_trajectories` for more details about this.

As an example, the user can use again the `tests/example/generate_traffic_example.py` file and change uncomment line 30. The file generated will have trajectories almost straight (efficiency equal to 0.99).

5.2 Caveats

The user needs to be careful with the rectification of the trajectories. Since it was designed as a hack to use the strategic model in the SESAR scenario (with free-routing), the output of the rectification function IS NOT compliant with the capacities of the sectors. In order to produce straight trajectories compliant with capacities, the users can design the network such a way that the air companies are able to use straight trajectories. For instance, by putting links directly between entries and exits, which will force the air companies to select this trajectory for their shortest paths. This has not been tested though.

5.3 Using the Tactical layer

The Tactical layer aims at transforming planned trajectories into "actual" trajectories by deconflicting them with a simulated super-controller. The standard input is an "M1 file" whereas the output is an "M3 file", similarly to what exists in DDR data.

The code of the Tactical ABM is stored in abm_tactical folder. In the abm_tactical/config folder there is the config.cfg that contains the values of most of the parameters.

The tactical layer can be used independently of the other features of the models. It can be directly controlled via command line with the binary or via a python interface, described in 5.4.

You can directly compile the source code in the following way:

```
LC_ALL=C gcc -O3 -c *.c && gcc *.o -o ElsaABM.so -lm
```

or use the `compile.py` script which is also compiling the files for the python interface. Using the `setup.py` script described at the beginning also calls `compile.py`.

Once the binary `ElsaABM.so` is produced it is possible to execute the simulations:

```
./ElsaABM.so M1_file M3_file Config_file,
```

where `M1_file` is the file containing the trajectories which will serve as input for the model and `M3_file` is the file that will contain the output, i.e. the trajectories deconflicted by the controller.

Using the files provided as examples in the tests/example/ folder, the user can run:

```
./ElsaABM.so ../tests/example/M1_example.dat ../tests/example/M3_example.dat
 ../tests/example/config/config.cfg
```

which should produce the `M3_example_0.dat` file with deconflicted trajectories.

5.3.1 Input-Output File Format

Either M1 input file and M3 output files have the same format. The first line must contain the number of flights (trajectories contained in the files):

```
1475\tNflight
```

the following lines are the trajectory of each aircraft. In particular the first two positions, separated by a \\t, represent respectively the ID, a unique int number that identify the flight, and the number of NVP in the related route.

The following positions are the nvps of the route. They consist in 5 values, separated by a comma: the horizontal position (latitude and longitude), the Flight Level, the time in the format '2010-05-06 10:20:32', and an integer number that refers to the sector to which the navpoint belongs.

5.3.2 Configuration File

Most of the parameters are stored in the configuration file. Changing these values do not require to re-compile the code.

An example of a config file can be found in `abm_tactical/config/`. Each line starting with # is a comment, whereas all other lines are values. The name of the variable it refers to needs to be written after a '`\t#`'. For instance:

```
#This is a comment
24\t#t_w\n
```

gives $t_w = 24$. The position of the values in the configuration file does not matter.

Note that the ABM does not perform any consistency checks on the variables. For example if one wants to increase the lookahead t_w, one needs to fix the product $t_w \times t_r \times t_i$ which represents the time-step[5]. These three values are the most crucial ones for the simulations and should be chosen with care. The user can find their descriptions in the deliverable D2.4 included in the doc/ folder.

Most of the values of the configuration file can be changed without too much trouble. The most simple features are tunable with the parameters `nsim` (number of simulations), and `Nm_shock`, (number of shocks), `radius` (radius of the shocks in nautical miles). Using and modifying other parameters requires some more deeper understanding of the models, which can be achieving by reading D2.4, included in the doc/ folder. See also the caveats section 5.5.

5.3.3 Additional files

Other files are required to run the simulations, which are called respectively `temp_nvp.dat`, `shock_tmp.dat`, `sector_capacities.dat`, and `bound_latlon.dat`. Some examples

[5]The smaller is this product, the better works the ABM by the way. See deliverable D2.4 included in the doc/ folder, however a small time-step implies a high computational effort. A good choice for the time-step is about 3min.

are stored in the config folder `abm_tactical/config/`. Their paths have to be specified in the config file, in the corresponding entries.

The `temp_nvp.dat` consists in a two columns text file with latitude and longitude of the temporary points used for rerouting. The number of temporary nvps used in the simulation is defined via a #define variable called `NTMP` in mSector.h. A python utility is included in `abm_tactical/ generate_temporary_points.py`, called compute_temporary_points, to generate new files with temporary points. This function takes as input the boundaries of the airspace under control (given as a list of coordinates latitude/longitude), the number of points to generate and where the file should be saved. The user must be careful because the code:

- does not check if the points are within the ACC.
- does not check the number of points matches `NTMP`.

The `shock_tmp.dat` has the same format of `temp_nvp.dat` and it contains the centers of the shocks used in the simulations. This also has to be created via an external script, which is not included at the moment in the repository.

The `sector_capacities.dat` has two columns: the first one stores the ID of the sector, the second the related capacity. These labels need to be consistent with the ones present in the M1 file. Note that the user can use the '-1' for navpoints which do not have to be under control. This is more specifically used for the first and the last points of the trajectories (see caveats 5.5).

Finally, the file `bound_latlon.dat` contains the boundaries of the airspace as a list of latitudes/longitudes, like the `temp_nvp.dat` file.

5.3.4 Define

Other parameters are defined in the header files. Every time you want to change these parameters you need to recompile the code. For example in mABM.h is defined LS that is the minimum improvement of the trajectory for a direct:

```
#define LS 1000
```

Another useful #define is DTMP_P that is the Maximum distance in meters of the selected temporary points for the rerouting operations.

Others #define do not have associated values. For example:

```
#define SINGLE_TOUCH
```

If you comment this #define the ABM can perform a multiple modification of the route of the same aircraft in the same time-step if it does not find any solution.

5.4 Python interface to the Tactical layer

The folder `interface`, and more specifically the file `abm_interface` provides some function to call the C code from python. The most important function is called `do_ABM_tactical` and calls an interface to the code using SWIG. It arguments include the paths to all necessary files, i.e. the input file, the output file, the config file, and all the additional files `temp_nvp.dat`, `shock_tmp.dat` etc.

Another useful function is called `choose_paras` which able to modify a configuration file in a transparent way. The user just needs to indicate the path to the config file to modify, the name of the parameter to change, the new value, and the function will change the value in the file. So one possible workflow using this interface could be:

```
from value in values:
    choose_paras(value, name_of_a_parameter)
    do_ABM_tactical(input, output, config)
```

As an example, the user can run the file `tests/example/interface_example.py`, which runs the same simulation than the command in the introduction of this section.

5.5 Caveats

Here are some potential traps for new users concerning the tactical layer:

- The time step t_s is the one the most important parameter of the model. It is equal to $t_s = t_i \times t_w \times t_r$:

    1. The time increment t_i is the time resolution for the trajectories. It should be very small (around 8 seconds),
    2. The time window t_w represents the time horizon of the controller on which it will compute the future potential conflicts.
    3. The time roll t_r, which is a fraction of the time window, represents the time after which the controller updates trajectories of the flights.

    In particular, a user who would like to increase the time horizon of the controller would need to keep the time-step $t_s = t_i \times t_w \times t_r$ fixed by reducing the time-roll, otherwise different effects will mix up.

- The life duration of shocks is computed with respect to the time step. Hence changing t_i, t_w, or t_r will change the duration of the flights.

- The shocks appear only on 10 flight levels, not on a whole column of air. The variables `shock_f_lvl_min` and `shock_f_lvl_max` do not change this fact. Instead, they are fixing the possible interval of flight levels of apparition of the shocks.

- The starting and ending dates have to be informed in the config file at the corresponding lines. They need to be consistent with the trajectories provided.

6 Miscellaneous

6.1 Trajectories formats

There are different formats for trajectories in the code and we use the following abbreviations in the in-code documentation to keep track of them:

- `(x, y, z, t)` : trajectories are made of 4-tuples with latitude, longitude, altitude and time.
- `(x, y, z, t, s)` : same with label of sector as fifth element.
- `(n), t` : trajectories are made of ONE 2-tuple. The first element is a list of labels of nodes, the second one is the time of entrance, i.e. the time of the first point of the trajectory.
- `(n, z), t` : same with altitude attached to each point.

There are also two formats of time:

- `t` : a float representing the number of minutes elapsed since the beginning of the day (which is stored somewhere else).
- `tt` : a tuple `(yy, mm, dd, h, m , s)`.

6.2 Post-processing and visualization

The code is shipped with several utilities in order to see the results. Some of them are more specifically useful:

- `draw_network_and_patches` in `libs/general_tools.py` is able to display an hybrid network as well as trajectories using it.
- the class `TrajConverter` in `libs/tools_airports.py` is a general converter for formats of trajectories. It features a clustering algorithm which is able to gather coordinate-based navigation points into labelled-based ones.
- `build_traffic_network` in `libs/tools_airports.py` is a function able to infer the traffic network from data. The traffic network of a set of trajectories is the network having the navigation points as nodes, which are linked if at least one flight goes from one to the other. Links are weighted with the number of flights travelling through the them. This functions uses the `TrajConverter`.
- `write_trajectories_for_tact` and `read_trajectories_for_tact` in `abm_strategic_utili` are useful functions for reading and writing on disk the trajectories generated.
- `draw_traffic_network` in `abm_strategic_utilities.py` is a wrapper of the `draw_network_and_patches`, which is useful to draw the trajectories generated with the strategic and the tactical. It uses also `build_traffic_network` and `read_trajectories_for_tact`.

- the script `show_trajectories` in the `scripts/` folder makes use of all the previous functions to display a set of trajectories using the syntax:

  ```
  ./how_trajectories.py trajectories_file.dat save_path_for_image
  ```

- `performance_plots.py` in the `abm_strategic/` folder includes many possible plotting functions and is easily usable with the strategic, in particular when sweeping parameters.
- the script `sweep_tactical` is useful when one wants to sweep some configuration parameters for tactical simulations.

6.3 Paths

Some paths are automatically inferred from the parameter dictionaries. For instance, the function `build_path` in simulationO.py takes some values of the parameters and builds file name with them. So one can easily override this by changing the function for its own purpose.

6.4 Going further

The description of the models are available in two different deliverable of the ELSA project, the D1.3 and the D2.4 of the extension of ELSA. They describe in details which mechanisms and algorithms are used, what is the modelling framework and so on, as well as the results obtained so far.

Other useful and more detailed information is contained in the *docstrings* of the Python code. The docstrings are just some comments within the code which can be extracted and gathered together in order to give a more comprehensive documentation. This documentation is generated during the setup and is included in the doc/ folder. It can be displayed with an internet browser, allowing hyperlinks between functions and classes. The C code has also some minimal comments.

The folder scripts/ contains different scripts that one can modify to suit its needs. They may or may not work with your configuration and data, so they should be considered as starting points for custom pieces of codes.

Finding some help

GitHub is a great tool to track bugs and other issues. If you run into trouble, you can open a new “issue” on GitHub by clicking on the tab on the right hand side of the main page of the ELSA simulator. We will try to tackle your issues as well as we can.