Getting Started with the Elsa Simulator

1 Setup

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.

1.1 Using git

git is install with 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 bsd-like system like Ubuntu. Then run:

```
git clone https://github.com/GGurtner/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, then extract it wherever you want.

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

On bsd-like system like Ubuntu, you can use `easy_install`. First install `easy_install` using:

```
sudo apt-get install easy_install
```

Then installing all modules:

```
easy_install scipy shapely networkx basemap MySQLdb maplotlib descartes
```

for tactical: TBD.

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.

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 their own purpose by feeding them with the right input. They can also be used conjointly, even if this feature is very basic at the moment. Indeed, the interface is still under construction and should feature more advanced capabilities for the public release (June or July) with a more integrated code.

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[1], 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. Note that most of them requires pickle files, but it is planned that in the future more formats will be supported, like jason.

4 Using the strategic ABM

The main functions concerning the ABM stratefic can be loaded via[2]:

```
from abm_strategic import *
```

The main functions are:

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

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

[2] 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[3].

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 abm_strategic/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.

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.

5 Rectification of the trajectories

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 *
```

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

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 effiency 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 to 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.

6 Miscellaneous

6.1 Trajectories formats

There are different formats for trajectories in the code and we use the following abbreviations in the 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 Visualization

The code is shipped with several utilities in order to see the results. Two 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.
- performance_plots.py includes many possible plotting function and is easily usable with the strategic, in particular when sweeping parameters.

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 of its own purpose.

7 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 go in details about which mechanisms and algorithms are used, what is the modelling framework and so on, as well as the results obtained so far.

The public release of this code will happen in June-July and will be shipped with a more detailed documentation about the features of the code.

The folder scripts/ contain 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 studies.

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 right hand side tab when you are on the main page of the ELSA simulator. We will try to tackle your issues as well as we can.