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 via the http address.

1.1 Using git

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

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

on a bsd-like system like Ubuntu. Then run:

```
git clone <link>
```

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 Using http

Just click on TBD.

1.3 Installing dependencies

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.

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.py'. 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 the 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.

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