SLAPP (Swarm-Like Protocol in Python) Reference Handbook

Pietro Terna
mailto:pietro.terna@unito.it

April 25, 2016

Contents

List of Figures

Chapter 1

Introduction

1.1 SLAPP and Swarm

SLAPP, as Swarm-Like Protocol in Python, is both a tutorial on agent-based programming and a shell to run large simulation projects, having in mind the original Swarm[1] scheme.

The repository of SLAPP is at https://github.com/terna/SLAPP.

To read about Swarm and SLAPP, to examine several SLAPP applications, and . . . a lot more, you can have a look to Boero *et al.* (2015).

1.2 Installing SLAPP

To install SLAPP you need to provide several Python libraries. To do that, please follow Appendix A.

1.3 The README and related files: discovering two ways of using SLAPP

The GitHub repository of SLAPP contains two `README` files.

- The `_readmeFirst.txt` file clarifies the content of the whole project.

  We have both a tutorial and an agent-based simulation shell, coming from the Swarm (http://www.swarm.org) project, and named SLAPP for Swarm-Like Agent Protocol in Python.

[1]About Swarm, have a look to Minar *et al.* (1996). You can access Swarm website via http://www.swarm.org. The project started at the Santa Fe Institute (first release: 1994). It represents a milestone in agent-based simulation.

You can find SLAPP as an Agent-based Model (ABM) shell, in the folder number 6.[2]

Both the basic scheme of the tutorial, and all the files having in their names the prefix `Swarm_original`, are coming from the tutorial that was distributed by the Swarm Development Group via the **`swarmapps`** file (the last version, that we use here, is swarmapps-objc-2.2-3.tar.gz).

Those files are unmodified in SLAPP, but the correction of a few typos.

We can find the original package at http://download.savannah.gnu.org/releases/swarm/apps/objc/sdg/swarmapps-objc-2.2-3.tar.gz or at http://nongnu.askapache.com/swarm/apps/objc/sdg/swarmapps-objc-2.2-3.tar.gz or at http://terna.to.it/swarm/swarmapps-objc-2.2-3.tar.gz.

- The `README.md` file, written using *Markdown*,[3] underlines that we have two possibile ways of using SLAPP: both as a tutorial on agent-based programming (see Section 1.3.1) or as an agent-based shell (see Section 1.3.2).

1.3.1 Using SLAPP as a tutorial on agent-based programming

- To study the tutorial, read the content of the file `SLAPP tutorial.txt`.
- The file `SLAPP tutorial.txt` guides the user through the development of a SLAPP model that makes use of a lot of the functionalities of Swarm.

  The model refers to the movement of a bug, randomly walking in a 2D space.

  We start introducing a very simple and plain program, with the bug taking a random walk. Through a progression of models, we introduce both object-oriented and Swarm style programming.

  Although this is a quite simple exercise, it shows how to build complex software from simple building blocks.

  In this folder, we have several subfolders, each with a complete application and a README file that helps you to walk through the code.

  You should start with the `1 plainProgrammingBug` folder, and then proceed in the following order (the start files have a number, corresponding to that of their folder):

  `1 plainProgrammingBug`

[2] *6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX*

[3] http://whatismarkdown.com

2 `basicObjectProgrammingBug`

3 `basicObjectProgrammingManyBugs`

4 `basicObjectProgrammingManyBugs_bugExternal_+_shuffle`

5 `objectSwarmModelBugs`

6 `objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`

7 `(toBeDeveloped_aFewHints)`

- We used Python to write the tutorial: you can find a lot of wonderful resources introducing the Python language. I suggest Downey (2012), a book that you can also read online at the address reported in the references; the book also exists in a slight different version as a learning interactive tool (Elkner *et al.*, 2013). An alternative way to start learning Python is the introductory part of the wonderful online book of Sargent and Stachurski (2013) on quantitative economics. (There, you can also find an introduction to Julia[4], a quite new and highly powerful language.)

- We report here the file `Swarm_original README in tutorial folder.txt`, related to the original tutorial. The file is in the main folder of the repository.

  Note that the names of the txt files, here and in the subfolders, start with the prefixes `SLAPP` or `Swarm_original`. This choice is just to underline if we are referring the to my reformulation in Python or to the original Swarm. (Swarm was based on Objective C[5] and successively also partially on Java; the tutorial was strictly in Objective C).

1.3.2 Using SLAPP as an agent-based shell

- To start running the agent-based shell, you can read the content of the file:
  `SLAPP shell.txt`
  and install the required libraries (to install them, you can follow the explanations of Appendix A).
  Then open a terminal, go into the SLAPP main folder (we suppose here that it is in

  `/Users/pt/GitHub/SLAPP`

  but this is relative to my computer and my initials "pt") and:[6]

[4] `http://julialang.org`; highly interesting `https://www.juliabox.org`
[5] `https://en.wikipedia.org/wiki/Objective-C`
[6] About *running* SLAPP, read Section 2.1.

1 - launch the application `basic` as in Figure 1.1:[7]

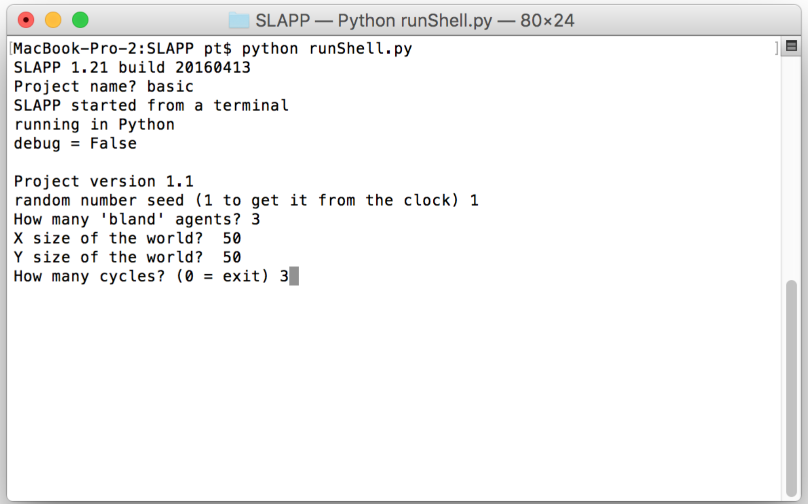
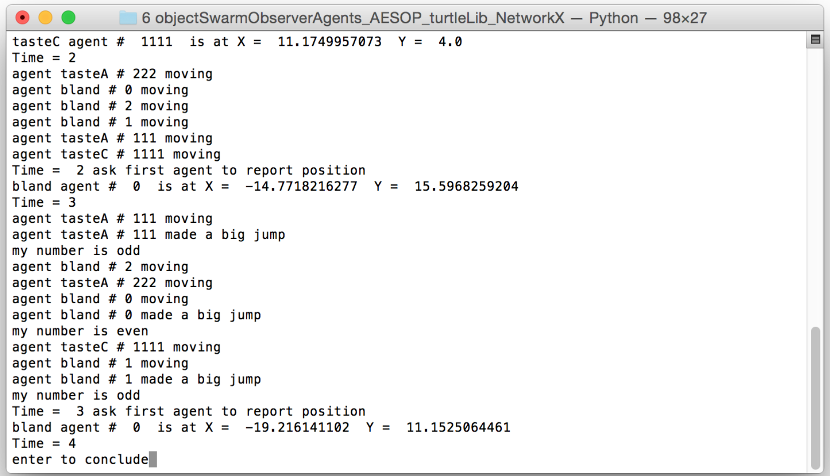


Figure 1.1: Starting the *basic* project

The effect is (plain text output only):

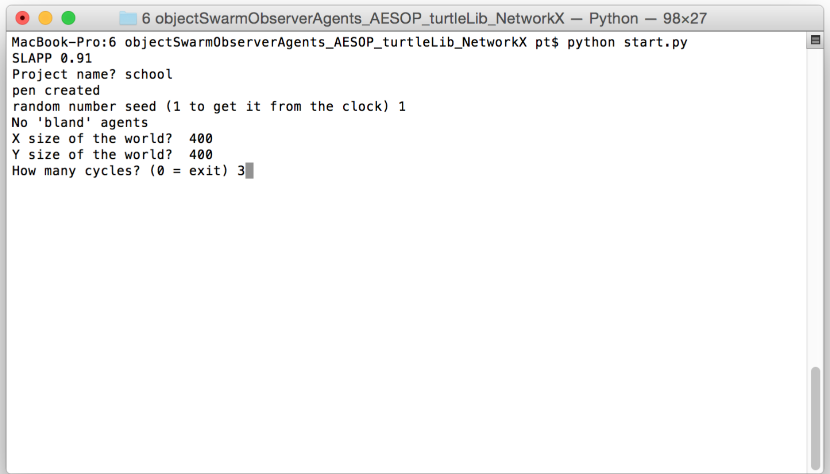
[7] Launching ways are introduced in the initial *bullets* of Chapter 2. In Figures 1.3 and 1.6 we use the second launching methods of Chapter 2



```
6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX — Python — 98×27
tasteC agent #  1111  is at X =  11.1749957073  Y =  4.0
Time = 2
agent tasteA # 222 moving
agent bland # 0 moving
agent bland # 2 moving
agent bland # 1 moving
agent tasteA # 111 moving
agent tasteC # 1111 moving
Time =  2 ask first agent to report position
bland agent #  0  is at X =  -14.7718216277  Y =  15.5968259204
Time = 3
agent tasteA # 111 moving
agent tasteA # 111 made a big jump
my number is odd
agent bland # 2 moving
agent tasteA # 222 moving
agent bland # 0 moving
agent bland # 0 made a big jump
my number is even
agent tasteC # 1111 moving
agent bland # 1 moving
agent bland # 1 made a big jump
my number is odd
Time =  3 ask first agent to report position
bland agent #  0  is at X =  -19.216141102  Y =  11.1525064461
Time = 4
enter to conclude
```

Figure 1.2: The output of the *basic* project

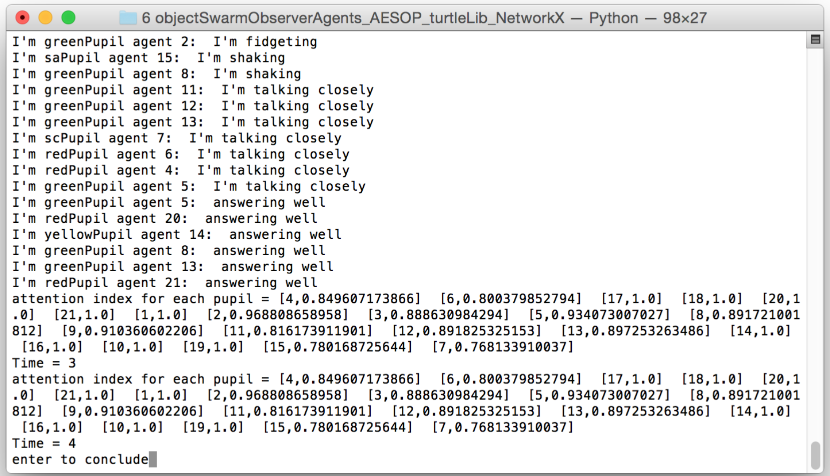
2 - launch the application "school" as in the following window:



```
6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX — Python — 98×27
MacBook-Pro:6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX pt$ python start.py
SLAPP 0.91
Project name? school
pen created
random number seed (1 to get it from the clock) 1
No 'bland' agents
X size of the world?  400
Y size of the world?  400
How many cycles? (0 = exit) 3
```

Figure 1.3: Starting the *school* project

The effect is (plain text output):



```
6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX — Python — 98×27
I'm greenPupil agent 2:  I'm fidgeting
I'm saPupil agent 15:  I'm shaking
I'm greenPupil agent 8:  I'm shaking
I'm greenPupil agent 11:  I'm talking closely
I'm greenPupil agent 12:  I'm talking closely
I'm greenPupil agent 13:  I'm talking closely
I'm scPupil agent 7:  I'm talking closely
I'm redPupil agent 6:  I'm talking closely
I'm redPupil agent 4:  I'm talking closely
I'm greenPupil agent 5:  I'm talking closely
I'm greenPupil agent 5:  answering well
I'm redPupil agent 20:  answering well
I'm yellowPupil agent 14:  answering well
I'm greenPupil agent 8:  answering well
I'm greenPupil agent 13:  answering well
I'm redPupil agent 21:  answering well
attention index for each pupil = [4,0.849607173866]  [6,0.800379852794]  [17,1.0]  [18,1.0]  [20,1
.0]  [21,1.0]  [1,1.0]  [2,0.968808658958]  [3,0.888630984294]  [5,0.934073007027]  [8,0.891721001
812]  [9,0.910360602206]  [11,0.816173911901]  [12,0.891825325153]  [13,0.897253263486]  [14,1.0]
 [16,1.0]  [10,1.0]  [19,1.0]  [15,0.780168725644]  [7,0.768133910037]
Time = 3
attention index for each pupil = [4,0.849607173866]  [6,0.800379852794]  [17,1.0]  [18,1.0]  [20,1
.0]  [21,1.0]  [1,1.0]  [2,0.968808658958]  [3,0.888630984294]  [5,0.934073007027]  [8,0.891721001
812]  [9,0.910360602206]  [11,0.816173911901]  [12,0.891825325153]  [13,0.897253263486]  [14,1.0]
 [16,1.0]  [10,1.0]  [19,1.0]  [15,0.780168725644]  [7,0.768133910037]
Time = 4
enter to conclude
```

Figure 1.4: The plain text output of the *school* project

and as graphical output:

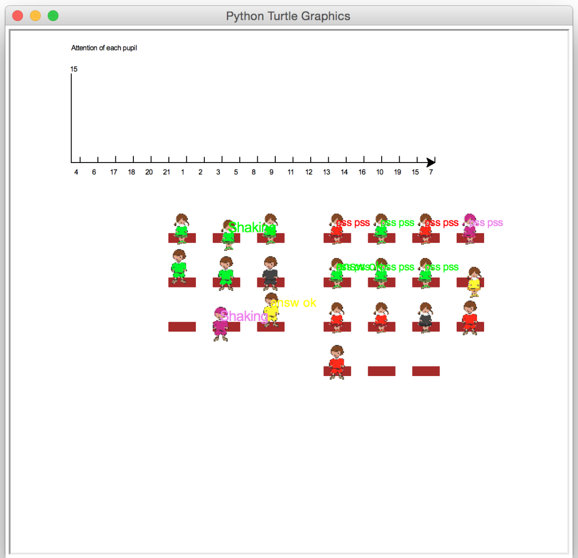
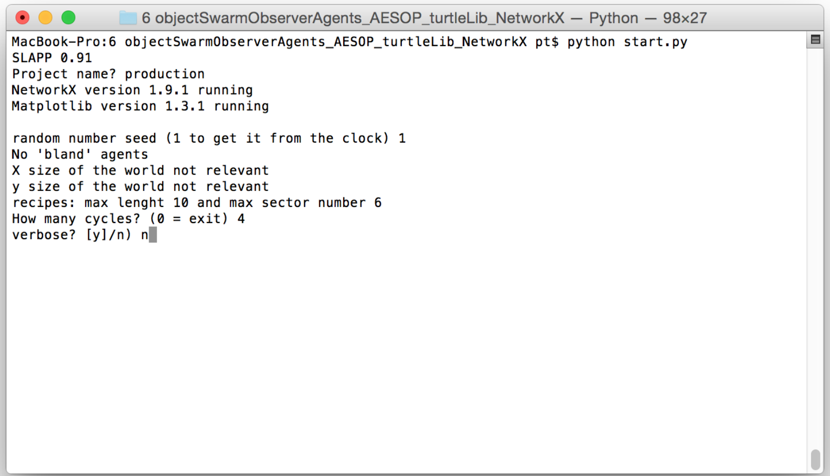


Figure 1.5: The graphical output of the *school* project

3 - launch the application "production" as in the following window:



```
6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX — Python — 98×27
MacBook-Pro:6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX pt$ python start.py
SLAPP 0.91
Project name? production
NetworkX version 1.9.1 running
Matplotlib version 1.3.1 running

random number seed (1 to get it from the clock) 1
No 'bland' agents
X size of the world not relevant
y size of the world not relevant
recipes: max lenght 10 and max sector number 6
How many cycles? (0 = exit) 4
verbose? [y]/n) n
```

Figure 1.6: Starting the *production* project

The effect is (plain text output):

```
6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX — bash — 94×24
1001 [1005, 1002]
1006 [1004, 1001]
1004 [1005, 1004, 1001]
1002 [1004, 1006, 1003]

betweenness_centrality
1001 4.5
1002 9.0
1003 0.0
1004 1.5
1005 1.5
1006 0.5

closeness_centrality
1001 0.625
1002 0.714285714286
1003 0.0
1004 0.5
1005 0.625
1006 0.555555555556
Hit enter key to continue
Time = 5
enter to conclude
MacBook-Pro:6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX pt$
```

Figure 1.7: The plain text output of the *production* project

and as graphical output:

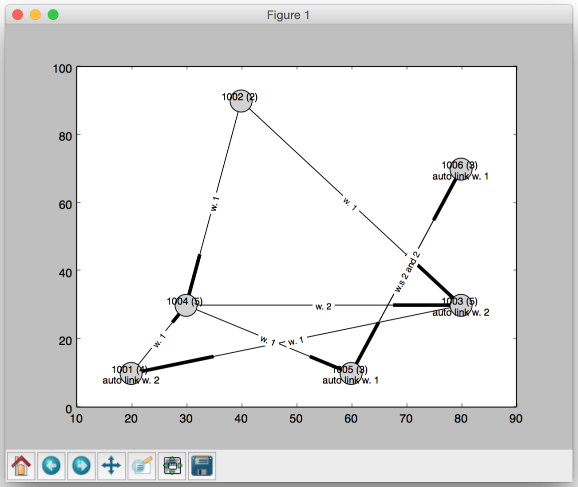


Figure 1.8: The graphical output of the *production* project

Chapter 2

The *basic* project as a guide to the making of a new project

The `basic` project (in Section 1.3.2 you had a view on it) is introduced to familiarize with SLAPP.

2.1 How to *run* SLAPP

The starting phase is introduced in the next *bullet*, in a detailed way.

- In the SLAPP distribution, we have the `basic` folder[8], containing the introductory application.
  - We can launch SLAPP as a simulation shell—via the `runShell.py` file that we find in the main folder of SLAPP—from a terminal, with:[9]
    `python runShell.py`
  - Alternatively, we can launch SLAPP as a simulation shell—via the `start.py` file that we find in the folder of SLAPP, i.e.
    `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`—from a terminal, with:
    `python start.py`
  - To use SLAPP in IPython (in a `jupyter notebook`) go to the main folder of SLAPP via a terminal and then start
    `jupyter notebook`
    and finally click on `iRunShell.ipynb`.

[8]Within the *6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX* folder.

[9]Is is not possibile to run *start.py* or *runShell.py* via Python dedicated shells, such as IDLE, Spyder, ... It is instead possible to use Spyder to run *start.py* or *runShell.py* in IPython.

  - We can also run SLAPP in IPython via the Spyder environment, running `%run start.py` or `%run runShell.py`, going to their folder with `%cd` followed by the path to the folder.
  - We can also run SLAPP from a Jupiter QtConsole—e.g. Anaconda launcher— running `%run start.py` or `%run runShell.py`, going to their folder with `%cd` followed by the path to the folder.
  - A further possibility is that of launching IPython from a terminal with `ipython` command line, being in the SLAPP directory, and then executing `%run runShell.py`, In this case the graphic results will be the same of the execution from a terminal using Python.

  In all cases, we immediately receive the request of choosing a project:
  `Project name?`

- In our case, we reply `basic` (or `school` or `production`, for the other examples). To create a new project, we simply add a new folder; the folder name will also be automatically that of the project, and we will choose it at the prompt above.

  We also have a special folder, named `$$slapp$$`,[10] that the user is not supposed to modify. It is the folder where we store the kernel of SLAPP, i.e., its simulation engine. If you do not modify it, always building your applications in separate folders, your work will not be affected by the modifications introduced by the new versions of SLAPP.

- We can predefine a default project: if we place *in the main SLAPP folder or in the folder* `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX` a file named `project.txt` containing the path to a folder (`basicTmp` as an example, so the content of the file could be something as `/Users/pt/Desktop/basicTmp`[11]), and the initial message of SLAPP will be:

```
path and project = /Users/pt/Desktop/basicTmp
do you confirm? ([y]/n):
```

  The feature is useful in two perspectives: (i) we can place our projects outside the SLAPP folder; (ii) we can avoid typing the name of the project when, in the debugging phase, we launch it a lot of times.

- Resuming the explanation: now we are looking at the message:

[10] Always within *6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX*.

[11] In Windows it would be better to use backslashes "\" instead of slashes "/". Anyway (verified in Windows 10) also slashes work.

```
running in Python
debug = False

Project version 1.0
random number seed (1 to get it from the clock)
```

The `Project version` message is implemented as a suggestion only in the `basic` project, specifying the version of the project into the file `commonVar.py` and managing it via the file `parameters.py`, both in the project folder.

To reply to the open question about the random seed, we have to enter an integer number (positive or negative) to trigger the sequence of the random numbers used internally by the simulation code. If we reply 1, the seed—used to start the generation of the random series—comes from the internal value of the clock at that instant of time. So it is different anytime we start a simulation run. This reply is useful to reproduce the simulated experiments with different conditions. If we chose a number different from 1, the random sequence would be repeated anytime we will use that seed. This solution is useful (1) while debugging, when we need to repeat exactly the sequence generating some error, but also (2) to give to the user the possibility of replicating exactly an experiment.

The `running in Python` sentence signals the we are running the program in plain Python. Alternatively, the message could be `running in IPython`, following Chapter 5.

- Then the code asks us to enter the number of unspecified agents; this is related to the AESOP (Agents and Emergencies for Simulating Organizations in Python) perspective, introduced below as an abstract layer upon SLAPP. There we have both well-defined agents (*tasty*) and unspecified ones (*bland*).

  ```
  How many 'bland' agents?
  ```

  Finally, after a few information, we have to enter the number of the cycles we want:

  ```
  X size of the world?  50
  Y size of the world?  50
  How many cycles? (0 = exit)
  ```

  Replying 2 as the number of bland agents and 4 as the number of cycles, we obtain the output reported (only the final part) in Figure 1.2.

- We introduce now time management, split into several (consistent) levels of scheduling.

  The general picture is that of Figure 2.1: in an abstract way we can imagine to have a clock opening a series of containers or boxes. Behind the metaphor of the boxes, in SLAPP, as it was in Swarm, we have the *action groups*, where we store the information about the actions to be done.[12]

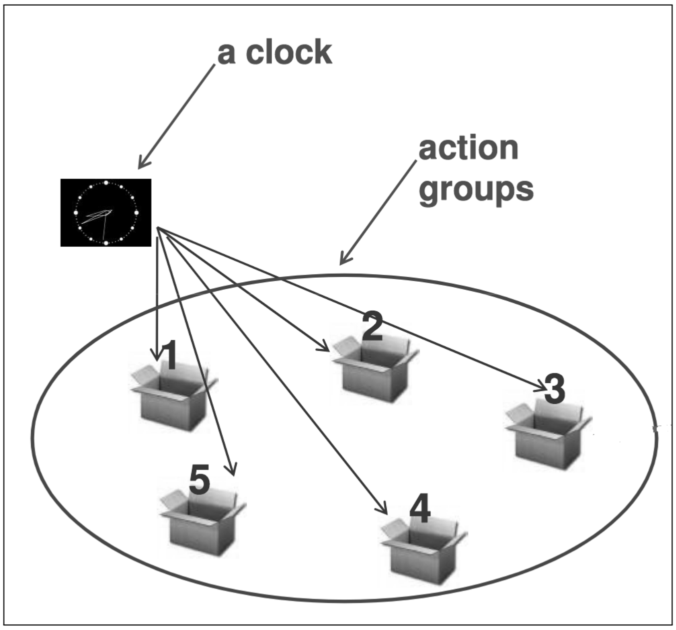


Figure 2.1: The representation of the schedule

  Imagining the events as objects, in the object-oriented programming perspective, is one of the key points of success in the original Swarm system. We implement the same idea in SLAPP.

2.2 Scheduling

In SLAPP, we have the following three schedule mechanisms, or processes, driving the events.

- The first mechanism is at the level of the Observer (Section 2.2.1) and the second one at the level of the Model (Section 2.2.2), both with recurrent

[12] The structure is highly dynamical because we can associate a probability to an event, or an agent of the simulation can be programmed to add or eliminate one or more events into the *boxes* or, better, into the *action groups*.

sequences of action to be done.[13] We will introduce the third mechanism, more detailed, in Section 2.2.3.

- In our `basic` code, these sequences are reported in the files
  `observerActions.txt` and
  `modelActions.txt`
  in folder `basic`.[14]

2.2.1 The scheduling mechanism at the level of the Observer

.

- To discover the first schedule mechanism, we refer to the first file (`observerActions.txt`), containing (row changes are not relevant):

  ```
  modelStep ask_all clock
  modelStep ask_one clock
  modelStep ask_one clock
  ```

  The interpretation is the following.

  - First of all, we have to take into consideration that the execution of the content of the file is "with repetition", until an `end` item will appear (see below). If we do not need differentiations within the repetition cycle, also a content as the following should work:

    ```
    modelStep ask_all clock
    ```

    However, the content can be as articulated as we need.

  - `modelStep` orders to the model to make a step in time. The order acts via the code of the file ObserverSwarm.py[15], where we have (example *i*) a simple rule ordering to the Model code to make a step.

  - `ask_all` orders to all the agents to talk. In this case, always in ObserverSwarm.py, we have (example *ii*) one of the four stable instances[16]

[13] The level of the Observer is our level, where the experimenter looks at the model (the level of the Model) while it runs. This structure is a key feature in Swarm, and so we reproduce it in SLAPP. Other simulation shells follows the same scheme: as an example, the observer is a key feature in NetLogo `https://ccl.northwestern.edu/netlogo/`.

[14] Within folder *6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX*.

[15] Which is in the "$$slapp$$" folder (see above in this Chapter).

[16] The instances of the class *ActionGroup* contained in the file *ActionGroup.py* in folder *$$slapp$$* are related to: "clock"; "visualizeNet", used with network analysis; "ask_all"; and "ask_one".

of the class ActionGroup within the Observer. That related to `ask_all` contains the `do2a` variable, linking a method which is specified as a function in the file `oActions.py` of folder `basic`. In this way, the application of the basic method `ask_all` can be flexibly tailored to the specific applications.

 - `clock` ask the clock to increase its counter of one unit. When the count will reach the value we have entered replying to the `How many cycles?` query, the ActionGroup instance (example *iii*) related to the clock (`actionGroup1` in ObserverSwarm.py) will add the `end` item into the sequence of the file `observerActions.txt`. The item is placed immediately after the `clock` call. The `end` item stops the sequence contained in the file.
 - `ask_one` orders to the first component of the agent collection to talk. As above (example *ii*, being this the example *iv*), we have an instance of the class ActionGroup within the Observer. That related to `ask_one` contains the `do2b` variable, linking a method which is specified as a function in the file `oActions.py` in the folder `basic`. In this way, the application of the basic method `ask_one` can be flexibly tailored to the specific applications.
 - It is useful to underline that the example (*i*) has no reference in the file `oActions.py`. We can add similar items for the scheduling, directly "wiring" them via the function
 `def otherSubSteps(subStep, address):`
 in `oActions.py`, without modifying
 `ObserverSwarm.py` in `$$slapp$$`
 (look at the `production` project to see how, with `pause` and `prune`).

- The examples (*ii*), (*iii*), and (*iv*) use the double structure of the instance of the class ActionGroup and of the related method[17] construction that we have in `ObserverSwarm.py` (in `$$slapp$$`), with the definition in `oActions.py` of the folder `basic` (in our current case). It is a more complicated structure, but very flexible.
- Looking at the `oActions.py` files of the other projects (currently, `school` and `production`), you can analyze the different ways of using the options (*i*), (*ii*), (*iii*), and (*iv*).

[17]Technically, our *pseudo*-methods—that we pass to the instance via a variable—are always functions. So, we have to manage explicitly the value of the usual *self* value. To avoid any possible confusion, the term used in these cases—into the SLAPP code—is *address*.

- If we use a missing keyword in the files collecting the first two levels of scheduling, i.e. `observerActions.txt` or `modelActions.txt`—maybe in error or referring to a not jet implemented item—we receive a *warning*. See:
  `Warning: step ask_on not found in Observer`
  where the item `ask_one` is misspelled,

2.2.2 The scheduling mechanism at the level of the Model

.

- The file `modelActions.txt`, quoted above at the beginning of Section 2.2, is related to the second schedule mechanism: that of the Model. (About the Observer/Model dualism, the reference is to note 13.)

  The file contains (unique row, remembering that row changes are not relevant in this group of files):

  ```
  reset move read_script
  ```

  The interpretation is the following.

  – Also at the Model level, we have to take into consideration that the execution of the content of the file is "with repetition", never ending. It is the Observer that stops the experiment, operating at its level.

  – `reset` orders to the agents to make a reset, related to their variables. The variables can be specified as explained in the next few rows. The order acts via the code of the file ModelSwarm.py[18]. In this case, always in ModelSwarm.py, we have (example I) one of the three stable instances[19] of the class ActionGroup within the Model.

    The item `reset` contains the `do0` variable, linking a method that is specified as a function in the file `mActions.py` in the folder `basic`. The application of the basic method `reset` can so be flexibly tailored to the specific applications, defining which are the variables we are reseting.

    In our specific case, the content of the `do0` function in `mActions.py` asks all the agents to execute the method `setNewCycleValues`. The method is defined in an instrumental file (`agTools.py` in `$$slapp$$`) and it is, as default, doing nothing. We can redefined it in `Agent.py` in

[18]That is in the "$$slapp$$" folder (see above in this Chapter).

[19]The instances of the class *ActionGroup* contained in the file *ActionGroup.py* in folder *$$slapp$$* are related to: "reset"; "move"; and "read_script"

the project folder. Into the `basic` project, `reset` is not operating, but it is reported above as a memo for future uses.

The case is strictly similar to the examples *ii*, and subsequent ones, introduced above (Section 2.2.1).

– `move` orders to the agents to move. The order acts via the code of file ModelSwarm.py. We have here (example *II*) the second of the three stable instances of the class ActionGroup within the Model. That related to `move` contains the `do1` variable, linking a method that is specified as a function in the file `mActions.py` in the folder `basic`. In this way, the application of the basic method `move` can be flexibly tailored to the specific applications, defining what kind of movement (if any) we order to the agents.

  In our specific case, the content of the `do1` function in `mActions.py` asks all the agents to execute the method `randomMovement`. We defined that method in the file `Agent.py`, in the project folder.

  The case is strictly similar to the examples *ii*, and subsequent ones, introduced above (Section 2.2.1).

  The structure managing the movement is quite complicated, just to propose a not trivial example.

  The Python code (in `mActions.py`) determining the movement is:

```
askEachAgentInCollectionAndExecLocalCode \
      (address.agentListCopy, Agent.randomMovement,
                      jump=random.uniform(0,5))
```

  A few details:

  * `address` substitutes the implicit usual *self* as explained above in Section 2.2.1;
  * `agentListCopy` is a shuffled copy to ask the agent to move in an ever-changing sequence;
  * `Agent.randomMovement` is the address (within the class) of the method that we send to the agent list;
    an example helps to clarify (we are here using Python interactively, in a shell):

```
>>> class A:
        def __init__(self,b):
            self.b=b
         def prnt(self):
           print self.b
```

```
>>> a=A(10)
>>> aa=A(100)
>>> a.prnt()
10
>>> aa.prnt()
100
>>> A.prnt
<unbound method A.prnt>
>>> A.prnt(a)
10
>>> A.prnt(aa)
100
```

* `jump=random.uniform(0,5)` is optional; if it is placed there, it assigns a random value to a dictionary key named *jump*.

The method `randomMovement`, reported in Agent.py in folder `basic` (this example), is defined with an optional[20] dictionary in the head, as:

```
def randomMovement(self,**k):
```

The call to the method assigns a default value to the key `jump`; the method verifies its existence;

```
self.jump=1
     if k.has_key("jump"): self.jump=k["jump"]
```

The value of the `jump` multiplies the length of the movement.

– `read_script` orders to the Model to open a new level of scheduling, described in Section 2.2.3. The order acts via the code of file ModelSwarm.py. We have here (example *III*) the third of the stable instances

[20] The optional dictionary works as in the following example (created interactively in a Python shell):

```
>>>class A:
      def test(self,**k):
         self.b=1
         if k.has_key('b'):self.b=k['b']

>>> a=A()
>>> a.test()
>>> a.b
1
>>> a.test(b=10)
>>> a.b
10
```

of the class ActionGroup within the Model. The ActionGroup related to `read_script` item is the `actionGroup100` that contains the `do100` function, used internally within `ModelSwarm.py` to manage the script reported into the `schedule.xls` file (or directly into the `schedule.txt` one).

- The method `randomMovement` defined in Agent.py in folder `basic` of this project, is also interesting, as it introduces the feature of the *local code execution*.

  The code is:

```
if int(self.number/2)*2 == self.number:
      oddEven = "print 'my number is even'"
else: oddEven = "print 'my number is odd'"
setLocalCode("print 'agent %s # %d made a big jump';"\
         % (self.agType,self.number) +\
         oddEven)
```

  We use the function `setLocalCode` of `Tools.py` (in `$$slapp$$`) to define a code to be executed "on the fly" via the function
  `askEachAgentInCollectionAndExecLocalCode`
  of `Tools.py`. The function simply executes:

```
exec(localCode)
```

  having received the code to be executed (if you want to replicate this kind of code implementation, have a look both to the function and to its internal links). In this way, we have a flexible and powerful way for adding activities in our agents.

  We can employ the *local code execution* also within the agents' methods used in the third scheduling mechanism described in Section 2.2.3.

- The cases (I), (II) shown above are conceptually similar to the cases (ii) and subsequent ones, seen above (Section 2.2.1); instead, the case (III) is quite special.

  We can also have schedule structures as the i above (always Section 2.2.1, adding the code after
  `def otherSubSteps(subStep, address):`
  in `mActions.py`.

  We have an example of this solution in the project `production`.

2.2.3 The detailed scheduling mechanism within the Model (AESOP level)

.

- The third scheduling mechanism, as anticipated in Section 2.2, is based on a detailed script system that the Model executes while the time is running. The time is managed by the `clock` item in the sequence of the Observer.

  The script system is activated by the item `read_script` in the sequence of the Model.

- This kind of script system does not exist in Swarm, so it is a specific feature of SLAPP, introduced as implementation of the AESOP (Agents and Emergencies for Simulating Organizations in Python) idea: a layer that describes in a fine-grained way the actions of the agents in our simulation models.

- Let us deepen the scheduling hierarchy, with the three levels:

  – at the Observer level (via the file `observerActions.txt`) we run a high level sequence of events: (a) one of the events is the request to the Model of making a step (`modelstep`) and (b) another is the request to the `clock` to go by;

  – at the Model level (via the file `modelActions.txt`) we run a medium level sequence of events (a sub-cycle within the previous one): one of the events is now the request (`readScript`) to the fine-grained scripting system (if any) to execute the *action container* related to the time step we are in;

  – at the AESOP level (i.e. within the Model detailed scripting system) we activate the set of rules and actions introduced in this Section.

- At the AESOP layer, the action containers are specified—with the `#` indicators (see below)—upon the time; we put them into a spreadsheet describing the actions and the subjects doing them.

  We adopt the spreadsheet formalism because it is well known and diffused, but you can bypass it creating directly a text file containing the same elements, as explained below.

  Other details are in the files `SLAPP 6 objectSwarmObserverAgents.txt` and `a_note_on_AESOP.txt` in the usual folder `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.

- Now we take in a detailed exam an example of AESOP layer schedule (the third layer), i.e. the timetable where we describe minutely the actions that the agents are doing at each time step.

- The file `schedule.xls` can be composed of several sheets, with: (a) the first one with name `schedule`; (b) the other ones with any name (those names are *macro* names.)[21] We can recall the macro instructions in any sheet, but not within the sheet that creates the macro (that with the same name of the macro), to avoid infinite loops.

  We start with the sheet in `scheduleBase.xls` of folder `basic`. To use a sheet, you have to rename it to `schedule.xls` (keep safe the original file).

  Within the sheet, we have the action containers as introduce above, starting with the sign `#`.

  In `scheduleBase.xls` we have (comments start at column E):

```
                          COL. E

                          comments here or in successive columns
#       1                 standard (background) actions, like move, are applied to "all"
bland   eat               bland agents are those not specified in dedicated .txt files,
bland   dance             with the related names reported in the agTypeFile.txt file
#       2
#       4
all     0.5    dance      all agents acting
tasteC  eat               tasteC agents acting
#       5
all     eat
all     dance
#       7
tasteA  0.5    dance      tasteA agents acting
#       8
tasteB  dance             tasteB agents acting (no agents of this type exist here)
```

- In column `A`, we can place: (i) the sign `#` or (ii) the word `macro` or (iii) a *name* identifying a group of agents (the number of the agents in the group can vary from 1 to any value; about the groups, see Section 2.3.1):

  – with `#` (action container), we state that, when the clock reaches the time (in units of the conventional internal clock of the model) set in column `B`, the content of the rows[22] following that containing the `#` sign and until the next similar sign, will be executed;

  – with a *macro* name, we indicate that at a given time, SLAPP will activate the set of instructions reported in a specific sheet; see below for macros;

[21] We deeply use *macros* in project *school*.

[22] Maybe none, or: (i) one or more, empty; (ii) one or more, with operating contents.

- with a *name* identifying a set of agents[23] in column `A`, we send to this/these agent/s the method (as an action to be done) set in column `B` in a deterministic way; if in column `B` we have a number, this is the probability (related to 1) of execution of the method, in this case, reported in column `C`; the probability can be close to 0, but always ≥ 0;
    - the probability can be interpreted both as: (a) the share of the set of agents—recalled in column `A`—requested to act; or (b)—which is quite the same—as the probability of each individual of the same list to be in turn of executing the action;
    - if the number in column `B` is both less than `"0"` and integer, exactly that number (multiplied times -1 to have it positive) of agents is asked to execute the actions; the agents are randomly extracted from the list. We use this feature both in project `school` and in project `production`.

- The containers of action identified by the `#` sign can be also introduced in a nonsequential way into each sheet of the spreadsheet If we repeat the same `#` *time* pair in the same sheet, only the last one is kept. The `#` sign can be employed into the macro sheets (also repeating a `#` *time* place card already existing in another sheet); when the macro is called, the content of those time blocks is properly placed in the related time steps, orderly (e.g., a block coming from the third sheet will be placed after a block coming from the second and before a block coming from the fourth sheet).

- *Time loops*: we can also manage time loops using a block such as

```
#       1       3
somethingA
somethingB
```

that will be internally transformed to

```
#       1
somethingA
somethingB
#       2
somethingA
somethingB
```

[23]Both coming from the agTypeFile.txt list or from the agOperatingSets.txt one; in this example we do not have operating sets.

```
#      3
somethingA
somethingB
```

It is not possible to use the *time loops feature* operating directly into the `schedule.txt` file. Sure, we can repeat block using copy and paste and modifying by hand the variable parts.

- We also have a more complicated schedule in the file `scheduleBaseWithMacros_WorldState.xls` (to be copied to `schedule.txt` for the use), where we employ both the WorldState feature and macros.

  Running a project, at the beginning of the output, we read:

  ```
  World state number  0  has been created.
  ```

  What does it mean?

  The WorldState class interacts with the agents; at present, in the 'basic' case and in the other ones, we have a unique instance of the class, but the code is any way built upon a list of any number of instances of the class. The variables managed via WordState have to be added, with their methods, within the class, following the existing example in project 'basic', where WorldState has set/get methods for the variable `generalMovingProb`.

  In `Agent.py` of `basic` the method `randomMovement` asks to the WorldState the probability threshold to be compared with a random value, to decide to move. By construction the default threshold is 1 (move always); if we modify it to 0.1 as in the example below, movements will a lot less frequents.

  In `scheduleBaseWithMacros_WorldState.xls` we have:

  – in sheet *schedule* (comments not reported here)

  ```
  #           1
  bland       eat
  bland       dance
  #           2
  #           4
  all         0.5   dance
  tasteC      eat
  WorldState  0.1   setGeneralMovingProb
  #           5
  all         eat
  all         dance
  #           7
  tasteA      0.5   dance
  #           8
  tasteB      dance

  macro       repeat

  macro       dancing
  ```

– in sheet *dancing*

```
#           9
all         dance
#            10
tasteC      dance
```

– in sheet *repeat*

```
#           5     10
tasteA      eat
```

- In the example we also have the use of *macros* (as for *time loops*, macros cannot be programmed directly in the file `schedule.txt`.

  In the example above we have two macros, defined in the sheets `dancing` and "repeat". The effect in file `schedule.txt` follows. (NB, the call to a macro can be repeated, mainly if the macro has no time reference via `#` sign).

- The content of the file `schedule.txt` with the effects of the macros:

```
# 1
bland eat
bland dance
# 2
# 4
all 0.5 dance
tasteC eat
WorldState 0.1 setGeneralMovingProb
# 5
all eat
all dance
tasteA eat
# 6
tasteA eat
# 7
tasteA 0.5 dance
tasteA eat
# 8
tasteB dance
tasteA eat
# 9
tasteA eat
all dance
# 10
tasteC dance
```

- `If` structures can be easily implemented, as in the `school` project, where the file `scheduleIf.xls` (in sheet `checkToObtainAttention`) contains the row:

  `saPupil shakeIf_greenPupil`

  The method `shakeIf_greenPupil` is developed on `Agent.py` in folder `school`. It orders to the agents of type `saPupil` (a unique one, in this case) to shake but only if at least one of the agents of the group greenPupil has been shaking (verified checking their last executed method).

2.3 The agents and their sets

We have files containing the agents of the different types. Those files are listed in a file with name `agTypeFile.txt`: in our case, it simply contains the record `tasteA tasteB tasteC`. The names have no interpretation here; this is just an example.

- `tasteA.txt` lists the agents of type ("taste") `A`; in our case it reports only the identification numbers:

  ```
  111
  222
  ```

- `tasteC.txt` lists the unique agent of type ("taste") `C`, with the identifying number:

  ```
  1111
  ```

- `tasteB` agent are missing, so we have no file `tasteB.txt`; lacking the file, we receive the message:

  ```
  No tasteB agents: lacking the specific file tasteB.txt
  ```

  Instead, a `tasteB.txt` file empty (zero bytes or containing a few spaces) in the folder, would eliminate the message above. The program will raise no errors in the execution in any case.

The agents are created by `ModelSwarm.py` (in folder `$$slapp$$`) via the application specialized file `mActions.py` (in folder `basic`).

```
def createTheAgent(self,line,num,leftX,rightX,bottomY,topY,agType):
                #explictly pass self, here we use a function
                #print "leftX,rightX,bottomY,topY", leftX,rightX,bottomY,topY

                if len(line.split())==1:
                 anAgent = Agent(num, self.worldStateList[0],
                          random.randint(leftX,rightX),
                          random.randint(bottomY,topY),
                          leftX,rightX,bottomY,topY,agType=agType)
                 self.agentList.append(anAgent)

                else:
                 print "Error in file "+agType+".txt"
                 os.sys.exit(1)
```

Each project has an analogue structure dedicated to its agents.
The following bullets describe how this code works.

- As an ex-ante information, the identifying number of the agent is read outside this function, as a mandatory first element in the lines of any file containing agent descriptions. Also, the content of the `agType` variable is coming from outside, being the name of the agent file currently open.

- We check the input file, which—in the case of the project *basic*—has to contain a unique datum per row.

  Other projects can have several data in each row, related to multiple attributes of each agent. [24]

  Each agent is added to the `agentList`.

2.3.1 Sets of agents

The files containing the agents are of two families, the second one with two types of files:

- files listing the agents with their characteristics (if any): in folder `basic` we have the files `tasteA.txt` and `tasteC.txt`;

- files defining groups of agents:

  – the list of the types of agents (mandatory; from this list SLAPP searches the file describing the agents (first bullet here above); in folder `basic` we have the file `agTypeFile.txt` (the name of this file is mandatory) containing:

    ```
    tasteA tasteB tasteC
    ```

  – the list of the operating sets of agents (optional); in folder `basic` this file is missing. Indeed we receive the message
    `Warning: operating sets not found.`
    In `school` project we have the file `agOperatingSets.txt` (the name of this file is mandatory), with content:

    ```
    threeGreen leftS rightS r1l r2l r3l r1r r2r r3r r4r
    lRow cZone 1Pupil 2Pupil 3Pupil 4Pupil 5Pupil
    ```

[24] The files defining each set of agents can also have the extension *.txtx*. In case, they will be translated in regular *.txt* file, as explained in Section 2.3.2.

```
6Pupil 7Pupil 8Pupil 9Pupil 10Pupil 11Pupil
12Pupil 13Pupil 14Pupil 15Pupil 16Pupil 17Pupil
18Pupil 19Pupil 20Pupil 21Pupil
```

All the names contained in the file are related to other `.txt` files reporting the identifiers of agents specified in the lists of the previous bullet. The goal of this feature is that of managing clusters of agents, recalling them as names in Col. `A` in Section 2.2.3.

2.3.2 The use of files .txtx to define the agents

The files with extension `.txtx` (*txt eXtended*) are used to define the agents in a flexibile way.

An example is reported in the `basic` project where the tasteA.txt file contains:

```
111
222
```

and generates the taste A agents with id (number) 111 and 222.

Redefining `tasteA.txtx_` to `tasteA.txtx` the mechanism activated by the files `.txtx` operates. The file `tasteA.txtx` contains:

```
111@120
```

and produces a file `tasteA.txt` containing:

```
111
112
113
114
115
116
117
118
119
120
```

(to roll back copy `tasteA.txt_` to `tasteA.txt`). Definitions:

- n and v are mandatory names;
- n is the value in first position of the record (cannot be a formula);
- v is the result of the calculation in a formula;

- & starts and concludes a formula.

We can have more than one formula in a row of the `.txtx` file. Typical complex rows (with the extension `.txtx` as *eXtended txt*) are:

```
1@3 1 2 &if n==1:v=100#else:v=10& 3
```

with effect:

```
1  1  2  100  3
2  1  2  10  3
3  1  2  10  3
```

or (with the same effect):

```
1@3 1 2 &if n==1:v=100#if n>1:v=10& 3
```

or

```
1@3 &v=100*n& 3
```

```
1@3 1 2 3
```

The sign `;`—as separator of instruction in the same row–is quite complicated to be used within an `if` construct, so we use # as special sign, internally translated to `\n` (change row sign).

Details (NB, *a* and *b* are only used for the examples):

```
exec("a=2; if a< 3: print 'phew'") rises and error
exec("a=2\nif a< 3: print 'phew'") works
exec("a=2\n if a< 3: print 'phew'") rises and error again
```

The sign `;` is not forbidden, but we suggested to use it (carefully) to build blocks after an if structure; e.g.:

```
exec("a=2\nif a<2:b=11;print b\nelse: b=22;print b")
```

2.3.3 Future developments about agents

Currently, we have a unique Agent class for each project, containing heterogeneous methods addressed to different types of agents. The agent types are specified as in Section 2.3.1, so anything is working, but from the coding point of we this construction is a bit annoying.

Look at project "production" to see how we logically subdivide the unique class of the agents.

SLAPP 2.0 will introduce the possibility of having any number of classes for the agents, with internal modification: (a) in the AESOP mechanism (`read_script` execution), but also (b) at the higher level of Model scheduling, when a function like `askEachAgentInCollection` (of Tools.py) is internally used.

Chapter 3

Debugging a new project

TO BE DEVELOPED. Temporary have a look to the file `debug.txt` in the folder `debug` within the folder
`6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.

Chapter 4

Other existing and upcoming projects

4.1 Adding *turtles*: the *school* project

We add here turtle graphical capabilities. TO BE DEVELOPED.

About the name *(*turtle), have a look at Appendix D.

4.2 Adding networks: the *production* project

TO BE DEVELOPED.

4.3 New projects and extensions

4.3.1 Connecting to R, via Rserve

TO BE DEVELOPED.

4.3.2 Connecting to other applications, via Redis

TO BE DEVELOPED.

Redis is at `http://redis.io`.

We can—as an example—connect a SLAPP model to a NetLogo one (NetLogo address in Appendix D).

Chapter 5

SLAPP in IPython

5.1 Running SLAPP in an IPython notebook

SLAPP runs in IPython.

To use it as a notebook go to the main SLAPP folder via a terminal, then start:
`jupyter notebook`
and load in `jupyter` the file `iRunShell.ipynb`.

We do not explain here how to install Jupiter (https://jupyter.org), but a short cut is

```
pip install jupyter
```

or

```
sudo pip install jupyter
```

Look at the contents of Appendix A about the use of `pip`.

We also plan to have a SLAPP version running online.

5.2 *Turtle* graphics and IPython

Turtle graphics does not work in an IPython notebook (maybe in the future, existing several projects in that direction); the turtle display is generate outside the notebook. To run SLAPP on line, a possible solution is that of opening a VNC[25] connection parallel to the notebook interaction. More to come.

[25] https://en.wikipedia.org/wiki/Virtual_Network_Computing.

Appendices

Appendix A

Libraries for SLAPP

To use SLAPP you need to install a few Python libraries.

An easy way to have anything installed at once is the Anaconda Scientific Python distribution. You can find it at `https://store.continuum.io/cshop/anaconda/`, with clear installing instructions. Anaconda contains installers for Python 2.7 and 3.4: for SLAPP chose Python 2.7.

After the installation, your environment variable (PATH in Mac OS and Linux; PATH o path in Windows)[26] will contain the information to use Python and IPython from the `anaconda` folder (usually in the user home) and its subfolders.

If you do not want to use the Anaconda distribution, the *do it yourself* way is feasible.

[26]It is possible to see the content of the path from the *Terminal* (*Command Prompt* or *Windows PowerShell* in Windows) with:

echo $PATH
in Linux/Mac OS terminal

set path
in *Command Prompt* of Windows

$env:Path
in *Windows PowerShell* of Windows.

It is highly useful to familiarize with the Unix-like commands of the Linux/Mac OS Terminal and Windows PowerShell, e.g., at `https://en.m.wikipedia.org/w/index.php?title=Command-line_interface&redirect=no` and with the DOS-like commands of Command Prompt of Windows, e.g., at `http://pcsupport.about.com/od/termsc/p/command-prompt.htm`.

A.1 Using Linux (e.g., via the Ubuntu distribution)

- Verify the Python version in your system (with `python --version`) and upgrade it if not recent (in the series of the version 2.x at least 2.7.7; do not use version 3.x).

  A simple way to install Python from the terminal, is (sudo requires your password)
  `sudo apt-get update`
  to update the list of the packages, then
  `sudo apt-get install python`
  to upgrade (or to install, if python is not there)

- If the program `pip` (Python Package Index) is not installed (try `pip` in the terminal), run (always in the terminal)
  `sudo apt-get install python-pip`

- Install the `xlrd`[27] library to read spreadsheet files (`.xls` extension) in Python, via terminal with
  `sudo pip install xlrd`

- Until here, we have been copying the requirements of file `WARNING.txt` of the folder `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.

- The tools above are sufficient to run the `basic` example, having no graphic output, or the `school` project, which is entirely based on the graphical capabilities of the Python `turtle` library (installed with Python). About *turtles* see the Appendix D.

- If you want run the project `production`, graphically displaying networks, the reference is `WARNING bis - Production required libraries.txt` in the same folder above.

- Before installing `matplotlib`, it is useful to install `scipy` via Ubuntu Software Center; in this way you have also `numpy` installed (`numpy` is required by `matplotlib`).

- Install `matplotlib` (`http://matplotlib.org`) via Ubuntu Software Center.

- Install `NetworkX` (`https://networkx.github.io`) with
  `sudo pip install networkx`

[27] `https://github.com/python-excel/xlrd`

A.2 Using Mac OS X

- Verify the Python version in your system (with `python --version`) and upgrade it if not recent (in the series of the version 2.x at least 2.7.7; do not use version 3.x).

  To install Python download the `Mac OS X 64-bit/32-bit installer` from https://www.python.org; with the current 2.7.10 version, the installer file is `python-2.7.10-macosx10.6.pkg`. Run it (no security warning with OS X Yosemite).

- `pip` (Python Package Index) is coming with recent versions of Python; anyway, upgrade it via terminal with
  `sudo pip install --upgrade pip`
  (root user password required).

- Install the `xlrd`[28] library to read spreadsheet files (`.xls` extension) in Python, via terminal with
  `sudo pip install xlrd`
  (root user password required).

- Until here, we have been copying the requirements of file `WARNING.txt` of the folder `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.

- The tools above are sufficient to run the `basic` example, having no graphic output, or the `school` project, which is entirely based on the graphical capabilities of the Python `turtle` library (installed with Python). About *turtles* see the Appendix D.

- If you want run the project `production`, graphically displaying networks, the reference is `WARNING bis - Production required libraries.txt` in the same folder above.

- Before installing `matplotlib`, it is useful to install `scipy`, via terminal with:
  `sudo pip install scipy`
  (root user password required). In this way you have also `numpy` installed (`numpy` is required by `matplotlib`).

- Install `matplotlib` (http://matplotlib.org), via terminal with
  `sudo pip install matplotlib`
  (root user password required).

[28] https://github.com/python-excel/xlrd

- Install NetworkX (https://networkx.github.io) with
  `sudo pip install networkx`
  (root user password required).

A.3 Using Windows (referring to Windows 10)

We refer here to Windows 10, but the following notes work also for the versions 7, 8, 8.1 (always supposing a 64 bits system).

Use the `Command Prompt` or the `Windows PowerShell`, introduced in note 26 above.

- Python 2.7.x, if installed, is in `C:\Python27\`
- Verify the Python version in your system (with `python --version`) and upgrade it if not recent (in the series of the version 2.x at least 2.7.7; do not use version 3.x).

  From https://www.python.org/ you can download an installer; e.g. for version 2.7.10 on a 64 bits system: `python-2.7.10.amd64.msi`. Run the file clicking on it.

  If you run `python` from a terminal (`Command Prompt` or `Windows PowerShell`), the reply is that the program does not exist. You have to run

  ```
  \python27\python
  ```

  because the *path* of your system does not contemplate that folder as a repository for programs. As an example, it could be (using `Command Prompt`; for `Windows PowerShell` use `$env:Path` instead of `set path`):

  ```
  >set path
  Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
  C:\Windows\System32\WindowsPowerShell\v1.0\
  ```

  You have to modify the *environment variables*: from `Settings` go to `System`, then to `About`, scroll down to find `System Info`, then proceed choosing `Advanced System Settings`, press the `Environment Variables` button. In `System variables`, chose `Path`, then `Edit` and add at the end of the path the string (pay attention to the initial semicolon):

  `;c:\Python27\;c:\Python27\Scripts\`.

  Restart the terminal you were using to apply the new settings and the `python` command will work.

  Now you have:

```
>set path
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
C:\Windows\System32\WindowsPowerShell\v1.0\;c:\Python27\;
c:\Python27\Scripts\
```

- `pip` (Python Package Index) is coming with recent versions of Python; anyway, upgrade it via terminal with
  `pip install --upgrade pip`
- Install the `xlrd`[29] library to read spreadsheet files (`.xls` extension) in Python, via terminal with
  `pip install xlrd`
- Until here, we have been copying the requirements of file `WARNING.txt` of the folder `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.
- The tools above are sufficient to run the `basic` example, having no graphic output, or the `school` project, which is entirely based on the graphical capabilities of the Python `turtle` library (installed with Python). About *turtles* see the Appendix D.
- If you want run the project `production`, graphically displaying networks, the reference is `WARNING bis - Production required libraries.txt` in the same folder above.
- Before installing `matplotlib`, it is useful to install `scipy`, via terminal with:[30]
  `pip install scipy`
  In this way you have also `numpy` installed (`numpy` is required by `matplotlib`).

  NB, the above suggestion has to be fixed, so temporary install directly `numpy` (see note 30):
  `pip install numpy`
- Install `matplotlib` (http://matplotlib.org), via terminal with
  `pip install matplotlib`
- Install NetworkX (https://networkx.github.io) with
  `pip install networkx`

[29] https://github.com/python-excel/xlrd

[30] in case of an error signalling the file *vcvarsall.bat* as missing, run VCForPython27.msi from http://www.microsoft.com/en-us/download/details.aspx?id=44266. It is the Microsoft Visual C++ Compiler for Python 2.7.

Appendix B

On SLAPP execution

SLAPP runs only via a terminal or in IPython (`jupyter notebook`), using `runShell.py` or `iRunShell.ipynb`.

In IPython, the magic command `'%matplotlib inline'` is internally added if missing; if `'%matplotlib'` is the explicit choice, the `'inline'` option is internally stated.

In the main folder now we have `runShell.py` to start the shell in Python and `iRunShell.ipynb` to start it in IPython (using `jupyter notebook`).

We do not explain here how to install Jupiter (https://jupyter.org), but a short cut is

```
pip install jupyter
```

or

```
sudo pip install jupyter
```

Look at the contents of Appendix A about the use of `pip`.

We stop the execution if starting from IDLE or Spyder, for compatibility with the graphic operations.[31]

[31] At http://matplotlib.org/users/shell.html we read "the python IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend".

Appendix C

Problems with libraries

C.1 A temporary problem with *matplotlib* 1.3.4

With the version of 1.3.4 of `matplotlib`, we have an annoying warning message when producing graphics. The warning (the path is local to the running machine):

We have a patch into the main folder of version 1.2 of SLAPP (look for it in the branch v.1.2 at https://github.com/terna/SLAPP).

C.2 A warning about fonts coming from *matplotlib* 1.5.1

`matplotlib` produces an annoying warning about creating fonts; to avoid it.

Several hints online suggest to delete the folders fontconfig or matplotlib that you can find in folder `.cache` within your home.

Instead, in MacOS go to the folder `.matplolib` in your home and delete the file fontList.cache.

The annoying warning will appear only one more time.

Appendix D

On *turtles*

The `turtle` library mimics the behavior both of NetLogo,[32] of OpenStarLogo,[33] and (partially) of StarLogo TNG[34] agent-based shells. The name *turtle* attributed to the agents in those shells (and in the Python related library) comes from Logo, a special language of the 1960s. At http://el.media.mit.edu/logo-foundation/what_is_logo/logo_primer.html we read that:

> The most popular Logo environment has involved the Turtle, originally a robotic creature that moved around on the floor.
>
> It can be directed by typing commands at the computer. The command forward 100 causes the turtle to move forward in a straight line 100 "turtle steps". Right 45 rotates the turtle 45 degrees clockwise while leaving it in the same place on the floor. Then forward 50 causes it to go forward 50 steps in the new direction.
>
> With just the two commands forward and right, the turtle can be moved in any path across the floor. The turtle also has a pen which may be lowered to the floor so that a trace is left of where it has traveled. With the pen down, the turtle can draw geometric shapes, and pictures, and designs of all sorts.
>
> (. . .)
>
> The turtle migrated to the computer screen where it lives as a graphics object. Viewing the screen is like looking down on the mechanical turtle from above.

But . . . why the name *turtle*? In Epstein (2014, p.88) we have a nice and openly subjective explanation:

[32] https://ccl.northwestern.edu/netlogo/
[33] http://web.mit.edu/mitstep/openstarlogo/index.html
[34] http://education.mit.edu/portfolio_page/starlogo-tng/

> *NetLogo*'s name for a generic agent is "turtle". I choose to imagine that this is in honor of a famous exchange between Bertrand Russel and an audience member who told Russel that the earth was supported on the nack of a great turtle. Russel asked, 'And what, pray tell, is supporting *that* turtle?' The answer was immediate. "Oh, another turtle . . . it's turtles all the way down."

My humble explanation is less fashinating: when I was told about Logo for the first time, in the 1970s, they explained me that the agent was a turtle . . . because it was slowly moving.

Anyway, what is crucial is that NetLogo and StartLogo TNG (deriving from what now is named OpenStarLogo) have their roots in Logo and turtles (in Fig. D.1 the *logo* of the Logo Foundation).

Figure D.1: The Logo Foundation, at
http://el.media.mit.edu/logo-foundation/

Bibliography

Boero, R., Morini, M., Sonnessa, M. and Terna, P. (2015). *Agent-based Models of the Economy Agent-based Models of the Economy – From Theories to Applications*. Palgrave Macmillan, Houndmills.
URL http://www.palgrave.com/page/detail/agentbased-models-of-the-economy-/?K=9781137339805

Downey, A. B. (2012). *Think Python*. How to Think Like a Computer Scientist. O'Reilly Media, Inc., Sebastopol, CA.
URL http://www.greenteapress.com/thinkpython/

Elkner, J., Downey, A. B. and Meyers, C. (2013). *Learning with Python: Interactive Edition 2.0*. How to Think Like a Computer Scientist. Runestone Interactive.
URL http://interactivepython.org/runestone/default/user/login?_next=/runestone/default/index

Epstein, J. M. (2014). *Agent_Zero: Toward Neurocognitive Foundations for Generative Social Science*. Princeton University Press.

Minar, N., Burkhart, R., Langton, C. and Askenazi, M. (1996). *The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations*. In «SFI Working Paper», vol. 06(42).
URL http://www.santafe.edu/media/workingpapers/96-06-042.pdf

Sargent, T. and Stachurski, J. (2013). *Quantitative Economics*.
URL http://quant-econ.net

Index