

# Oligopoly

Marco Mazzoli, Matteo Morini, and Pietro Terna

January 9, 2016

# Contents

<b>1</b>	<b>The <i>oligopoly</i> project: the making of the simulation model</b>	<b>3</b>
1.1	The agents and their sets . . . . .	5
1.1.1	Sets of agents . . . . .	7
1.2	Scheduling . . . . .	8
1.2.1	The scheduling mechanism at the level of the Observer . . . .	8
1.2.2	The scheduling mechanism at the level of the Model . . . . .	10
1.2.3	The detailed scheduling mechanism within the Model (AE-SOP level) . . . . .	11
1.2.4	Model versions via the AESOP level in scheduling . . . . .	12
1.2.5	The items of our AESOP level in scheduling . . . . .	13
1.2.6	Other features in scheduling [NB the notes with the \$\$ mark have to be reported in the Handbook and require code modification] . . . . .	23
	<b>Bibliography</b>	<b>25</b>
	<b>Index</b>	<b>26</b>

## List of Figures

1.1	The representation of the schedule . . . . .	5
1.2	Time series generated by the model . . . . .	9
1.3	The agents (nodes), with random displacements, and links connecting entrepreneurs and workers . . . . .	9

# Chapter 1

## The *oligopoly* project: the making of the simulation model

Using SLAPP<sup>1</sup>, the *oligopoly* project is contained in a stand alone folder, having the same name of the model.

Let us introduce the starting phase in a detailed way.

- In the SLAPP distribution, we have the folder  
6 objectSwarmObserverAgents\\_AESOP\\_turtleLib\\_NetworkX} folder,  
being the starting point of the simulator engine.
  - We launch SLAPP, via the `start.py` file that we find in the folder of  
SLAPP as a simulation shell, i.e.  
6 objectSwarmObserverAgents\\_AESOP\\_turtleLib\\_NetworkX, from a  
terminal, with:  
`python start.py`
  - Alternatively, we can launch SLAPP via the `runShell.py` file that we  
find in the main folder of SLAPP, from a terminal, with:  
`python runShell.py`

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

- We can predefining a default project: if we place *in the main SLAPP folder or  
in the folder* 6 objectSwarmObserverAgents\\_AESOP\\_turtleLib\\_NetworkX

---

<sup>1</sup><https://github.com/terna/SLAPP>; SLAPP has a Reference Handbook at the same address and it is deeply described in Chapters 2–7 in Boero *et al.* (2015).

a file named `project.txt` containing the path to a folder (`oligopoly` in our case, with `/Users/pt/GitHub/oligopoly`, as an example of location), the initial message of SLAPP is:

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

- Resuming the explanation, we continue receiving the messages:

```
running in Python
debug = False
random number seed (1 to get it from the clock)
```

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 replicate 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 while debugging, when we need to repeat exactly the sequence generating errors, but also 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`. About running SLAPP in IPython have a look the the Handbook, in the SLAPP web site.<sup>2</sup>

- The program sends several messages about the project parameters, as specified into the file `commonVar.py` and managed via the file `parameters.py`, both in the project folder.

One of these messages reports the version of the project.

- The program informs us about the «sigma of the normal distribution used in randomizing the position of the agents/nodes», e.g., 0.7; this is uniquely a graphic effect, as in Figure 1.3.
- We introduce now time management, split into several (consistent) levels of scheduling.

---

<sup>2</sup><https://github.com/terna/SLAPP>.

The general picture is that of Figure 1.1: in an abstract way we can imagine having a clock opening a series of containers or boxes. Behind the boxes, we have the *action groups*, where we store the information about the actions to be done.<sup>3</sup>

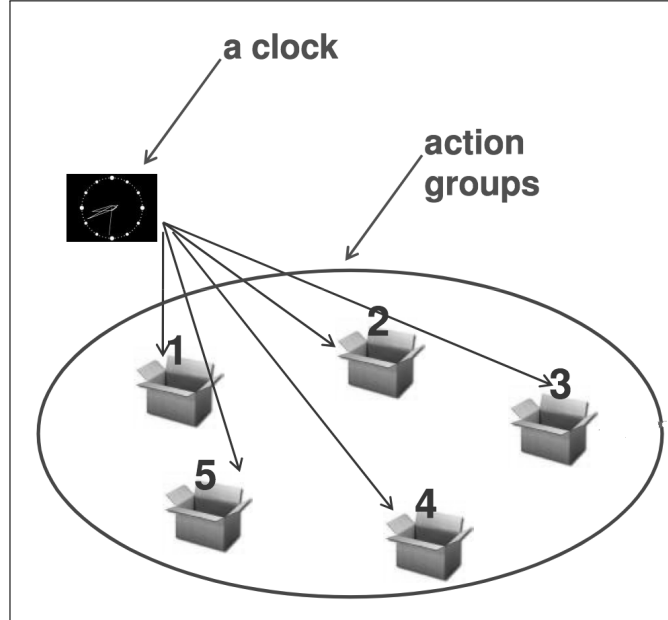


Figure 1.1: The representation of the schedule

## 1.1 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 `entrepreneurs workers`.

- `entrepreneurs.txt` lists the agents of type `entrepreneurs`; It reports the identification numbers (currently from 1 to 5) and the  $x$  and  $y$  positions on the screen. See above the *sigma* value determining random shift from the stated positions; in this way, we can attribute close or equal positions to several entrepreneurs having them anyway visible in the map; if necessary, we can increase *sigma*:

<sup>3</sup>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.

```
1  -10 75
2  -10 65
3  -10 55
4  -10 45
5  -10 35
```

- `workers.txt` lists the agents of type `workers`. It reports the identification numbers (currently from 1 to 20) and the  $x$  and  $y$  positions on the screen. See above the  $\sigma$  value determining random shift from the stated positions; in this way, we can attribute close or equal positions to several entrepreneurs having them anyway visible in the map; if necessary, we can increase  $\sigma$ :

```
1   10 105
2   10 100
3   10  90
4   10  85
5   10  80
6   10  75
7   10  70
8   10  65
9   10  60
10  10  55
11  10  50
12  10  45
13  10  40
14  10  35
15  10  30
16  10  25
17  10  20
18  10  15
19  10  10
20  10   5
```

The agents are created by `ModelSwarm.py` (in folder `$$$lapp$$$`) via the specific rules contained into the file `mActions.py`, specific for this project (indeed, it is into the folder `oligopoly`).

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

    # workers
    if agType=="workers":
        anAgent = Agent(num, self.worldStateList[0],
```

```
float(line.split()[1])+random.gauss(0,common.sigma),
float(line.split()[2])+random.gauss(0,common.sigma),
agType=agType)
self.agentList.append(anAgent)
anAgent.setAgentList(self.agentList)

# entrepreneurs
elif agType=="entrepreneurs":
    anAgent = Agent(num, self.worldStateList[0],
                    float(line.split()[1])+random.gauss(0,common.sigma),
                    float(line.split()[2])+random.gauss(0,common.sigma),
                    agType=agType)
    self.agentList.append(anAgent)
    anAgent.setAgentList(self.agentList)

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

The following bullets describe how this code works.

- The number identifying the agent is read outside this function, as a mandatory first element in each line into a file containing agent descriptions. The content of the `agType` variable is directly the name of the agent file currently open.
- We check the input file, which has to contain three data per row. We modify the second and the third values with the *sigma* correction.

Each agent is added to the `agentList`.

### 1.1.1 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 `oligopoly` we have the files `entrepreneurs.txt` and `workers.txt`;
- files defining groups of agents:
  - the list of the types of agents (mandatory); from this list SLAPP searches the file describing the agents; as seen, in folder `oligopoly` we have the file `agTypeFile.txt` (the name of this file is mandatory) containing:

```
entrepreneurs workers
```



- the list of the operating sets of agents (optional); in folder `oligopoly` this file is missing. Indeed we receive the message

`Warning: operating sets not found.`

In the file `ag0operatingSets.txt` (the name of this file is mandatory), with could place names of groups of agents, corresponding to files listing the agents in the group. Project verb "school" can be used as a useful example.

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 `schedule.xls` file.

## 1.2 Scheduling

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

- Two of those mechanisms are one at the level of the Observer and the other of the Model, with recurrent sequences of action to be done.<sup>4</sup>
- In our `oligopoly` code, these two sequences are reported in the files `observerActions.txt` and `modelActions.txt` in the folder of the project.

The explanations are in Section 1.2.1 and 1.2.2.

- The third sequence is the more detailed one (see Section 1.2.3).

### 1.2.1 The scheduling mechanism at the level of the Observer

- The first schedule mechanism is described in the first file (`observerActions.txt`), having content (unique row, remembering that anyway row changes are not relevant to this group of files):

- version *without pauses* contained in `observerActions no pause.txt`, to be copied to `observerActions.txt` to run it:

```
modelStep visualizePlot visualizeNet clock
```

---

<sup>4</sup>The level of the Observer is our level, where the experimenter looks at the model (the level of the Model) while it runs.

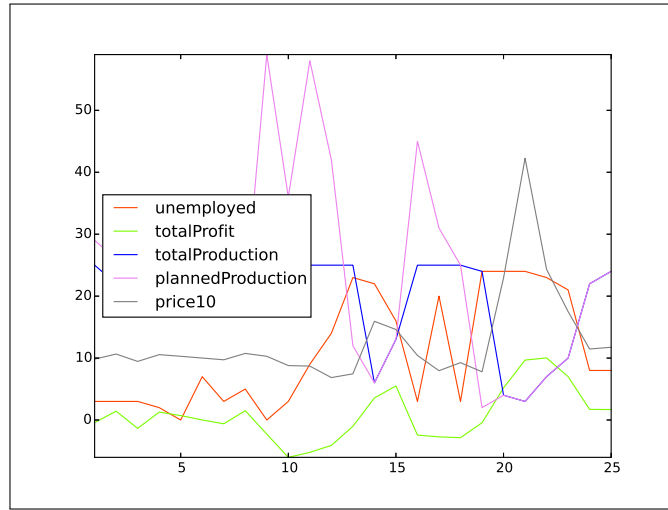


Figure 1.2: Time series generated by the model

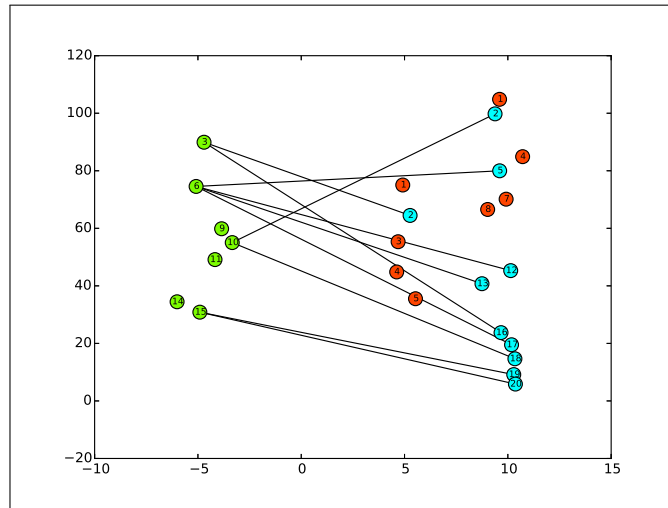



Figure 1.3: The agents (nodes), with random displacements, and links connecting entrepreneurs and workers

- version *with pauses* contained in `observerActions` with `pause.txt`, to be copied to `observerActions.txt` to run it:

```
modelStep visualizePlot visualizeNet pause 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).
- `modelStep` orders to the model to make a step forward in time.
- `visualizePlot` update the plot of the time series generated by the model (Figure 1.2).
- `visualizeNet` update the windows reporting the links connecting entrepreneurs and workers, on a network basis (Figure 1.3).
- `pause`, if any, puts the program in wait until we reply to the message Hit `enter` key to `continue`, hitting the key . This action is useful to examine the graphical outputs (as in Figures 1.2 and 1.3), step by step.
- `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 internal scheduler of the Observer 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.
- We also have a potential `prune` item, eliminating the links on the basis of their weight (in case, asking for a threshold below which we cut); weights could be introduced to measure the seniority (skill, experience) of the workers.

### 1.2.2 The scheduling mechanism at the level of the Model

- The second file—`modelActions.txt`—quoted above at the beginning of Section 1.2, is related to the second of the schedule mechanisms, i.e., that of the Model. About the Observer/Model dualism, the reference is to note 4.

It contains (unique row, remembering that anyway row changes are not relevant to this group of files):

`reset 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, but operating at its level.
- `reset` orders to the agents to make a reset, related to their variables. The order acts via the code in the file `ModelSwarm.py`.<sup>5</sup> `reset` contains the `do0` variable, linking a method that is specified as a function in the file `mActions.py` in the folder of the project. In this way, the application of the basic method `reset` can be flexibly tailored to the specific applications, defining which variables to reset.

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 redefine it in `Agent.py` in the project folder.

In our model, we clean the variables `totalProductionInA_TimeStep` and `totalPlannedConsumptionInValueInA_TimeStep` at the beginning of each step of the time. The code, in `Agent.py` is:

```
# reset values, redefining the method of agTools.py in $$$slapp$$
def setNewCycleValues(self):
    common.totalProductionInA_TimeStep=0
    common.totalPlannedConsumptionInValueInA_TimeStep=0
```

- `read_script` orders to the Model to open a new level of scheduling, described in Section 1.2.3. The order acts via the code of the file `ModelSwarm.py`. We have here one of the stable instances 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).

### 1.2.3 The detailed scheduling mechanism within the Model (AESOP level)

.

---

<sup>5</sup>That is in the `$$$slapp$$` folder.

*AESOP* comes from Agents and Emergencies for Simulating Organizations in Python.

- The third scheduling mechanism, as anticipated in Section 1.2, 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.

- Now we take in exam the timetable of our Oligopoly model.

- 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). 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 differentiate the execution sequences in our model via the `schedule.xls` sheet contained in the folder `oligopoly`.

Within the sheet, we have the action containers as introduced above (Figure 1.1), starting with the sign `#`.

#### 1.2.4 Model versions via the AESOP level in scheduling

We have several versions of the model defined via the sequences of actions. To use one of them, we have to copy its schedule to the basic `schedule.xls` file.

**Version 0, preliminary step (GitHub, master).** In `schedule0.xls` (to be copied to `schedule.xls` for the use) we have (comments start at column E and are missing) three columns:

#	1	100
entrepreneurs	produce	
entrepreneurs	evaluateProfitV0	
entrepreneurs	0.5	hireIfProfit
entrepreneurs	0.5	fireIfProfit

**Version 1, Random production as engine (GitHub, V1).** In `schedule1.xls` (to be copied to `schedule.xls` for the use) we have (comments start at column E and are missing) three columns:

#	1	100
entrepreneurs	makeProductionPlan	
entrepreneurs	hireFireWithProduction	
entrepreneurs	produce	
WorldState	specialUse	setMarketPriceV1
entrepreneurs	evaluateProfit	
entrepreneurs	0.5	fireIfProfit

**Version 2 (GitHub, V2).** Here we have (i) random production as engine, (ii) individual demand curves with more realistic price determination, (iii) new entrant firms.

In `schedule2.xls` (to be copied to `schedule.xls` for the use) we have (comments start at column E and are missing) three columns:

#	1	100
entrepreneurs	makeProductionPlan	
entrepreneurs	hireFireWithProduction	
entrepreneurs	produce	
entrepreneurs	planConsumptionInValue	
workers	planConsumptionInValue	
WorldState	specialUse	setMarketPriceV2
entrepreneurs	evaluateProfit	
entrepreneurs	0,5	fireIfProfit
workers	toEntrepreneur	
entrepreneurs	toWorker	

### 1.2.5 The items of our AESOP level in scheduling

We have several items, not all used in each version of the model.

- # 1 100 fills 100 steps of the time schedule (or any other number of them) with the sequence below it, creating 100 (in this case) time containers.  
The actual step repetition upon time can be  $\leq 100$ ; if  $> 100$  the steps after the 100<sup>th</sup> will be lacking of activity of the detailed scheduling activity (AESOP layer).

#### Methods used in Version 0

- The method (or command) `evaluateProfitV0`<sup>6</sup> sent to the `entrepreneurs` order them to calculate their profit. Being  $P_t^i$  the production and  $\pi$  the labor productivity, we have the labor force  $L_t^i = P_t^i / \pi$   
 $R$  is `revenuesOfSalesForEachWorker`, set to 1.005 in `common` variable space, not changing with  $t$ ;  $w$  is the `wage` per employee and time unit, set to 1.0

---

<sup>6</sup>Related to Version 0.

in common variable space, not changing with  $t$ .  $u_t^i \sim \mathcal{N}(0, 0.05)$  is a random normal addendum.

The profit evaluation is:

$$\Pi_t^i = L_t^i(R - w) + u_t^i \quad (1.1)$$

The code is:

```
# calculateProfit
def evaluateProfitV0(self):

    # this is an entrepreneur action
    if self.agType == "workers": return

    # the number of prducing workers is obtained indirectly via
    # production/laborProductivity
    #print self.production/common.laborProductivity
    self.profit=(self.production/common.laborProductivity) * \
        (common.revenuesOfSalesForEachWorker - \
         common.wage) + gauss(0,0.05)
```

- The method (or command) **hireIfProfit**<sup>7</sup> sent to the **entrepreneurs** order them—in a probabilistic way (50% of probability in Version 0 case), in each unit of time—to hire a worker (random choosing her/him in a temporary list of unemployed people) if the profit (last calculation, i.e., current period as shown in the sequence contained in **schedule.xls**) is greater than the value **hiringThreshold** (temporary: 0):

$$\Pi_t^i > \text{hiringThreshold} \rightarrow \text{hire} \quad (1.2)$$

As first attempt the **hiringThreshold** is 0 (in **commonVar.py**). We can modify this internal value, as others, while the simulation is running, via the *WorldState* feature, introduced below.

The code of the **hireIfProfit** method is:

```
# hireIfProfit
def hireIfProfit(self):

    # workers do not hire
    if self.agType == "workers": return

    if self.profit<=common.hiringThreshold: return

    tmpList=[]
    for ag in self.agentList:
        if ag != self:
            if ag.agType=="workers" and not ag.employed:
                tmpList.append(ag)
```

---

<sup>7</sup>Used in Version 0.

```

if len(tmpList) > 0:
    hired=tmpList[randint(0,len(tmpList)-1)]

    hired.employed=True
    gvf.colors[hired]="Aqua"
    gvf.createEdge(self, hired) #self, here, is the hiring firm

# count edges (workers) of the firm, after hiring (the values is
# recorded, but not used directly)
self.numOfWorkers=gvf.nx.degree(common.g, nbunch=self)
# nbunch : iterable container, optional (default=all nodes)
# A container of nodes. The container will be iterated through once.
print "entrepreneur", self.number, "has", \
    self.numOfWorkers, "edge/s after hiring"

```

## Methods used in Versions 0, 1, 2

- The method (or command) **produce**<sup>8</sup> sent to the **entrepreneurs** order them—in a deterministic way, in each unit of time—to produce proportionally to their labour force, obtaining profit  $\Pi_t^i$ , where  $i$  identifies the firm and  $t$  the time.

$L_t^i$  is the number of workers in firm  $i$  at time  $t$ , and also the number of its links. We add 1 to  $L_t^i$ , to account for the entrepreneur as a worker.  $\pi$  is the **laborProductivity**, with its value set to 1 in **common** variable space, currently not changing with  $t$ .  $P_t^i$  is the production of firm  $i$  at time  $t$ .

The production is:

$$P_t^i = \pi(L_t^i + 1) \quad (1.3)$$

The production of the  $i^{\text{th}}$  firm is added to the total production of the time step, in the variable **totalProductionInA\_TimeStep** of the *common* space.

The code is:

```

# produce
def produce(self):

    # this is an entrepreneur action
    if self.agType == "workers": return

    # to produce we need to know the number of employees
    # the value is calculatated on the fly, to be sure of accounting for
    # modifications coming from outside
    # (nbunch : iterable container, optional (default=all nodes)
    # A container of nodes. The container will be iterated through once.)

    laborForce=gvf.nx.degree(common.g, nbunch=self) + \
        1 # +1 to account for the entrepreneur itself

```

---

<sup>8</sup>Related to Versions 0, 1, 2.



```
# productivity is set to 1 in the beginning
self.production = common.laborProductivity * \
    laborForce

# totalProductionInA_TimeStep
common.totalProductionInA_TimeStep += self.production
```

We calculate the `laborForce`, i.e.  $L_t^i$ , counting the number of links or edges from the firm to the workers. We prefer this ‘on the fly’ evaluation to the internal variable `self.numOfWorkers`, to be absolutely sure of accessing the last datum in case of modifications coming from other procedures. E.g., a random subtraction or addition of workers to firms coming simulating some kind of shock ...

- The method (or command) `fireIfProfit`<sup>9</sup> sent to the `entrepreneurs` order them—in a probabilistic way (50% of probability in Version 0 and 1 cases), in each unit of time—to fire a worker (random choosing her/him in the list of the employees of the firm) if the profit (last calculation, i.e., current period as shown in the sequence contained in `schedule.xls`) is less than the value `firingThreshold` (temporary: 0):

$$\Pi_t^i < firingThreshold \rightarrow fire \quad (1.4)$$

```
# fireIfProfit
def fireIfProfit(self):

    # workers do not fire
    if self.agType == "workers": return

    if self.profit >= common.firingThreshold: return

    # the list of the employees of the firm
    entrepreneurWorkers = gvf.nx.neighbors(common.g, self)
    # print "entrepreneur", self.number, "could fire", entrepreneurWorkers

    if len(entrepreneurWorkers) > 0:
        fired = entrepreneurWorkers[randint(0, len(entrepreneurWorkers)-1)]

        gvf.colors[fired] = "OrangeRed"
        fired.employed = False

        common.g_edge_labels.pop((self, fired))
        common.g.remove_edge(self, fired)

    # count edges (workers) after firing (recorded, but not used
    # directly)
    self.numOfWorkers = gvf.nx.degree(common.g, nbunch=self)
    # nbunch : iterable container, optional (default=all nodes)
    # A container of nodes. The container will be iterated through once.
    print "entrepreneur", self.number, "has", \
        self.numOfWorkers, "edge/s after firing"
```

---

<sup>9</sup>Used in Versions 0, 1, (temporaty) 2.

### Methods used in Version 1

- The method (or command) `setMarketPriceV1`,<sup>10</sup> sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 1.2.6.

### Methods used in Versions 1, 2

- The method (or command) `makeProductionPlan`<sup>11</sup> sent to the `entrepreneurs` order them to guess their production for the current period. The production plan  $\hat{P}_t^i$  is determined in a random way, using a Poisson distribution, with  $\lambda = 5$  as mean (suggested value kept in the *common* space).

As a definition, the production plan is:

$$\hat{P}_t^i \sim \text{Pois}(\lambda) \quad (1.5)$$

We suggest temporary a value of 5 for  $\lambda$ , due to the quantities: entrepreneurs 5, workers 20 + the 5 entrepreneurs, labor productivity 1. (The value of  $\lambda$  can be modified in the prologue of the run).

The code is:

```
# makeProductionPlan
def makeProductionPlan(self):

    # this is an entrepreneur action
    if self.agType == "workers": return

    self.plannedProduction=npr.poisson(5,1)[0] # 1 is the number
    # of element of the returned matrix (vector)
```

- The method (or command) `evaluateProfit`<sup>12</sup> sent to the `entrepreneurs` order them to calculate their profit. Being  $P_t^i$  the production and  $\pi$  the labor productivity, we have the labor force  $L_t^i = P_t^i / \pi$

The method has been improved in versione 2, to manage extra costs for the new entrant firms, but keeping safe the backward compatibility of the method.

$p_t$  is the **price**, clearing the market at time  $t$  and it calculated by the abstract item `WorldState` via the method `setMarketPrice`, as explained in Section 1.2.6.

---

<sup>10</sup>Related to Version 1.

<sup>11</sup>Related to Versions 1, 2.

<sup>12</sup>Related to Versions 1, 2.

$w$  is the **wage** per employee and time unit, set to 1.0 in **common** variable space, not changing with  $t$ .  $C$  are extra costs for new entrant firms.

The profit evaluation is:

$$\Pi_t^i = p_t P_t^i - w L_t^i - C \quad (1.6)$$

The new entrant firms have extra costs to be supported, retrieved in **XC** variables, but only for  $k$  periods, as stated in **commonVar.py** and activated by method to **toEntrepreneur**.

The code is:

```
# calculateProfit
def evaluateProfit(self):

# this is an entrepreneur action
    if self.agType == "workers": return

    # backward compatibily to version 1
    try: XC=self.newEntrantExtraCosts
    except: XC=0
    try: k=self.extraCostsResidualDuration
    except: k=0

    if k==0: XC=0
    if k>0: self.extraCostsResidualDuration-=1

    # the number of pruding workers is obtained indirectly via
    # production/laborProductivity
    #print self.production/common.laborProductivity
    self.profit=common.price * self.production - \
        common.wage * (self.production/common.laborProductivity) - \
        XC
```

- The method (or command) **hireFireWithProduction**<sup>13</sup> sent to the **entrepreneurs** order them to hire or fire comparing the labor forces required for the production plan  $\hat{P}_t^i$  and the labor productivity  $\pi$ ; we have the required labor force ( $L_t^i$  is the current one):

$$\hat{L}_t^i = \hat{P}_t^i / \pi \quad (1.7)$$

Now:

1. if  $\hat{L}_t^i = L_t^i$  nothing has to be done;
2. if  $\hat{L}_t^i > L_t^i$ , the entrepreneur is hiring with the limit of the number of unemployed workers;
3. if  $\hat{L}_t^i < L_t^i$ , the entrepreneur is firing the workers in excess.

---

<sup>13</sup>Related to Versions 1, 2.

The code is:

```
def hireFireWithProduction(self):

    # workers do not hire/fire
    if self.agType == "workers": return

    # to decide to hire/fire we need to know the number of employees
    # the value is calculated on the fly, to be sure of accounting for
    # modifications coming from outside
    # (nbunch : iterable container, optional (default=all nodes)
    # A container of nodes. The container will be iterated through once.)

    laborForce0=gvf.nx.degree(common.g, nbunch=self) + \
        1 # +1 to account for the entrepreneur itself

    # required labor force
    laborForceRequired=int(
        self.plannedProduction/common.laborProductivity)

    # no action
    if laborForce0 == laborForceRequired: return

    # hire
    if laborForce0 < laborForceRequired:
        n = laborForceRequired - laborForce0
        tmpList=[]
        for ag in self.agentList:
            if ag != self:
                if ag.agType=="workers" and not ag.employed:
                    tmpList.append(ag)

        if len(tmpList) > 0:
            k = min(n, len(tmpList))
            shuffle(tmpList)
            for i in range(k):
                hired=tmpList[i]
                hired.employed=True
                gvf.colors[hired]="Aqua"
                gvf.createEdge(self, hired)
                #self, here, is the hiring firm

        # count edges (workers) of the firm, after hiring (the values is
        # recorded, but not used directly)
        self.numOfWorkers=gvf.nx.degree(common.g, nbunch=self)
        # nbunch : iterable container, optional (default=all nodes)
        # A container of nodes. The container will be iterated through once.
        print "entrepreneur", self.number, "has", \
            self.numOfWorkers, "edge/s after hiring"

    # fire
    if laborForce0 > laborForceRequired:
        n = laborForce0 - laborForceRequired

        # the list of the employees of the firm
        entrepreneurWorkers=gvf.nx.neighbors(common.g,self)
        #print "entrepreneur", self.number, "could fire", entrepreneurWorkers

        if len(entrepreneurWorkers) > 0: # has to be, but ...
            shuffle(entrepreneurWorkers)
            for i in range(n):
                fired=entrepreneurWorkers[i]
```

```

gvf.colors[fired]="OrangeRed"
fired.employed=False

common.g_edge_labels.pop((self,fired))
common.g.remove_edge(self, fired)

# count edges (workers) after firing (recorded, but not used
# directly)
self.numOfWorkers=gvf.nx.degree(common.g, nbunch=self)
# nbunch : iterable container, optional (default=all nodes)
# A container of nodes. The container will be iterated through once.
print "entrepreneur", self.number, "has", \
      self.numOfWorkers, "edge/s after firing"

```

## Methods used in Version 2

- The method (or command) `planConsumptionInValue`,<sup>14</sup> sent to **entrepreneurs** or **workers**, produces the following evaluations, detailed in `commonVar.py` file.

Consumption behavior with

$$C_i = a_i + b_i Y_i + u \quad (1.8)$$

with  $u \sim \mathcal{N}(0, common.consumptionRandomComponentSD)$

$i$  can be: "(1) entrepreneurs as consumers, with  $Y_1 = profit_{t-1} + wage$ ; (2) employed workers, with  $Y_2 = wage$ ; (3) unemployed workers, with  $Y_3 = socialWelfareCompensation$ ."

The  $a_i$  and  $b_i$  values are set via the file `commonVar.py` and reported in output, when the program starts, via the `parameters.py`.

The code in `Agent.py` is:

```

# compensation
def planConsumptionInValue(self):
    self.consumption=0
    #case (1)
    #Y1=profit(t-1)+wage NB no negative consumption if profit(t-1) < 0
    # this is an entrepreneur action
    if self.agType == "entrepreneurs":
        self.consumption = common.a1 + \
                           common.b1 * (self.profit + common.wage) + \
                           gauss(0, common.consumptionRandomComponentSD)
        if self.consumption < 0: self.consumption=0
        #profit, in V2, is at time -1 due to the sequence in schedule2.xls

    #case (2)
    #Y2=wage

```

---

<sup>14</sup>Related to Version 2.

---

```

if self.agType == "workers" and self.employed:
    self.consumption = common.a2 + \
        common.b2 * common.wage + \
        gauss(0, common.consumptionRandomComponentSD)

#case (3)
#Y3=socialWelfareCompensation
if self.agType == "workers" and not self.employed:
    self.consumption = common.a3 + \
        common.b3 * common.socialWelfareCompensation + \
        gauss(0, common.consumptionRandomComponentSD)

#update totalPlannedConsumptionInValueInA_TimeStep
common.totalPlannedConsumptionInValueInA_TimeStep+=self.consumption
#print "C sum", common.totalPlannedConsumptionInValueInA_TimeStep

```

The conclusion updates the *common* value—cleaned at each reset, i.e., at each time step in `modelActions.txt`—of `totalPlannedConsumptionInValueInA_TimeStep`

- The method (or command) `setMarketPriceV2`,<sup>15</sup> sent to the `WorldState`, orders it to evaluate the market clearing price. This method uses two common variables:
  - `totalProductionInA\_TimeStep`, generated by the agents (*entrepreneurs*), via `produce`;
  - `totalPlannedConsumptionInValueInA\_TimeStep`, generated by the agents (*entrepreneurs* and *workers*) via `planConsumptionInValue`.

See below the Section 1.2.6.

- With the method (or command) `toEntrepreneur`,<sup>16</sup> sent to *workers*, the agent, being a worker, decides if to became an entrepreneur at time  $t$ , if its employer has a profit  $\geq$  a given *threshold* in  $t$ . The threshold is retrieved from the variable `thresholdToEntrepreneur`.

The agent changes its internal type, position (not completely at the left as the original entrepreneurs, but if it was an entrepreneur moved to worker and coming back, it goes completely at the left) and color and it deletes the previous edge to the entrepreneur/employer. Finally, it starts counting the  $k$  periods of extra costs (to  $k$  is assigned the value `common.ExtraCostsDuration`, in the measure stated in `common.newEntrantExtraCosts`).

The code in `Agent.py` is:

---

<sup>15</sup>Related to Version 2.

<sup>16</sup>Related to Version 2.

```
myEntrepreneur=gvf.nx.neighbors(common.g, self)[0]
myEntrepreneurProfit=myEntrepreneur.profit
if myEntrepreneurProfit >= common.thresholdToEntrepreneur:
    print "I'm %2.0f and myEntrepreneurProfit is %4.2f" %\
        (self.number, myEntrepreneurProfit)
    common.g.remove_edge(myEntrepreneur, self)
    self.xPos-=15
    gvf.pos[self]=(self.xPos,self.yPos)
    # colors at http://www.w3schools.com/html/html_colornames.asp
    gvf.colors[self]="LawnGreen"
    self.agType="entrepreneurs"
    self.employed=True
    self.extraCostsResidualDuration=common.extraCostsDuration
```

- With the method (or command) **toWorker**,<sup>17</sup> an entrepreneur moves to be an unemployed worker if its profit at time  $t$  is  $\leq$  a given *threshold* in  $t$ . The threshold is retrieved from the variable **thresholdToWorker**.

The agent changes its internal type, position (not completely at the right as the original workers, but if it was a worker moved to entrepreneur and coming back, it goes completely at the right) and color and it deletes the previous edge to the workers/employee if any.

The code in **Agent.py** is:

```
if self.profit <= common.thresholdToWorker:
    print "I'm entrepreneur %2.0f and my profit is %4.2f" %\
        (self.number, self.profit)

# the list of the employees of the firm, IF ANY
entrepreneurWorkers=gvf.nx.neighbors(common.g,self)
print "entrepreneur", self.number, "has", len(entrepreneurWorkers),\
    "workers to be fired"

if len(entrepreneurWorkers) > 0:
    for aWorker in entrepreneurWorkers:
        gvf.colors[aWorker]="OrangeRed"
        aWorker.employed=False

        common.g.remove_edge(self, aWorker)

self.numOfWorkers=0

#originally, it was an entrepreneur
if self.xPos<0:gvf.pos[self]=(self.xPos+15,self.yPos)
#originally, it was a worker
else:gvf.pos[self]=(self.xPos,self.yPos)
# colors at http://www.w3schools.com/html/html_colornames.asp
gvf.colors[self]="OrangeRed"
self.agType="workers"
self.employed=False
```

---

<sup>17</sup>Related to Version 2.

### 1.2.6 Other features in scheduling [NB the notes with the \$\$ mark have to be reported in the Handbook and require code modification]

We also have two more sophisticated structures: the WorldState feature and the 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 we use a unique instance of the class, but the code is 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, with set/get methods for each variable.

In `Agent.py` we can ask to the WorldState, via `get`, for the values of the variables.

In the `oligopoly` project we make a step ahead, asking to the WorlState to make a specific evaluation.

\$\$ The normal use has in Col. B a value and in Col. C the method used to set that value in WorldState; the will be retrieved by the agents. Here, in Col. B we have a name, any name, in our case `specialUse` (an empty cell does not work) to signal the content of Col. C as a special method making *world calculations*. The final structure has to follow the usual one, having in Col. B a value or a method and, in the second case, with Col. C empty.

- The method (or command) `setMarketPriceV1`,<sup>18</sup> sent to the WorldState, orderr it to evaluate the market clearing price.

Setting the aggregate-demand  $D_t$  as equal to the production:

$$D_t = \sum_i P_t^i \quad (1.9)$$

We have the *demand function*, with  $p_t$  as price:

$$p_t = a + bD_t \quad (1.10)$$

---

<sup>18</sup>Introduced above as related to Version 1.



With the planned production coming from a Poisson distribution as in Eq. 1.5, considering  $\lambda$  set to 4, we can set two consistent points  $(p, D)$  as  $(1, 20)$  and  $(0.8, 30)$  obtaining:

$$p_t = 1.4 - 0.02D_t \quad (1.11)$$

The resulting code in `WorldState.py` is:

```
# set market price
def setMarketPriceV1(self):
    # to have a price around 1
    common.price= 1.4 - 0.02 * common.totalProductionInA_TimeStep
    print "Set market price to ", common.price
    common.price10=common.price*10 #to plot
```

- The method (or command) `setMarketPriceV2`,<sup>19</sup> sent to the `WorldState`, orders it to evaluate the market clearing price considering each agent behavior.

Having:

$$p_t = D_t/O_t \quad (1.12)$$

with  $p_t$  clearing market price at time  $t$ ;  $D_t$  demand in value at time  $t$ ;  $O_t$  offer in quantity (the production) at time  $t$ .

As defined above (p. 21), the method uses two common variables:

- `totalProductionInA_TimeStep`, generated by the agents (*entrepreneurs*), via `produce`;
- `totalPlannedConsumptionInValueInA_TimeStep`, generated by the agents (*entrepreneurs* and *workers*) via `planConsumptionInValue`.

The resulting code in `WorldState.py` is:

```
# set market price V2
def setMarketPriceV2(self):
    common.price= common.totalPlannedConsumptionInValueInA_TimeStep / \
                  common.totalProductionInA_TimeStep
    print "Set market price to ", common.price
    common.price10=common.price*10 #to plot
```

- *Just a memo:* we also have the possibility of using *macros* contained in separated sheets of the `schedule.xls` file (not used presently here).

---

<sup>19</sup>Introduced above as related to Version 2.

# 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>

# Index

action container, 12  
AESOP, 12  
agent creation, 6  
  
demand, 23  
demand function with numeric coefficients, 24  
demand functionV1, 23  
demand functionV2, 24  
  
evaluateProfit, 17  
evaluateProfitV0, 13  
  
fireIfProfit, 16  
  
hireFireWithProduction, 18  
hireIfProfit, 14  
  
macros, 23, 24  
makeProductionPlan, 17  
marketPriceV2, 21  
Methods used in Version 0, 13  
Methods used in Version 1, 17  
Methods used in Version 2, 20  
Methods used in Versions 0, 1, 2, 15  
Methods used in Versions 1, 2, 17  
Model, 8  
  
Observer, 8  
operating sets of agents, 8  
  
planConsumptionInValue, 20  
planned consumptions, 20  
predefining a default project, 3  
produce, 15  
production plan, 17  
production version 0, 15  
profit version 0, 14  
profit version 1, 18  
  
required labor force, 18  
schedule, 8, 10  
set of agents, 7  
setMarketPriceV1, 17  
Swarm, 12  
  
toEntrepreneur, 21  
toWorker, 22  
types of agents, 7  
  
V0, 12  
V1, 12  
V2, 13  
  
world state, 23  
WorldState, 14, 23