Oligopoly: the Making of the Simulation Model

Marco Mazzoli, Matteo Morini, and Pietro Terna

May 29, 2016

Contents

List of Figures

Chapter 1

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

Using SLAPP[1], 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.

- We can launch the SLAPP shell in several ways.
  - 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`
  - Alternatively, 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`
  - Using IPython (e.g., in a `Jupyter notebook`) we go to the main folder of SLAPP (or we start `Jupyter notebook`) form there, and we can launch SLAPP via the `iRunShell.ipynb` file that we find in that main folder, simply clicking on it.

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

[1] 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).

- We can predefining 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 the folder of the project we are working on (`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 second 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.[2]

- 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 value produces uniquely a graphic effect, as in Figure 1.3.

[2] `https://github.com/terna/SLAPP`.

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

  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.[3]

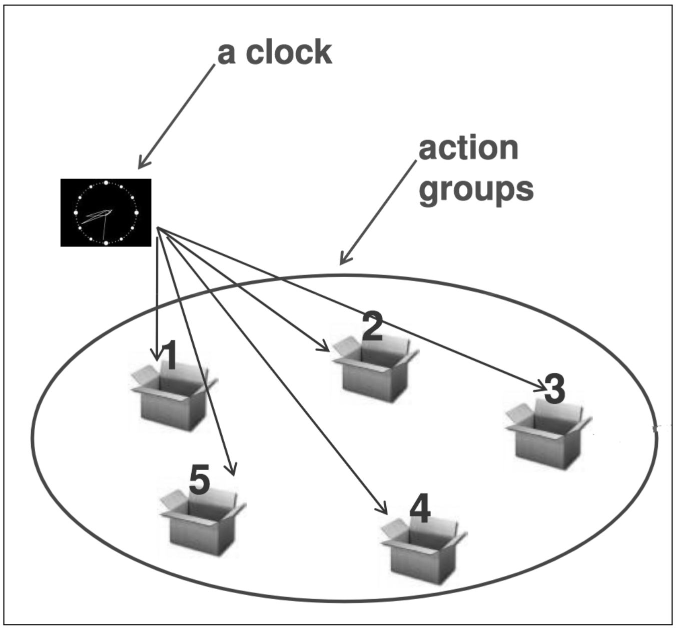

Figure 1.1: The representation of the schedule

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 10) and the x and y positions on the screen. See above the *sigma* value determining random shift from

[3]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.

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 75
2  -10 65
3  -10 55
4  -10 45
5  -10 35
6  -10 70
7  -10 60
8  -10 50
9  -10 40
10 -10 30
```

- `in`Versions 0 to 2, "workers.txt" list the agents of type `workers`, *not used here*; it is reporting the identification numbers 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*;
- the Version 3 of the `oligopoly` project uses the file `workers.txtx` where the extension `.txtx` or eXtended text, means that the file si built following the rule described into the Reference Handbook[4], subsection ”The use of files .txtx to define the agents”.

  In version 3 the content is:

  ```
  0@9999   10 &v=10*int(n/50)+5&
  ```

  that we read in the following way:

  - `0@9999` as the order of creating 10 thousand workers, from number 0 to number 9,999;
  - `10` is the constant value of the x coordinate of the worker-agents;
  - `&v=10*int(n/50)+5&` is a formula calculating the y coordinate of each agent:

    & opens and closes the formula;

[4] https://github.com/terna/SLAPP.

v is the result of the calculation, in our case the y coordinate;

n is the number of the agent, in the sequence generated in the interval from 0 to 9999.

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

                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 `agOperatingSets.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` or `.txtx` 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 Macro scheduling

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

- Two of those mechanisms are operating in a *macro* way: one at the level of the Observer and the other of the Model, with recurrent sequences of actions to be done.[5]

- 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, operating in a *micro* way, is the more detailed one (see Section 1.2.3).

[5]The level of the Observer is our level, where the experimenter looks at the model (the level of the Model) while it runs.

1.2.1 The scheduling mechanism at the level of the Observer

.

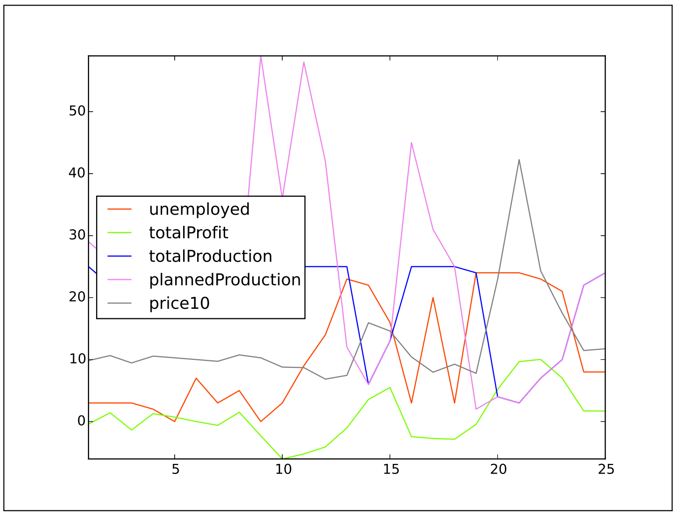

Figure 1.2: Time series generated by the model

- 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`

  - 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).[6]

[6]We can use both *visualizePlot* and *visualizeNet*—strictly in this order—or only one of them.

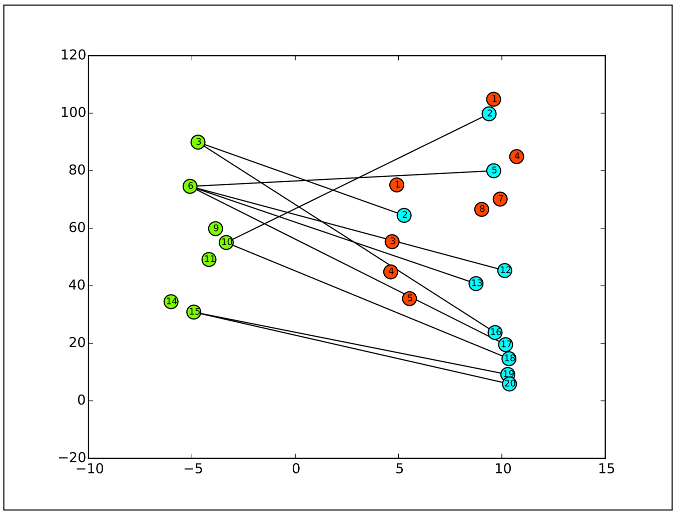

Figure 1.3: The agents (nodes), with random displacements, and links connecting entrepreneurs and workers

The plot of the variable price is scaled multiplying the price times 10, 100, 1000, to represent the price series in a readable way.

– `visualizeNet` update the windows reporting the links connecting entrepreneurs and workers, on a network basis (Figure 1.3). See also note 6.

– `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 can also consider 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 ??, 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 5.

  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.[7] `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 redefined 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

[7] That is in the "$$slapp$$" folder.

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)

.

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

- The third scheduling mechanism, as anticipated in Section ??, operates at a *micro* scale and it 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 instruction* 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 oligoply.

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

1.3 Micro scheduling: the AESOP level

From now on we explain the micro level of AESOP, i.e., the structure of the implementation of the Agents and Emergencies for Simulating Organizations in Python for the Oligopoly model,

1.3.1 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
```

Version 3 (GitHub, V3). Here we have (i) random production only at time 1, (ii) adaptation in production plans , (iii) individual demand curves with more realistic price determination, (iv) new entrant firms.

In `schedule3.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      adaptProductionPlan
entrepreneurs      hireFireWithProduction
entrepreneurs      produce
entrepreneurs      planConsumptionInValue
workers            planConsumptionInValue
WorldState         specialUse            setMarketPriceV3
entrepreneurs      evaluateProfit
entrepreneurs      0.0001                fireIfProfit
workers            toEntrepreneurV3
entrepreneurs      toWorkerV3
```

1.3.2 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$^{\text{th}}$ will be lacking of activity of the detailed scheduling activity (AESOP layer).

1.3.3 Methods used in Versions 0, 1, 2, 3

1.3.3.1 produce

- The method (or command) `produce`[8] sent to the `entrepreneurs` order them—in a deterministic way, in each unit of time—to produce proportionally to their labour force, obtaining profit Π^i_t, where i identifies the firm and t the time.

  L^i_t is the number of workers of firm i at time t, and also the number of its links. We add 1 to L^i_t, to account for the entrepreneur as a worker. π is the `laborProductivity`, with its value set to 1 in `common` variable space, currently not changing with t. P^i_t is the production of firm i at time t.

  The production is:
$$P^i_t = \pi(L^i_t + 1) \tag{1.1}$$

  The production of the i^{th} firm is added to the total production of the time step, in the variable `totalProductionInA_TimeStep` of the *common* space.

  The code is:

[8] Related to Versions 0, 1, 2. 3

```
# 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 calcutated 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

    # productivity is set to 1 in the benginning
    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 ...

1.3.3.2 fireIfProfit

- The method (or command) `fireIfProfit`[9] sent to the `entrepreneurs` order them—in a probabilistic way (50% of probability in versions 0, 1, 2; in version 3, considering that the probability is set directly in the `schedule.xls` file, we eliminate the effect of this command setting the probability to 0.01[10]), in each unit of time—to fire a worker (choosing her/him randomly 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):

$$\Pi_t^i < firingThreshold \rightarrow fire \tag{1.2}$$

```
# fireIfProfit
def fireIfProfit(self):
```

[9]Used in Versions 0, 1, 2, (temporary) 3.

[10]Being 0 not allowed, see the Reference Handbook, subsection *The detailed scheduling mechanism within the Model (AESOP level)*

```
# 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

#the list returnes by nx is unstable as order
entrepreneurWorkers = mySort(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"
```

See page 21 for the technical detail of the function `mySort`.

1.3.4 Methods used in Versions 1, 2, 3

1.3.4.1 makeProductionPlan

- The method (or command) `makeProductionPlan`[11] sent to the `entrepreneurs` order them to guess their production for the current period. The production plan $\widehat{P}_t^i$ is determined in a random way, using a Poisson distribution, with $\lambda = 10$ as mean (suggested value kept in the *common* space).

  As a definition, the production plan is:

$$\widehat{P}_t^i \sim \; Pois(\lambda) \tag{1.3}$$

  We suggest temporary a value of 5 for λ, with (in Versions 1 and 2) the quantities: entrepreneurs 5, workers 20 + the 5 entrepreneurs, labor productivity 1. Always in Versions 1 and 2, the value of λ can be modified in the prologue of the run).

[11]Related to Versions 1, 2; in the 3 case, only at time=1

With Version 3, the `makeProductionPlan` method works uniquely with $t = 1$ being t internally `common.cycle` created and set to 1 by `ObserverSwarm` when starts.

Version 3 calculates the initial value λ (used uniquely in the first step) as:

$$\lambda = \rho \frac{(N_{workers} + N_{entrepreneurs})}{N_{entrepreneurs}} \tag{1.4}$$

In this way about a ρ ratio of the agents is producing in the beginning. Internally, the total numbers of the agents $N_{workers} + N_{entrepreneurs}$ can be obtained as the length of the `agentList`; the number of entrepreneurs is calculated from the same list considering only the etnrepreneurs.

The code is:

```
# makeProductionPlan
def makeProductionPlan(self):

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

    if common.projectVersion >= 3 and common.cycle==1:
        nEntrepreneurs = 0
        for ag in self.agentList:
            if ag.agType=="entrepreneurs":
                nEntrepreneurs+=1
        #print nEntrepreneurs
        nWorkersPlus_nEntrepreneurs=len(self.agentList)
        #print nWorkersPlus_nEntrepreneurs
        common.Lambda=(common.rho*nWorkersPlus_nEntrepreneurs)/nEntrepreneurs
        #print common.rho, common.Lambda

    if (common.projectVersion >= 3 and common.cycle==1) or \
        common.projectVersion < 3:
        self.plannedProduction=npr.poisson(common.Lambda,1)[0] # 1 is the number
        # of element of the returned matrix (vector)
        #print self.plannedProduction
```

1.3.4.2 evaluateProfit

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

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

[12] Related to Versions 1, 2, 3.

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 1.3.10.

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:

$$\Pi_t^i = p_t P_t^i - w L_t^i - C \tag{1.5}$$

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=common.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 pruducing workers is obtained indirectly via
    # production/laborProductivity
    #print self.production/common.laborProductivity
    self.costs=common.wage * (self.production/common.laborProductivity) + \
                XC
    self.profit=common.price * self.production - self.costs
    #print self.number, "profit", self.profit
```

1.3.4.3 hireFireWithProduction

- The method (or command) `hireFireWithProduction`[13] sent to the `entrepreneurs` order them to hire or fire comparing the labor forces required for the production plan $\widehat{P}_t^i$ and the labor productivity π; we have the required labor force (L_t^i is the current one):

$$\widehat{L}_t^i = \widehat{P}_t^i / \pi \tag{1.6}$$

Now:

[13] Related to Versions 1, 2, 3.

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

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 calcutated 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

#the list returnes by nx is unstable as order
entrepreneurWorkers = mySort(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"
```

An important technical detail is the use of the function `mySort` to avoid inconsistencies in the order of the agents returned by the graph of the networks as workers of the entrepreneur. Different orders would produce different sets of fired workers, becoming different sets of potential entrepreneurs and producing different sequences of events in the simulation.

Why the differences in the order of the list of the agents? The graph is managed by `networkX`, which is using internally a dictionary structure, whose order is neither defined in any way in Python, nor constant from execution to execution. The list, in our case, contains the addresses of the instances of the agents. A simple sort of this list does not give us a stable order, due to the fact that the addresses and their order can change form a run to another.

For these reasons we use here a custom function to sort the list, using the internal `number` of the agents, to reorder them.
The code of the function `mySort` is:

```
def mySort(ag):
    if ag==[]: return []
    numAg=[]
    for a in ag:
        numAg.append((a.number,a))
    numAg.sort()
    agSorted=[]
    for i in range(len(numAg)):
        agSorted.append(numAg[i][1])
    return agSorted
```

1.3.5 Methods used in Version 2, 3

1.3.5.1 planConsumptionInValue

- The method (or command) `planConsumptionInValue`,[14] sent to `entrepreneurs` or `workers`, produces the following evaluations, detailed in `commonVar.py` file.

  Consumption behavior with

  $$C_i = a_i + b_i Y_i + u \tag{1.7}$$

  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
    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
```

[14] Related to Version 2, 3.

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

1.3.5.2 setMarketPriceV2

- The method (or command) `setMarketPriceV2`,[15] 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 1.3.10.

1.3.6 Methods used in Version 3 only

1.3.6.1 toEntrepreneurV3

- With the method (or command) `toEntrepreneurV3`,[16] sent to `workers`, the agent, being a worker, decides to became an entrepreneur at time t, if its employer has a relative profit (reported to the total of the costs) $\geq$ a given *threshold* at time $t-1$. The threshold is retrieved from the variable `thresholdToEntrepreneur`.

  The decision is a quite rare one, so we have to pass a higher level threshold, that we define as `absoluteBarrierToBecomeEntrepreneur`; the value is defined in `commonVar.py` file and shown via `paprameters.py` file.

  This parameter represents a *potential max number of new entrepreneurs* in each cycle.

  It works in the following way: given an absolute value as number workers being candidates to became entrepreneur, let say one each two time steps (0.5—as a mean—in each time step), we transform that value in a probability, dividing by the total number of the agents, used as an adaptive scale factor.

[15] Related to Version 2, 3.
[16] Related to Version 3.

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:

```
#to entrepreneurV3
def toEntrepreneurV3(self):
    if self.agType != "workers" or not self.employed: return

    if random() <= common.absoluteBarrierToBecomeEntrepreneur:
      myEntrepreneur=gvf.nx.neighbors(common.g, self)[0]
      myEntrepreneurProfit=myEntrepreneur.profit
      myEntrepreneurCosts=myEntrepreneur.costs
      if myEntrepreneurProfit/myEntrepreneurCosts  >= \
        common.thresholdToEntrepreneur:
        print "I'm worker %2.0f and my entrepreneur relative profit is %4.2f" %\
              (self.number, myEntrepreneurProfit/myEntrepreneurCosts)
        common.g.remove_edge(myEntrepreneur, self)

        #originally, it was a worker
        if self.xPos>0:gvf.pos[self]=(self.xPos-15,self.yPos)
        #originally, it was an entrepreneur
        else: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
```

1.3.6.2 toWorkerV3

- With the method (or command) `toWorkerV3`,[17] an entrepreneur moves to be an unemployed worker if its a relative profit (reported to the total of the costs) 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:

```
#to workersV3
def toWorkerV3(self):
    if self.agType != "entrepreneurs": return
```

[17] Related to Version 3.

```
            #check for newborn firms
            try:
                self.costs
            except:
                return

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


                # 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
```

- The method (or command) `adaptProductionPlan`,[18] sent to `entrepreneurs` orders to the i^{th} firm to set its production plan for the current period to their (equal, being i independent) fraction of the total production of the previous period, plus $u_t^i \sim \mathcal{N}(0,k)$, as a random normal addendum.

  This method works only for time > 1.

  With $\widehat{P}_t^i$ being the planned production of firm i, we have:

$$\widehat{P}_t^i = \frac{\sum_i P_{t-1}^i}{N_{entrepreneurs}} + u_t^i \tag{1.8}$$

  with $u_t^i \sim \mathcal{N}(0,k)$ and $k = \frac{\sum_i P_{t-1}^i}{N_{entrepreneurs}}/10$

  The code in `Agent.py` is:

```
        # adaptProductionPlan
        def adaptProductionPlan(self):
```

[18]Related to Version 3.

```
if common.cycle > 1:
  nEntrepreneurs = 0
  for ag in self.agentList:
     if ag.agType=="entrepreneurs":
         nEntrepreneurs+=1

  self.plannedProduction = common.totalProductionInPrevious_TimeStep  \
        / nEntrepreneurs
  self.plannedProduction += gauss(0,self.plannedProduction/10)
```

1.3.6.3 setMarketPriceV3

- The method (or command) `setMarketPriceV3`,[19] sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 1.3.10.

1.3.7 Methods used in Version 0 only

1.3.7.1 evaluateProfitV0

- The method (or command) `evaluateProfitV0`[20] sent to the `entrepreneurs` order them to calculate their profit. Being P_t^i the production and π 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 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:

$$\Pi_t^i = L_t^i (R - w) + u_t^i \tag{1.9}$$

  The code is:

```
# calculateProfit
def evaluateProfitV0(self):

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

    # the number of producing 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)
```

[19]Related to Version 3.
[20]Related to Version 0.

1.3.7.2 hireIfProfit

- The method (or command) `hireIfProfit`[21] 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 a than the value `hiringThreshold` (temporary: 0):

$$\Pi_t^i > hiringThreshold \rightarrow hire \tag{1.10}$$

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)

        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"
```

[21] Used in Version 0.

1.3.8 Methods used in Version 1 only

1.3.8.1 setMaketPriceV1

- The method (or command) `setMarketPriceV1`,[22] sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 1.3.10.

1.3.9 Methods used in Version 2 only

1.3.9.1 toEntrepreneur

- With the method (or command) `toEntrepreneur`,[23] 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:

```
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
```

1.3.9.2 toWorker

- With the method (or command) `toWorker`,[24] 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`.

[22] Related to Version 1.
[23] Related to Version 2.
[24] Related to Version 2.

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
```

1.3.9.3 setMarketPriceV2

- The method (or command) `setMarketPriceV2`,[25] sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 1.3.10.

1.3.10 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 sophisticates structures: the WorldState feature and the macros.

[25] Related to Version 2.

- 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.

1.3.10.1 setMarketPriceV1 as in WorldState, with details

- The method (or command) `setMarketPriceV1`,[26] sent to the `WorldState`, orders it to evaluate the market clearing price.

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

  $$D_t = \sum_i P_t^i \tag{1.11}$$

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

  $$p_t = a + bD_t \tag{1.12}$$

  With the planned production coming from a Poisson distribution as in Eq. 1.3, considering λ set to 4, we can set two consistent points (p, D) as $(1, 20)$ and $(0.8, 30)$ obtaining:

  $$p_t = 1.4 - 0.02D_t \tag{1.13}$$

  The resulting code in `WorldState.py` is:

[26]Introduced above as related to Version 1 only.

```
# 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
```

1.3.10.2 setMarketPriceV2, as in WorldState, with details

- The method (or command) `setMarketPriceV2`,[27] sent to the `WorldState`, orders it to evaluate the market clearing price considering each agent behavior.

  Having:

  $$p_t = D_t/O_t \tag{1.14}$$

  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. 23), 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
```

1.3.10.3 setMarketPriceV3, as in WorldState, with details

- The method (or command) `setMarketPriceV3`,[28] sent to the `WorldState`, orders it to evaluate the market clearing price considering each agent behavior and *an external shock, potentially large.*

  We introduce a shock Ξ uniformly distributed between $-L$ and $+L$ where L is a rate on base 1, e.g., 0.10. To keep the effect as symmetric, we have now to equation determining the clearing price:

[27] Introduced above as related to Version 2 only.
[28] Introduced above as related to Version 3 only.

If the shock Ξ is (≥ 0):

$$p_t = \frac{D_t(1+\Xi)}{O_t} \tag{1.15}$$

if the shock Ξ is (< 0):

$$p_t = \frac{D_t/(1+\Xi)}{O_t} \tag{1.16}$$

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. 23), 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 V3
def setMarketPriceV3(self):
    shock0=random.uniform(-common.maxDemandRelativeRandomShock, \
                          common.maxDemandRelativeRandomShock)
    shock=shock0
    if shock >= 0:
      common.price= (common.totalPlannedConsumptionInValueInA_TimeStep * \
                     (1 + shock))  \
                     / common.totalProductionInA_TimeStep
      print "Set market price to ", common.price
      print "Relative shock (symmetric) ", shock0

    shock=shock0
    if shock <  0:
      shock *=-1. #always positive, boing added to the denominator
      common.price= (common.totalPlannedConsumptionInValueInA_TimeStep / \
                     (1 + shock))  \
                     / common.totalProductionInA_TimeStep
      print "Set market price to ", common.price
      print "Relative shock (symmetric) ", shock0
```

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

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