Oligopoly: the Making of the Simulation Model

Marco Mazzoli, Matteo Morini, and Pietro Terna

April 8, 2017

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:

```
1001@11000   10 &v=10*int((n-1001)/50)+5&
```

that we read in the following way:

- `1001@11000` as the order of creating 10 thousand workers, from number 1001 to number 11,000;

- `10` is the constant value of the x coordinate of the worker-agents;

- `&v=10*int((n-1001)/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 1001 to 11,000.

– numbering starts from 1001 for the reasons explained at page 22.

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 Agents and reset action

The `reset` (see page 13) action, working into the scheduling of the model (Section 1.2.2), activates the method `setNewCycleValues` defined, as an empty step, in the

class `SuperAgent` in `agTools` of *SLAPP* (folder `$$slapp$$`). In the `oligopoly` project, that method is redefined in `Agent.py`. The `reset` action acts once in each simulation cycle, because in our case is related only to common variables of the simulation. The agent executing the cleaning operation is that with the identifier (the variable *number*) equal to 1. If no agent has that identifier, all will be acting, with not useful repetitions of the same task.

As a consequence, in this project pay attention that at least one of the agents has 1 as identifier.

1.1.2 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).

1.2.1 The scheduling mechanism at the level of the Observer

.

Figure 1.2: Time series generated by the model. version 4

- 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:

---

[5]The level of the Observer is our level, where the experimenter looks at the model (the level of the Model) while it runs.
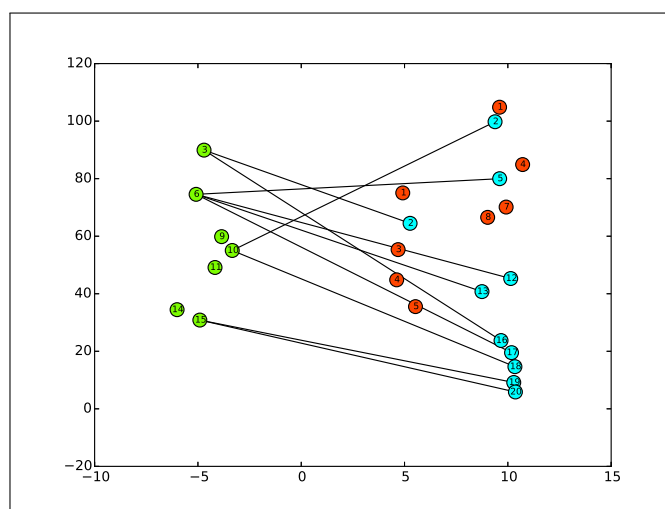
Figure 1.3: The agents (nodes), with random displacements, and links connecting entrepreneurs and workers

```
collectStructuralData modelStep collectTimeSeries
visualizePlot visualizeNet clock
```

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

```
collectStructuralData modelStep collectTimeSeries
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).

– `collectStructuralData` collects the number of workers and of entrepreneurs *at the beginning* of each period, both as a basis for internal calculations and for the final output of the model, when two files of data are generated.[6]

– `modelStep` orders to the model to make a step forward in time.

– `collectTimeSeries` collects the data of the outcomes of the simulation *at the end* of each period, both as a basis for the action of

---

[6] *collectTimeSeries*, *visualizePlot* and *saveTimeSeries* are contained in "oActions.py" and are all using *pandas* as dataframe manager (look at http://pandas.pydata.org).

       `visualizePlot` and for the final output of the model, when two files of data are generated (with extension .csv and date and time[7] in their names).[6]

  – `visualizePlot` update the plot of the time series generated by the model (Figure 1.2).[8]

  – `visualizeNet` update the windows reporting the links connecting entrepreneurs and workers, on a network basis (Figure 1.3).[8]

  – `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` asks 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 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 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.

---

[7] Avoiding : into the name, for compatibility reasons with Windows

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

– 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.[9] `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.

Always in our case, as explained in Section 1.1.1, we suppose that the acting agent in resetting step would be that with 1 as identifier.

In our model, we clean the variables:
`totalProductionInA_TimeStep`,
`totalPlannedConsumptionInValueInA_TimeStep`,
`totalProfit` and
`totalPlannedProduction`
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):
    # the if is to save time, given that the order is arriving to
    # all the agents (in principle, to reset local variables)
    if not common.agent1existing:
        print "At least one of the agents has to have number==1"
        print "Missing that agent, all the agents are resetting common values"

    if self.number==1 or not common.agent1existing:
        common.totalProductionInA_TimeStep=0
        common.totalPlannedConsumptionInValueInA_TimeStep=0
        common.totalProfit=0
        common.totalPlannedProduction=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).

---

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

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 1.2, 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.

1.3.1.1 Version 0 (GitHub: master), preliminary step

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
```

1.3.1.2 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
```

1.3.1.3 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
```

1.3.1.4 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
```

1.3.1.5 Version 4 (GitHub: V4)

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 `schedule4.xls` (to be copied to `schedule.xls` for the use) we have (comments start at column E) 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                          COMMENT: below an experimental step
                                            WorldState      specialUse        randomShockToWages
                                            Temporary step
                                            to check the model
                                            sensitivity
WorldState          specialUse              fullEmploymentEffectOnWages
WorldState          specialUse              incumbentActionOnWages
```

1.3.1.6 Version 5 and 5b (GitHub: V5 and V5b)

Version 5b is related uniquely to a correction in method `adaptProductionPlan`, as in subsection 1.3.5.3, now modified as in subsection 1.3.5.4. The schedule is unchanged form Versione 5 to 5b.

We add now the possibility of work troubles in firms, via the method entrepreneurs p work troubles, where p is a probability.

In `schedule5.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         0.05                      workTroubles
entrepreneurs         produceV5
entrepreneurs         planConsumptionInValueV5
workers               planConsumptionInValueV5
WorldState            computationalUse          setMarketPriceV3
entrepreneurs         evaluateProfitV5
entrepreneurs         0.0001                    fireIfProfit
workers               toEntrepreneurV3
entrepreneurs         toWorkerV3
WorldState            computationalUse          fullEmploymentEffectOnWages
WorldState            computationalUse          incumbentActionOnWages
```

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 ≤ 100; if > 100 the steps after the 100th will be lacking of activity of the detailed scheduling activity (AESOP layer).

1.3.3 Methods used in Versions 0, 1, 2, 3, 4, 5

1.3.3.1 fireIfProfit

- The method (or command) `fireIfProfit`,[10] sent to the `entrepreneurs`, orders them—in a probabilistic way (50% of probability in versions 0, 1, 2; in version 3 and 4, 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[11]), in each unit of time—to fire a worker (choosing her/him

---

[10]Used in Versions 0, 1, 2, (temporary) 3, 4 and 5.

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

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):

$$\Pi_t^i < firingThreshold \rightarrow fire \tag{1.1}$$

```
# 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

    #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 for the technical detail of the function `mySort`.

1.3.4   Methods used in Versions 1, 2, 3, 4, 5

1.3.4.1   makeProductionPlan

- The method (or command) `makeProductionPlan`,[12] sent to the `entrepreneurs`, orders 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 $\nu = 10$ as mean (suggested value kept in the *common* space).

---

[12]Related to Versions 1, 2; in the 3, 4 and 5 cases, only at time=1

As a definition, the production plan is:

$$\widehat{P}_t^i \sim Pois(\nu) \tag{1.2}$$

We suggest temporary a value of 5 for ν, 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 ν can be modified in the prologue of the run).

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 ν (used uniquely in the first step) as:

$$\nu = \rho \frac{(N_{workers} + N_{entrepreneurs})}{N_{entrepreneurs}} \tag{1.3}$$

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 entrepreneurs.

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.nu=(common.rho*nWorkersPlus_nEntrepreneurs)/nEntrepreneurs
        #print common.rho, common.nu

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

    common.totalPlannedProduction+=self.plannedProduction
```

1.3.4.2   hireFireWithProduction

- The method (or command) `hireFireWithProduction`,[13] sent to the `entrepreneurs`, orders 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):

$$\widehat{L}_t^i = \widehat{P}_t^i/\pi \tag{1.4}$$

Now:

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))
```

---

[13]Related to Versions 1, 2, 3, 4, 5.

```
            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 returns 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.[14]

As a related consequence, we have to keep in mind to avoid duplicated numbers: in the *Oligopoly* model the entrepreneurs can switch to workers and vice versa, so the numbers assigned to the entrepreneurs start from 1 and those assigned to the workers from 1001 (see the file `workers.txtx`).

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
```

1.3.4.3 planConsumptionInValue

- The method (or command) `planConsumptionInValue`,[15] sent to `entrepreneurs` or `workers`, produces the following evaluations, detailed in `commonVar.py` file.

The method (or command) `planConsumptionInValue` operates both with the `entrepreneurs` and the `workers`, producing the following evaluations, using the parameters reported into Table **??**. The description below is unique for both the cases.

@ptpt [speficato il tempo)The resulting consumption behavior if the agent i at time t is:

$$C_t^i = a_j + b_j Y_t^i + u \tag{1.5}$$

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

Considering w as wage, as above, and P for profit, the individual i can be :

- case $j = 1$: an entrepreneur, with $Y_t^i = P_{t-1}^i + w_t$;

---

[14]A related marginal problem, not eliminated, is the order in plotting the nodes in the graph plot: in the area where the nodes are superimposed, we can see the nodes exactly in the same position in every run, but differently placed as layer in the foreground/background sequence.

[15]Related to Version 2, 3, 4

- case $j = 2$: an employed worker at time t, with $Y_i = w$ and the special [16] case $Y_t^i = wc_t^i$, with wc_t^i defined in eq. 1.12;

- case $j = 3$: an unemployed worker at time t, with $Y_t^i = sw$ (sw = social wage, as a welfare intervention).

The a_j and b_j values are reported in the initial output of each run; we set them 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
```

The individual C_t^i updates `totalPlannedConsumptionInValueInA_TimeStep` (a *common* value), cleaned at each reset, i.e., at each new time step.

The `totalPlannedConsumptionInValueInA_TimeStep` measure will be then randomly corrected within the `setMarketPriceV3` method of the *WorldState* meta-agent, see page 43.

_____

[16] Activated if the parameter *cut also the wages* is set to *yes*

1.3.5 Methods used in Version 3, 4, 5, 5b

1.3.5.1 toEntrepreneurV3

- With the method (or command) `toEntrepreneurV3`,[17] 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` and shown via `parameters.py` file.

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

Internally, it works in the following way: given an absolute value as number workers actually became entrepreneur, we transform that value in a probability, dividing it by the total number of the agents, used as an adaptive scale factor.

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 "Worker %2.0f is now an entrepreneur (previous firm relative profit %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)
```

---

[17]Related to Version 3, 4, 5.

```
# colors at http://www.w3schools.com/html/html_colornames.asp
gvf.colors[self]="LawnGreen"
self.agType="entrepreneurs"
self.employed=True
self.extraCostsResidualDuration=common.extraCostsDuration
```

1.3.5.2 toWorkerV3

- With the method (or command) `toWorkerV3`,[18] 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

    #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
```

---

[18]Related to Version 3, 4, 5.

```
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
```

1.3.5.3 adaptProductionPlan until Version 5

- The method (or command) `adaptProductionPlan`,[19] sent to `entrepreneurs`, orders to the ith firm to set its production plan for the current period to their (equal, being i here not relevant) fraction of the total demand of the previous period, corrected with a random uniform relative correction in the interval $-k$ to $+k$, reported in the prologue as:
  `Random component of planned production`.

  This method works for time > 1.

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

    - if $u_t^i \geq 0$
      $$\widehat{P}_t^i = \frac{D_{t-1}}{N_{entrepreneurs}}(1 + u_t^i) \qquad (1.6)$$

    - if $u_t^i < 0$
      $$\widehat{P}_t^i = \frac{D_{t-1}}{N_{entrepreneurs}}/(1 + |u_t^i|) \qquad (1.7)$$

  with $u_t^i \sim \mathcal{U}(-k, k)$

  The code in `Agent.py` until Version 5 is:

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

      self.plannedProduction = common.totalDemandInPrevious_TimeStep  \
              / nEntrepreneurs

      #self.plannedProduction += gauss(0,self.plannedProduction/10)

      shock= uniform( \
       -common.randomComponentOfPlannedProduction,\
        common.randomComponentOfPlannedProduction)

      if shock >= 0:
       self.plannedProduction *= (1.+shock)
```

---

[19]Related to Version 3, 4, 5.

```
if shock < 0:
 shock *= -1.
 self.plannedProduction /= (1.+shock)
#print self.number, self.plannedProduction

common.totalPlannedProduction+=self.plannedProduction
```

1.3.5.4 adaptProductionPlan with Version 5b corrrection

- The method (or command) `adaptProductionPlan`,[20] sent to `entrepreneurs`, orders to the ith firm to set its production plan for the current period to their (equal, being i here not relevant) fraction of the total demand—transformed from its nominal value to the real one (i.e., in quantity)[21]—of the previous period, corrected with a random uniform relative correction in the interval $-v$ to v, reported in the prologue as:
`Random component of planned production`.

  This method works for time > 1.

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

  - if $u_t^i \geq 0$

  $$\widehat{P}_t^i = \frac{\frac{D_{t-1}}{p_{t-2}}}{N_{entrepreneurs}}(1 + u_t^i) \qquad (1.8)$$

  - if $u_t^i < 0$

  $$\widehat{P}_t^i = \frac{\frac{D_{t-1}}{p_{t-2}}}{N_{entrepreneurs}}/(1 + |u_t^i|) \qquad (1.9)$$

  with $u_t^i \sim \mathcal{U}(-v, v)$ and p_{t-2} the lagged price.[22]

  The double lagged price correction is justified because we are considering the production decisions at time t, which are based on the decisions of consumption at $t - 1$, related to the income at time $t - 1$; these decisions are made before the determination of the prices at $t - 1$ (emerging only when comparing the demand and the predetermined offer). If we want to evaluate the consumption in quantity, without the effect of a too limited or too abundant

---

[20]Related to Version 5b.

[21]The missing part until Version 5b was this transformation; as a consequence, the result was partially biased, anyway with limited effects being our prices always around the unity; I have to thank Enrico Minardi, a student of mine, for discovering the missing operation.

[22]The method is applied only with $t > 1$, so the use of the lagged price starts at time 2, when p_{t-2} would be the undefined p_0 value; as a simplification we use p_{t-1} in this case, look at the code.

offer, we have to use $t-2$ prices. This construction will be eliminated with a future version of the model, with the atomic interaction of buyers and sellers in a dispersed way.

The code in `Agent.py` from Version 5b is:

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

        #previous period price
        #print ("++++++++++++++++++++++", common.ts_df.price.values[-1])
        #print ("&&&&&&&&&&&&&&&&&&&&&&",len(common.ts_df.price.values))

        if len(common.ts_df.price.values)==1:
            previuosPrice=common.ts_df.price.values[-1] # t=2
        if len(common.ts_df.price.values)>1:
            previuosPrice=common.ts_df.price.values[-2] # t>2
        #NB adapt acts from t>1

        self.plannedProduction = (common.totalDemandInPrevious_TimeStep / \
                                  previuosPrice) \
                                  / nEntrepreneurs

        #self.plannedProduction += gauss(0,self.plannedProduction/10)

        shock= uniform( \
         -common.randomComponentOfPlannedProduction,\
          common.randomComponentOfPlannedProduction)

        if shock >= 0:
         self.plannedProduction *= (1.+shock)

        if shock < 0:
         shock *= -1.
         self.plannedProduction /= (1.+shock)
        #print self.number, self.plannedProduction

        common.totalPlannedProduction+=self.plannedProduction
        #print "entrepreneur", self.number, "plan", self.plannedProduction,\
        #    "total", common.totalPlannedProduction
```

1.3.5.5   setMarketPriceV3

- The method (or command) `setMarketPriceV3`,[23] sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 1.3.13 and specifically Section 1.3.13.3.

---

[23]Related to Version 3, 4, 5.

1.3.6   Methods used in Version 4, 5 only

1.3.6.1   fullEmploymentEffectOnWages

The method (or command) `fullEmploymentEffectOnWages`,[24] sent to the
`WorldState`, orders it to modify wages accordingly to full employment situ-
ation, in a reversible way. See below Section 1.3.13 and specifically Section
1.3.13.4.

1.3.6.2   randomShockToWages

The method (or command) `randomShockToWages`,[25] sent to the `WorldState`,
orders it to randomly modify wages. See below Section 1.3.13 and specifically
Section 1.3.13.5.

This method is only used in model building, to verify the sensitivity of the
model to changes in wages.

1.3.6.3   incumbentActionOnWages

The method (or command) `incumbentActionOnWages`,[26] sent to the `WorldState`,
orders it to modify wages for one period, accordingly to the attempt of cre-
ating an entry barrier when new firms are observed into the market.

As a consequence, wage measure contains a variable addendum, set to 0 as
regular value and modified temporary by this method.

See below Section 1.3.13 and specifically Section 1.3.13.6.

1.3.7   Methods used in Version 5 only

1.3.7.1   planConsumptionInValueV5

- The method (or command) `planConsumptionInValueV5`,[27] sent to `entrepreneurs`
  or `workers`, produces the following evaluations, detailed in `commonVar.py`
  file.

---

[24]Related to Version 4, 5.
[25]Related to Version 4, 5.
[26]Related to Version 4, 5.
[27]Related to Version 5

Consumption behavior with

$$C_i = a_k + b_k Y_i + u \tag{1.10}$$

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

The individual i can be:

1. an entrepreneur, with $Y_i = profit_{i,t-1} + wage$;

2. an employed worker, with $Y_i = wage$ and the special [28] case $Y_i = wc_t^i$, with wc_t^i defined in eq.1.12;

3. an unemployed workers, with $Y_i = socialWelfareCompensation$.

The a_k and b_k 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 planConsumptionInValueV5(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:
      # the followin if/else structure is for control reasons because if
      # not common.wageCutForWorkTroubles we do not take in account
      # self.workTroubles also if != 0; if = 0 is non relevant in any case
      if common.wageCutForWorkTroubles:
        self.consumption = common.a2 + \
                           common.b2 * common.wage*(1.-self.workTroubles) + \
                           gauss(0,common.consumptionRandomComponentSD)
        #print "worker", self.number, "wage x",(1.-self.workTroubles)
      else:
        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 + \
```

[28] Activated if the *common* value *wageCutForWorkTroubles* is *true*

```
                              gauss(0,common.consumptionRandomComponentSD)

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

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

The `totalPlannedConsumptionInValueInA_TimeStep` measure will be then randomly corrected within the `setMarketPriceV3` method of the *WorldState* meta-agent, see page 43.

1.3.7.2   workTroubles

- For each entrepreneur at time t, so for each firm i, we generate a shock $\psi_{i,t} > 0$ due to work troubles, with probability p_ψ (set for all the entrepreneurs via the `schedule.txt` file)[29] and value uniformly distributed between $V_\Psi/2$ and V_Ψ. The shock reduces the production of firm i in a relative way, as in:

$$Pc_t^i = P_t^i(1 - \psi_{i,t}) \tag{1.11}$$

where Pc means *corrected production*.

If the global logical value `wageCutForWorkTroubles` is *true*, also wages are cut in the same proportion that the production is suffering. With w indicating the constant basic wage level, cw_t^i is the corrected value at time t and for firm i; the correction is superimposed to the other possible corrections (due to full employment or to artificial barrier creation).

$$cw_t^i = w(1 - \psi_{i,t}) \tag{1.12}$$

The firm variable `hasTroubles` takes note—via $\psi_{i,t}$ assuming a value > 0, being 0 otherwise—if the firm has work problems in the current time step and the worker variable `workTroubles` takes note of the same amount for all the workers of that specific firm.

Both the variable are set again to 0 in the `reset` step at the beginning of each model cycle.

The code in `Agent.py` is:

---

[29]SLAPP displays—in its text output—a dictionary with the method probabilities, if at least one method is linked to a probability.

```
#work troubles
def workTroubles(self):

    # NB this method acts with the probability set in the schedule.txt
    # file
    if self.agType != "entrepreneurs": return

    # production shock due to work troubles

    psiShock=uniform(common.productionCorrectionPsi/2,
                     common.productionCorrectionPsi)
    self.hasTroubles=psiShock
    print "Entrepreneur", self.number, "is suffering a reduction of "\
          "production of", psiShock*100, "%, due to work troubles"

    if common.wageCutForWorkTroubles:
      # the list of the employees of the firm
      entrepreneurWorkers=gvf.nx.neighbors(common.g,self)
      for aWorker in entrepreneurWorkers:
         #avoiding the entrepreneur herself, as we are refering to her
         # network of workers
         aWorker.workTroubles=psiShock
         print "Worker ", aWorker.number, "is suffering a reduction of "\
               "wage of", psiShock*100, "%, due to work troubles"
```

1.3.7.3   produceV5

- The method (or command) `produceV5`,[30] sent to the `entrepreneurs`, orders them—in a deterministic way, in each unit of time—to produce proportionally to their labour force, obtaining profit Π_t^i, where i identifies the firm and t the time.

L_t^i is the number of workers of 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. π 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:
$$P_t^i = \pi(L_t^i + 1) \tag{1.13}$$

The production is corrected for work troubles (as in section 1.3.7.2) calculating the corrected value Pc_t^i with:

The production is:
$$Pc_t^i = P_t^i(1 - \psi_{i,t}) \tag{1.14}$$

The production (corrected or not) of the ith firm is added to the total production of the time step, in the variable `totalProductionInA_TimeStep` of the *common* space.

---

[30]Related to Version 5

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 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 . . .

1.3.7.4   evaluateProfitV5

- The method (or command) `evaluateProfit`,[31] sent to the `entrepreneurs`, orders them to calculate their profit. Being P_t^i the production and the labor force L_t^i measured via the network connecting the entrepreneur to her workers plus 1 to take in account the entrepreneur herself.

  The use of P_t^i, the actual production of the entrepreneurs, accounts both for the production plan decided with `adaptProductionPlan`, page 26, and for the limits in hiring, if any, as in `hireFireWithProduction`, page 20. The sum of all the actual productions of each entrepreneur is used, as at page 43, in `setMarketPriceV3`.

  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.

---

[31]Related to Version 5.

p_t is the `price`, clearing the market at time t and it is calculated by the abstract item WorldState via the method `setMarketPrice`, as explained in Section 1.3.13.

w is the `wage` per employee and time unit, set to 1.0 in `common` variable space, not changing with t, but the case of the important events of:

  – wage rise due both to full employment (Subsection 1.3.6.1) and

  – to the creation of barriers against new entrants (Subsection 1.3.6.3).

C are extra costs for new entrant firms. They are calibrated to assure the effectivness of the action described in Subsection 1.3.6.3, but in a non deterministic way, thanks to the movements in prices.

If the *common* variable `wageCutForWorkTroubles` is set to *True* the costs determination takes in account the reduction in the wages (but the wage of the entrepreneur, not changing).

Considering the presence of work troubles (see subsection 1.3.7.2) the determination of the clearing price, as at page 43, can signal an increase in the equilibrium price, due to the lacking production.

The (relative) shock $\psi_{i,t} > 0$ due to work troubles is defined in subsection 1.3.7.2.

In presence of work troubles the firm has to accept a reduction of its price, to compensate its customers for having undermined the confidence in the implicit commitment of producing a given quantity (the production plan, specified in subsection 1.3.4.1).

That penalty value, as a relative measure, is in *common* as `penaltyValue` and here shortly as pv. Locally, pv_t^i, for the firm i at time t, is set to pv if $\psi_{i,t} > 0$; otherwise ($\psi_{i,t} = 0$) is set to 0.

The profit evaluation, if `wageCutForWorkTroubles` is set to *True*, is:

$$\Pi_t^i = p_t(1 - pv_t^1)P_t^i - (w - \psi_{i,t})(L_t^i - 1) - 1w - C \qquad (1.15)$$

being $1w$ the wage of the entrepreneur.

If `wageCutForWorkTroubles` is set to *False*, the result is:

$$\Pi_t^i = p_t(1 - pv_t^i)P_t^i - wL_t^i - C \qquad (1.16)$$

The experiments run in April 2017 for the final version for the Italian economic journal have the penalty value pv_t^i set to 0.

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

    # the entrepreur sells her production, which is cotributing - via
    # totalActualProductionInA_TimeStep, to price formation
    self.profit=common.price * self.production - self.costs

    common.totalProfit+=self.profit
```

1.3.8 Methods used in Versions 0, 1, 2, 3, 4

1.3.8.1 produce

- The method (or command) `produce`,[32] sent to the **entrepreneurs**, orders them—in a deterministic way, in each unit of time—to produce proportionally to their labour force, obtaining profit Π_t^i, where i identifies the firm and t the time.

L_t^i is the number of workers of 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. π 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:

$$P_t^i = \pi(L_t^i + 1) \tag{1.17}$$

---

[32]Related to Versions 0, 1, 2. 3, 4, 5

The production of the ith 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 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 . . .

1.3.9   Methods used in Versions 1, 2, 3, 4

1.3.9.1   evaluateProfit

- The method (or command) `evaluateProfit`,[33] sent to the `entrepreneurs`, orders 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 use of P_t^i, the actual production of the entrepreneurs, accounts both for the production plan decided with `adaptProductionPlan`, page 26, and for the limits in hiring, if any, as in `hireFireWithProduction`, page 20. The sum of all the actual productions of each entrepreneur is used, as at page 43, in `setMarketPriceV3`.

---

[33]Related to Versions 1, 2, 3, 4.

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.

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 1.3.13.

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:

$$\Pi_t^i = p_t P_t^i - w L_t^i - C \tag{1.18}$$

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

      # the entrepreur sells her production, which is cotributing - via
      # totalActualProductionInA_TimeStep, to price formation
      self.profit=common.price * self.production - self.costs

      common.totalProfit+=self.profit
```

1.3.10 Methods used in Version 0 only

1.3.10.1 evaluateProfitV0

- The method (or command) `evaluateProfitV0`,[34] sent to the `entrepreneurs`, orders 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:

  $$\Pi_t^i = L_t^i(R - w) + u_t^i \tag{1.19}$$

  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)
  ```

1.3.10.2 hireIfProfit

- The method (or command) `hireIfProfit`,[35] sent to the `entrepreneurs`, orders 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):

  $$\Pi_t^i > hiringThreshold \rightarrow hire \tag{1.20}$$

  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.

---

[34]Related to Version 0.
[35]Used in Version 0.

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"
```

1.3.11   Methods used in Version 1 only

1.3.11.1   setMaketPriceV1

- The method (or command) `setMarketPriceV1`,[36] sent to the `WorldState`,
  orders it to evaluate the market clearing price. See below Section 1.3.13 and
  specifically Section 1.3.13.1.

1.3.12   Methods used in Version 2 only

1.3.12.1   toEntrepreneur

- With the method (or command) `toEntrepreneur`,[37] 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`.

---

[36]Related to Version 1.
[37]Related to Version 2.

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
```

1.3.12.2 toWorker

- With the method (or command) `toWorker`,[38] 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
```

---

[38]Related to Version 2.

```
            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
```

1.3.12.3 setMarketPriceV2

- The method (or command) `setMarketPriceV2`,[39] sent to the `WorldState`,
  orders it to evaluate the market clearing price. This method uses two com-
  mon 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 1.3.13 and specifically Section 1.3.13.2.

1.3.13 Other features in scheduling

.

  We also have two more sophisticates structures: the *WorldState* feature and
the *macros*.

- Running a project—if we define the `WorldState.py` file—at the beginning
  of the output, we read:

  `World state has been created.`

  What does it mean?

  The WorldState class interacts with the agents; we use a unique instance of
  the class.

  The variables managed via WordState have to be added, with their methods,
  within the instance of class, with set/get methods for each variable.

---

[39]Related to Version 2.

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

With the `oligopoly` project we made a step ahead, asking to the Worl-State to make a specific calculations about the whole state of the world. This capability has been incorporated in SLAPP since version `1.11` and has been definitively set with the reengineering of WorldState in version `1.33` of SLAPP.

The normal use has in Col. `B` a value and in Col. `C` the method used to set that value into the WorldState; it will be retrieved by the agents using a symmetric get method.[40]

If in Col. `B` we have the expression `computationalUse`[41], the content of Col. `C` is a special method making *world calculations*.

A few examples, with their code, are below.

1.3.13.1 setMarketPriceV1 as in WorldState, with details

- The method (or command) `setMarketPriceV1`,[42] sent to the `WorldState`, orders it to evaluate the market clearing price.

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

$$D_t = \sum_i P_t^i \tag{1.21}$$

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

$$p_t = a + bD_t \tag{1.22}$$

With the planned production coming from a Poisson distribution as in Eq. 1.2, considering ν set to 4, we can set two consistent points (p, D) as $(1, 20)$ and $(0.8, 30)$ obtaining:

$$p_t = 1.4 - 0.02D_t \tag{1.23}$$

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
```

---

[40]These methods have to be implemented by the user, see the example in the *basic* project.

[41]the expression *specialUse* is still working, but it is deprecated.

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

1.3.13.2 setMarketPriceV2, as in WorldState, with details

- The method (or command) `setMarketPriceV2`,[43] sent to the `WorldState`, orders it to evaluate the market clearing price considering each agent behavior.

  Having:

  $$p_t = D_t/O_t \tag{1.24}$$

  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. 41), 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
```

1.3.13.3 setMarketPriceV3, as in WorldState, with details

- The method (or command) `setMarketPriceV3`,[44] 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 the following equations determining the clearing price:

  If the shock Ξ is (≥ 0):

  $$p_t = \frac{D_t(1 + \Xi)}{O_t} \tag{1.25}$$

---

[43]Introduced above as related to Version 2 only.
[44]Introduced above as related to Version 3, 4 and 5.

if the shock Ξ is (< 0):

$$p_t = \frac{D_t/(1 + \Xi)}{O_t} \qquad (1.26)$$

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.

The Ξ parameter is reported in the prologue of the execution as
Total demand relative random shock, uniformly distributed between $-\Xi\%$ and $+\Xi\%$.

As defined above (p. 41), 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

    print "\n------------------------------------"

    if shock >= 0:
      common.totalDemandInPrevious_TimeStep = \
         common.totalPlannedConsumptionInValueInA_TimeStep * \
         (1 + shock)
      common.price= (common.totalPlannedConsumptionInValueInA_TimeStep * \
                 (1 + shock))  \
                 / common.totalProductionInA_TimeStep
      print "Relative shock (symmetric) ", shock0
      print "Set market price to ", common.price

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

    print "------------------------------------\n"
```

1.3.13.4 fullEmploymentEffectOnWages, as in WorldState, with details

Being U_t the unemployment rate at time t, ζ the unemployment threshold to recognize the *full employment* situation, s the proportional increase step (reversible) of the wage level and w_t the wage level at time t (being w_0 the initial level), we have:

$$\begin{cases} w_t = w_0(1+s) \text{ if } U_t \leq \zeta \\ w_t = w_0 \text{ if } U_t > \zeta \end{cases} \tag{1.27}$$

The code in `WorldState.py` is:

```
# shock to wages (full employment case)
def fullEmploymentEffectOnWages(self):

    # wages: reset incumbent action if any
    if common.wageAddendum > 0:
        common.wage -= common.wageAddendum
        common.wageAddendum = 0

    # employed people
    peopleList=common.g.nodes()
    totalPeople=len(peopleList)
    totalEmployed=0
    for p in peopleList:
        if p.employed: totalEmployed+=1
    #print totalPeople, totalEmployed
    unemploymentRate=1. - float(totalEmployed)/ \
                        float(totalPeople)
    if not common.fullEmploymentStatus and \
        unemploymentRate <= common.fullEmploymentThreshold:
            common.wage*=(1 + common.wageStepInFullEmployment)
            common.fullEmploymentStatus=True


    if common.fullEmploymentStatus and \
        unemploymentRate > common.fullEmploymentThreshold:
            common.wage/=(1 + common.wageStepInFullEmployment)
            common.fullEmploymentStatus=False
```

1.3.13.5 randomShocksToWages, as in WorldState, with details

- The method is used only in the model building phase, to verify the sensitivity of the model to changes in wages.

  Being w the wage per employee defined in the setup, so w_1, from $t = 2$ we have:

  – if $u_t \geq 0$

$$w_t = w_{t-1}(1 + ut) \tag{1.28}$$

− if $u_t < 0$

$$w_t = w_{t-1}/(1 + |u_t|) \tag{1.29}$$

with $u_t \sim \mathcal{U}(-k, k)$ and k tentatively set to 0.10 or 10%.

The code in `WorldState.py` is:

```
# random shock to wages (temporary method to experiment with wages)
def randomShockToWages(self):
    k=0.10
    shock= uniform(-k,k)

    if shock >= 0:
     common.wage *= (1.+shock)

    if shock < 0:
     shock *= -1.
     common.wage /= (1.+shock)
```

1.3.13.6 incumbentActionOnWages, as in WorldState, with details

The current number of entrepreneurs H_t is calculated from the network and the previous one H_{t-1} is extracted from the structural dataframe (see `collectStructuralData` at page 11).

The wage level has two components, mutually exclusive:

1. the effects of full employment on wages, as in Section 1.3.13.4;

2. the effect described in this Section about incumbent oligopolists strategically increasing wages to create an artificial barrier against new entrants; the new entrepreneurs suffer temporary extra costs, so for them the wage increment can generate so relevant losses to produce their bankruptcy.

   We have here two levels: K as the (relative) threshold of entrepreneur presence to determine the reaction on wages and k as the relative increment of wages.

Formally, in case 2 above:

$$\begin{cases} w_t = w_0(1 + k) \text{ if } \dfrac{H_t}{H_{t-1}} - 1 > K \\[2ex] w_t = w_0 \text{ if } \dfrac{H_t}{H_{t-1}} - 1 \leq K \end{cases} \tag{1.30}$$

The code in `WorldState.py` is:

```
# incumbents rising wages as na entry barrier
def incumbentActionOnWages(self):

    # current number of entrepreneurs
    peopleList=common.g.nodes()
    nEntrepreneurs=0
    for p in peopleList:
      if p.agType=="entrepreneurs": nEntrepreneurs+=1
    nEntrepreneurs=float(nEntrepreneurs)

    # previous number of entrepreneurs
    nEntrepreneurs0=common.str_df.iloc[-1,0] # indexes Python style

    #print nEntrepreneurs, nEntrepreneurs0

    # wages: reset incumbent action if any
    if common.wageAddendum > 0:
        common.wage -= common.wageAddendum
        common.wageAddendum = 0

    # wages: set
    if nEntrepreneurs/nEntrepreneurs0 - 1 > \
      common.maxAcceptableOligopolistRelativeIncrement:
      common.wageAddendum = common.wage*\
                  common.temporaryRelativeWageIncrementAsBarrier
      common.wage+=common.wageAddendum
```

1.3.13.7 Macros

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

  To know more, use the *SLAPP (Swarm-Like Protocol in Python) Reference Handbook* on line at https://github.com/terna/SLAPP, looking for the item *macros* within the Index.

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