Oligopoly: the Making of the Simulation Model

Marco Mazzoli, Matteo Morini, and Pietro Terna

March 14, 2019

Contents

List of Figures

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

A subset of the model is published in Mazzoli *et al.* (2017); a new publication is forthcoming, with a larger version of the simulation tools. This *Making of* reference offers the full possibility of replication of our results, adopting the AEA Data Availability Policy.[1]

To reproduce the resulta reported in Mazzoli *et al.* (2017);, please use the code version at:
https://github.com/terna/oligopoly/releases/tag/V5 or at
https://github.com/terna/oligopoly/releases/tag/V5bP2_fd[2]or download the zip file of
https://github.com/terna/oligopoly/tree/masterP2, running the project with SLAPP 2.0[3] and controlling that the parameters are those of Table 1 of the paper.

Using SLAPP[4], 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:

[1]https://www.aeaweb.org/journals/policies/data-availability-policy.

[2]The same of V5, underlining the use of Python 2 and adding the output of the data of each firm in each cycle; _fd as firm data.

[3]https://github.com/terna/SLAPP2.

[4]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).

Run *oligopoly* with the Python 3 version of SLAPP.

From its build *20170611* the *oligopoly* project, version *5bPy3*, adopts the PEP8 style. PEP8 contains the Style Guide for Python Code and it is at https://www.python.org/dev/peps/pep-0008/.

Due to this adoption, the reader can notice some aesthetic differences between the code reported here and that listed into the files.

`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`) from 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?`

- We can predefine a default project: if we place *in the main SLAPP folder or in the folder* `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX` a file named `project.txt` containing the path to 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.[5]

- Parameters: we the parameters of the model interactively or within the file `commonVar.py`.

  The program sends several messages about the project parameters, via the file `parameters.py`, in the folder of the project, such as `commonVar.py`.

  The first 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 3.

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

  The general picture is that of Figure 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.[6]

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 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
```

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

[6] 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.

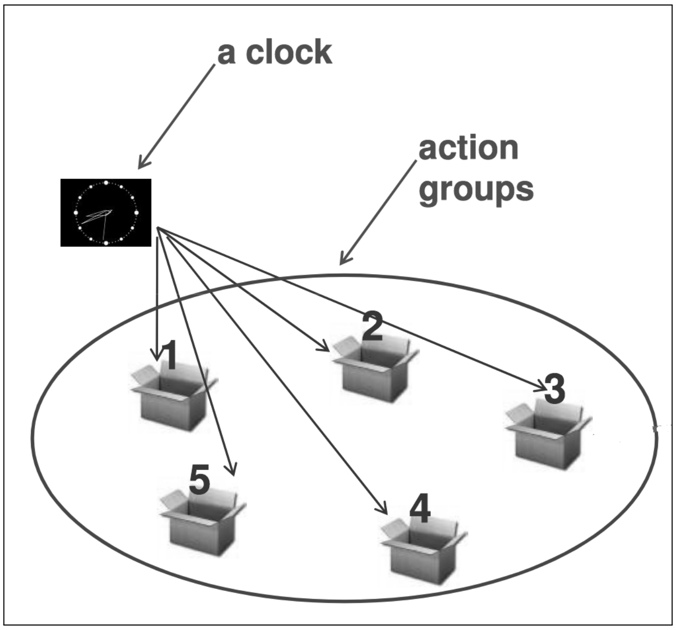

Figure 1: The representation of the schedule

```
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[7], subsection ”The use of files

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

.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;

  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 58.

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 Agents and reset action

The `reset` (see page 15) action, working into the scheduling of the model (Section 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.

An important application of the `reset` function is in Section 4.3.1 and specifically at p. 28.

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.

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.[8]

- 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 2.1 and 2.2.

- The third sequence, operating in a *micro* way, is the more detailed one (see Section 2.3).

2.1 The scheduling mechanism at the level of the Observer

.

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

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

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

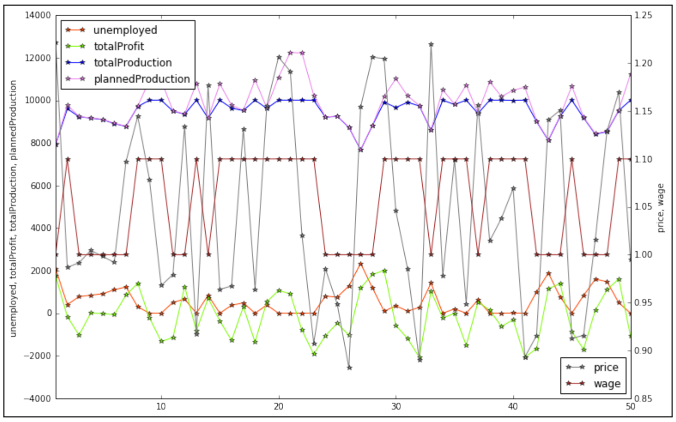

Figure 2: Time series generated by the model. version 4

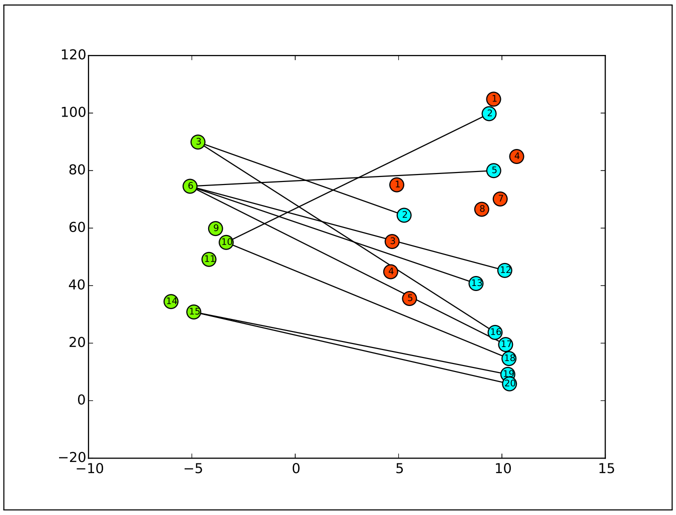

Figure 3: The agents (nodes), with random displacements, and links connecting entrepreneurs and workers

```
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.[9]

– `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 `visualizePlot` and for the final output of the model, when two files of data are generated (with extension .csv and date and time[10] in their names).[9]

– `visualizePlot` update the plot of the time series generated by the model (Figure 2).[11]

– `visualizeNet` update the windows reporting the links connecting entrepreneurs and workers, on a network basis (Figure 3).[11]

– `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 2 and 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

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

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

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

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).

2.1.1 The scheduling mechanism at the level of the Observer: using the *special action* feature to modify the parameters while the model is running

.

We use here the *special action* feature of SLAPP, described in the related Reference Manual (use the Index to find it). In this specific application that feature, we implement the following definition in `commonVar.py`:

```
specialAction = "makeSpecialAction()"
```

with, always in this specific case,

```
file_modPars=False
```

As a consequence, the `specialAction` item in `observerActions.txt` activates the function `makeSpecialAction()` in `oActions.py`.

If a file `modPars.txt` exists, the program asks us in which cycles the modified parameters will be used.

Within the file `modPars.txt` we specify the internal names of the Python variables used as parameters in the model, look for them in `parameters.py`; an example of use is in `specialAction` where the name of the variable is followed by its new value.

`observerActions with specialAction.txt` contains the `specialAction` item; to use that file, you have to rename or copy it as `observerActions.txt`.

2.2 The scheduling mechanism at the level of the Model

.

- The second file—`modelActions.txt`—quoted above at the beginning of Section 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 8.

  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`.[12] `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, we suppose that the acting agent in resetting step would be that with 1 as identifier.

  In our model, we clean the variables:
  `common.totalConsumptionInQuantityInA_TimeStep`,
  `common.HayekianMarketTransactionPriceList_inACycle` to [],
  `totalProductionInA_TimeStep`,
  `totalPlannedConsumptionInValueInA_TimeStep`,
  `totalProfit` and
  `totalPlannedProduction`,
  `ratioSellersBuyersAlreadySet` to *False*
  at the beginning of each step of the time.

  with version 6, a part of the code in `setNewCycleValues()` is related to initialize values fo the Hayekian market, both to set prices and the previous cycles consumption in quantity.

  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")
```

[12] Which is in the "$$slapp$$" folder.

```
if self.number == 1 or not common.agent1existing:

    # introduced with V6
    # V6 reset block starts hene
    # this part is specific of the first Hayekian cycle
    # where it replaces the lack of a previous value in
    # quantity
    # here, if possible, we use the price at t-2
    if common.startHayekianMarket > 1:
       if common.cycle == common.startHayekianMarket:
          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
          # the code above can act only if t>1
          if common.cycle > 1: # if == 1 do nothing
                               # makeProductionPlan acts
                               # establishing directly
                               # self.plannedProduction and the total
                               # common.totalPlannedProduction
             common.totalConsumptionInQuantityInPrevious_TimeStep = \
               common.totalPlannedConsumptionInValueInA_TimeStep  \
               / previuosPrice

    # not in case common.cycle == common.startHayekianMarket == 1
    elif common.cycle > common.startHayekianMarket:
        common.totalConsumptionInQuantityInPrevious2_TimeStep= \
         common.totalConsumptionInQuantityInPrevious1_TimeStep # init. in common
        common.totalConsumptionInQuantityInPrevious1_TimeStep = \
         common.totalConsumptionInQuantityInA_TimeStep
        if common.cycle==common.startHayekianMarket+1:
         common.totalConsumptionInQuantityInPrevious_TimeStep = \
         common.totalConsumptionInQuantityInPrevious1_TimeStep
        if common.cycle > common.startHayekianMarket+1:
         common.totalConsumptionInQuantityInPrevious_TimeStep = \
         common.w*common.totalConsumptionInQuantityInPrevious1_TimeStep +\
         (1-common.w)*common.totalConsumptionInQuantityInPrevious2_TimeStep

    # !!!! here we can use also delayed values, look at !!!! in
    # notesOnHayekianTransformation.md

    common.totalConsumptionInQuantityInA_TimeStep = 0

    # list of all the transaction prices in a cycle of the
    # Hayekian market
    common.HayekianMarketTransactionPriceList_inACycle=[]
    # v6 reset block ends here

    common.totalProductionInA_TimeStep = 0
    common.totalPlannedConsumptionInValueInA_TimeStep = 0

    common.totalProfit = 0
    common.totalPlannedProduction = 0

    # ratio sellers/buyers
    common.ratioSellersBuyersAlreadySet=False

# troubles related idividual variables
if self.agType == "entrepreneurs":
    self.hasTroubles = 0
if self.agType == "workers":
```

```
self.workTroubles = 0
```

– `read_script` orders to the Model to open a new level of scheduling, described in Section 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).

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 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), starting with the sign `#`.

3 Tools

3.1 readingCsvOutput.ipynb

.

The `readingCsvOutput.ipynb` IPython sheet reads the `.csv` output files of the *oligiopoly* runs.

The current version for the production of the book is: `readingCsvOutput_par_corr_BWter.ipynb` and requires the files:

- `partial_corr.py` and
- `labelsPositions.csv`
- both at https://github.com/terna/oligopoly.

3.2 readingCsvOutput_par_corr_BWter.ipynb

.

The `readingCsvOutput_par_corr.ipynb` IPython works as the sheet of Section 3.1 but adding the partial correlation calculations. BW stays for pictures also in black and white.

The code `partial_corr.py` it the complement to `readingCsvOutput_par_corr.ipynb` and comes from https://gist.github.com/fabianp/9396204419c7b638d38f.

The version for the book is `readingCsvOutput_par_corr_BWter.ipynb`.

About partial correlation have a look to http://en.wikipedia.org/wiki/Partial_correlation#Using_linear_regression.

3.3 databaseWizard.ipynb

.

The `databaseWizard.ipynb` IPython sheet reads the `.csv` output files of the *oligiopoly* runs and, via `quickviz`,[13] easily generates graphical representations of the content of those files. `quickviz` is based on `seaborn`[14] and `pandas`.[15]

[13] https://github.com/chmduquesne/quickviz.

[14] https://seaborn.pydata.org.

[15] https://pandas.pydata.org.

4 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,

4.1 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.

4.1.1 Version 0 (GitHub: V0 sub releases tab), 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
```

4.1.2 Version 1, random production as engine (GitHub: release V1&2)

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
```

4.1.3 Version 2 (GitHub: , random production as engine (GitHub: release V1&2)

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
```

4.1.4 Version 3 (GitHub: release 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
```

4.1.5 Version 4 (GitHub: release 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

4.1.6 Version 5, 5b, 5bPy3, 5c, 5c_fd (GitHub: versions V5bPy3, V5c, V5bP2_fd, V5c_fd)

Version 5c_fd adds to 5c the saving with the related output file of data of each firm in each period (production and profit).

The output file has name `date+hour+_firms.csv`.

The individual data of each firm are elaborated via the iPython program `readingCsvOutput.ipynb`,[16] to obtain mean and standard deviations about production and profits. The results are very close, but the dimensionality, to those obtained via the same program using the aggregated time series. The differences are due to the changes in number of entrepreneurs in each period, so the calculations based on the time series use data not always homogeneous.

Version 5c continues the set of small changes introduced to version 5, now adding the capability of changing the parameters of the simulation while the model is running; this capability is based upon the `specialAction` feature of SLAPP, at the level of the observer. See above, section 2.1.1.

Version 5b is related uniquely to a correction in method `adaptProductionPlan`, as in subsection 4.6.1, now modified as in subsection ??. The schedule is unchanged from Version 5 to 5b.

Version 5b3P is exactly the same as version 5b, but revised for Python 3 (SLAPP 3.0 or more).

NB NB NBTo replicate results calculated until May 2017, please use version 5b with SLAPP 2.0.

The differences are coming from a significant novelty in random number use. [17]

With version 5b, we added the possibility of work troubles in firms, via the method entrepreneurs p work troubles, where p is a probability.

[16] Look at Sections 3.1 and 3.2.

[17] Working with the example *basic* (via SLAPP) we can verify that a sequence of "random.random()" numbers has the same content in Python 2 and in Python 3 if "n" in *random.seed(n)* is the same.

Unfortunately, *random.shuffle()* behaves in a different way in the two Python versions, as you can read at http://stackoverflow.com/questions/38943038/difference-between-python-2-and-3-for-shuffle-with-a-given-seed and also, after a call to *shuffle* the successive sequence of random realizations will be different in the two Python versions.

Due to this behavior we cannot reproduce in a full detailed way a run of a project in SLAPP working with Python 2 and with Python 3.

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        planConsumptionInValueV6
workers              planConsumptionInValueV6
WorldState           computationalUse        setMarketPriceV3
entrepreneurs        evaluateProfitV5
entrepreneurs        0.0001                  fireIfProfit
workers              toEntrepreneurV3
entrepreneurs        toWorkerV3
WorldState           computationalUse        fullEmploymentEffectOnWages
WorldState           computationalUse        incumbentActionOnWages
```

4.1.7 Version 6, the simplified Hayekian market (currently, the master in Github, release V6, is under development)

We experiment with the creation of a simplified Hayekian market inside the *Oligopoly* model, remembering Bowles *et al.* (2017), Boettke (1990) and Lewis (2014) as examples of relevant references to the Hayekian market analysis.

The main reference among those above is Bowles *et al.* (2017), absolutely worth to be considered; from there, we report the Hayek quotation:

> [The market is] a system of the utilization of knowledge which nobody can possess as a whole, which ... leads people to aim at the needs of people whom they do not know, make use of facilities about which they have no direct information; all this condensed in abstract signals ... [T]hat our whole modern wealth and production could arise only thanks to this mechanism is, I believe, the basis not only of my economics but also much of my political views (Hayek, 1994, p. 69).

An deep analysis of the simplified Hayekian price mechanism introduce can be found online in the microHayekianMarket[18] GitHub repository.

The Section B.3 of the document online[19] in that web page contains the key information useful to connect that analysis and the solution proposed—in this document—to build the *Oligopoly* model.

[18] https://terna.github.io/microHayekianMarket/

[19] `https://github.com/terna/microHayekianMarket/blob/master/paperLaTeX_folder/microHayekianMarket/microHayekianMarket.pdf`

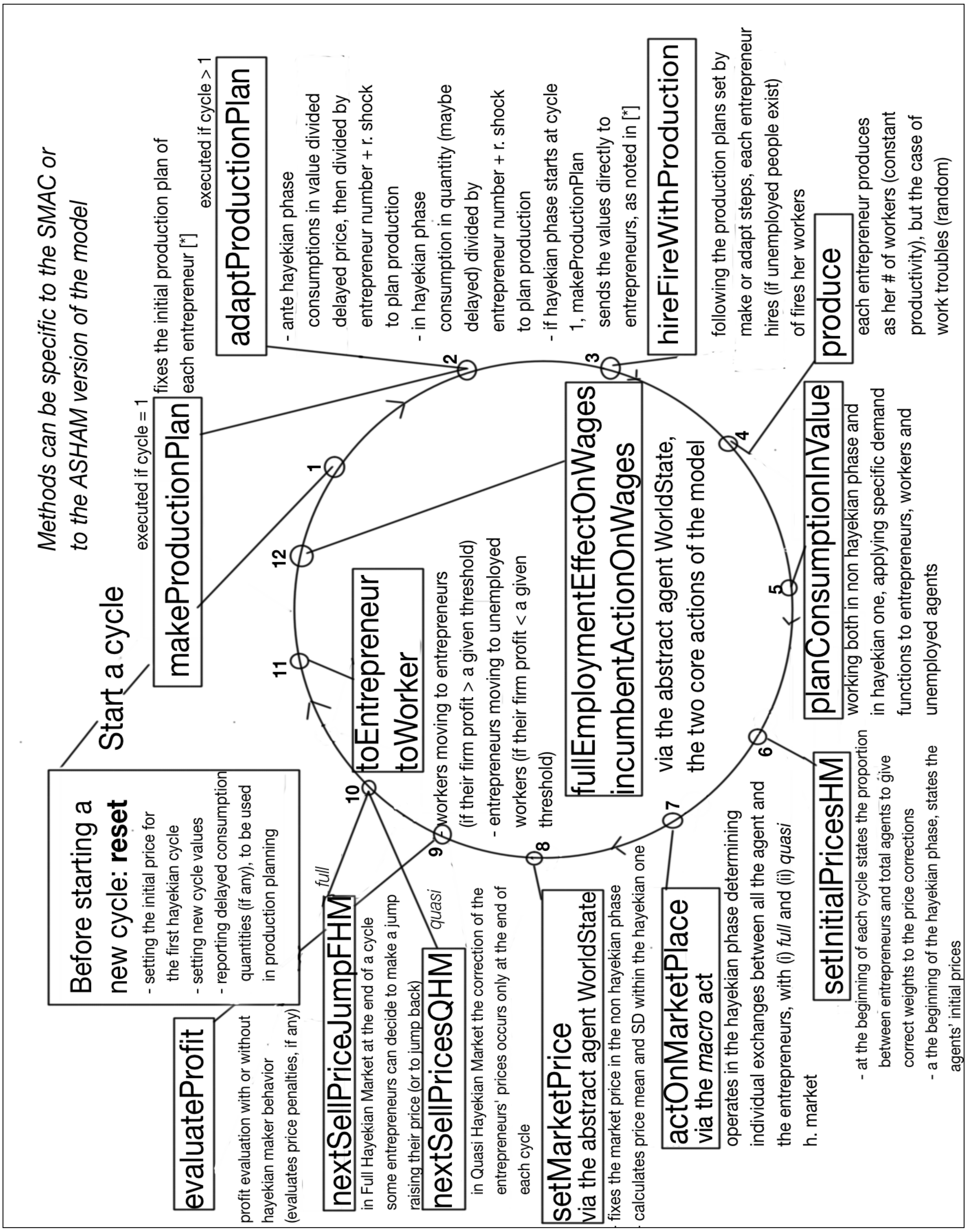

Figure 4: The outline of the model in twelve moves, particularly related to V.6

In Figure 4 we have the sequence of the events with and without the simplified Hayekian market addendum.

To run the V6 with the simplified Hayekian model choice, we need to introduce the feature below in the schedule. We underline that the structure of the V6 schedule is coincident with the results of V5 if the `startHayekianMarket` parameter is greater than the number of expected cycles of the simulation; the `act` macro is useless in that case.

We have two possible ways of interpreting the simplified Hayekian market model:

adopting a *full* Hayekian paradigm: the modification of the prices is continuous, both on the side the buyers (all the agents) and of the side of the sellers (the entrepreneurs); the frequencies of the modifications belong to two highly different scales and as a consequence the amplitude of the price corrections are very different with the rate of $\frac{number\ of\ sellers}{number\ of\ buyers}$ between sellers and buyers (the action `setInitialPricesHM`, reported into the schedule, calculates the ratio between sellers and buyers at the beginning of each period);

adopting a *quasi* Hayekian paradigm: the modification of the prices is here continuous for the buyers, but limited to one correction per cycle (time interval) for each seller.

The switch between *full* and *quasi* Hayekian paradigm operates via the parameter named `hParadigm`,, whose values are `full` or `quasi`. As described in the previuous bullet points, the effect of the choice is that of enabling the way followed by the sellers in fixing their price, after the initial step (described in Section 4.3.6). As a side effect, setting `hParadigm` to any other value, we restrain the sellers from modifying their prices.

In both cases, the function or method `setInitialPricesHM` set the sellers' prices before the starting point of the Hayekian period.

In `schedule6.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          adaptProductionPlanV6
entrepreneurs          hireFireWithProduction
entrepreneurs          0.05                     workTroubles
entrepreneurs          produceV5
entrepreneurs          planConsumptionInValueV6
workers                planConsumptionInValueV6
all                    setInitialPricesHM
macro                  act
WorldState             computationalUse         setMarketPriceV6
```

```
entrepreneurs        evaluateProfitV6
entrepreneurs        nextSellPriceJumpFHM
entrepreneurs        nextSellPricesQHM
workers              toEntrepreneurV6
entrepreneurs        toWorkerV3
WorldState           computationalUse        fullEmploymentEffectOnWages
WorldState           computationalUse        incumbentActionOnWages
```

In `schedule6.xls` we have a second sheet named `act` containing:

```
all                  actOnMarketPlace
all                  actOnMarketPlace
all                  actOnMarketPlace
all                  actOnMarketPlace
all                  actOnMarketPlace
...
```

The number of repetition of the method `actOnMarketPlace` in each cycle is determinate by the number of rows in the macro sheet `act`. Currently we have 30 or more[20], until 100, rows.

In a Hayekian run, the first step (the whole first cycle, with all the sub-steps of the *macro* `act`; the code is reported in Section 4.3.6) of the Hayekian sequence produces a complete output of the actions (*sell* and *buy*) of the agents into the market. The output goes to the file `firstStepOutputInHayekianMarket.csv` in the folder of the `oligopoly` project. The file can be read and, must of all, elaborated with the program `readingFirstStepOutputInHayekinaMarket.ipynb` and it is useful mainly for internal control reasons.

4.2 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).

[20]Increasing the number of row, we test the findings of Appendix B in the note on the Micro Simplified Hayekian Market.

4.3 Methods used in Version 6 only

4.3.1 adaptProductionPlanV6

- While we are in the pre Hayekian period, i.e., in *the warming up* phase of the run of the model, the method `adaptProductionPlanV6` works as the method described in Section 4.6.2.

  Indeed, the easier way to start the *Oligopoly* model with a Hayekian market inside is that of *warming up* the model with the price setting employed until version 5c_fd, i.e., comparing the total offer in quantity and the total demand in value and calculating a clearing price.

  After $k-1$ cycles of *warming*, the last price is used in cycle k, considering it to be the *previous cycle price* that both the entrepreneurs—as producers—and the entrepreneurs and the workers—as consumers—remember and use in their first step, starting to act in a decentralize market, in the Hayekian perspective.

  With $t \geq k$, the methods is operating in the direction of the Hayekian market.

  The entrepreneurs evaluate expected production in each period by dividing the total consumptions of the previous period, measured in quantity (prices are too heterogeneous in a Hayekian situation), by the number of entrepreneurs.

  With:

  - $\varphi_{i,t}$ as individual firm production in t;
  - $C_{n_s,t-1}$ as one of the N_{C_s} buying actions, measured in quantity, of the consumer s at time $t-1$;
  - N_E as the entrepreneur number;
  - u_t as a random addendum (drawn from a uniform distribution to have super-fat tails) representing the difficulty of having correct information about all the buy actions made into the economic system;
  - Q and $1-Q$ as the weights to be attributed to the consumption at time $t-1$ and $t-2$, with $0 \leq Q \leq 1$;

  we have:

$$\varphi_{i,t} = Q\frac{\sum_s \sum_{n_s} C_{n_s,t-1}}{N_E} + (1-Q)\frac{\sum_s \sum_{n_s} C_{n_s,t-2}}{N_E} + u_t \qquad (1)$$

  The code is:

```
# adaptProductionPlanV6
def adaptProductionPlanV6(self):

    # pre hayekian period
    if common.cycle > 1 and common.cycle < common.startHayekianMarket:
        # count of the entrepreneur number
        nEntrepreneurs = 0
        for ag in self.agentList:
            if ag.agType == "entrepreneurs":
                nEntrepreneurs += 1

        # with the scheme of prices until V.5c_fd
        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

        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

    # hayekian period
    if common.cycle >1 and common.cycle >= common.startHayekianMarket:
        #the case common.cycle==1, with common.startHayekianMarket==1, is
        #absorbed by makeProductionPlan

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


        self.plannedProduction = \
                  common.totalConsumptionInQuantityInPrevious_TimeStep \
                  / nEntrepreneurs

        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

    # to record sold production and revenue in hayekian phase
    self.soldProduction=0
    self.revenue=0
```

The internal variable
`common.totalDemandInQuantityInPrevious_TimeStep`
is updated in the following way:

- the internal variable
  `common.totalConsumptioInQuantityInPrevious_TimeStep`
  is set within the `reset` function in `Agent.py` (about `reset`, see Section 1.1), saving in it the value of previous step
  `common.totalConsumptionInQuantityInA_TimeStep`, before resetting it to 0;

  the code is:

```
        elif common.cycle > common.startHayekianMarket:
            common.totalConsumptionInQuantityInPrevious_TimeStep = \
               common.totalConsumptionInQuantityInA_TimeStep
```

  `common.totalConsumptionInQuantityInA_TimeStep` is determined in `actOnMarketPlace` in Section 4.3.6;

- we have a special case when
  `common.cycle == common.startHayekianMarket`
  and the previous value does not exist because `actOnMarketPlace` has still to start to operate;

  in this case we use the previous planned amount of consumption in value,
  `common.totalPlannedConsumptionInValueInA_TimeStep`,
  transformed in quantity using prices at $t-2$ using the scheme of `adaptProductionPlan` (Section 4.6.2), i.e.:

```
        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 the code above can act only from t>1
```

  we underline that `previuosPrice` (as a local variable) can be used in the unique case
  `common.cycle == common.startHayekianMarket`.

- The whole specific code is:

```
# introduced with V6
# V6 reset block starts hene
# this part is specific of the first Hayekian cycle
# where it replaces the lack of a previous values in
# quantity
if common.cycle == common.startHayekianMarket:
    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 the code above can act only from t>1

    common.totalConsumptionInQuantityInPrevious_TimeStep = \
    common.totalPlannedConsumptionInValueInA_TimeStep  \
    / previuosPrice

elif common.cycle > common.startHayekianMarket:
    common.totalConsumptionInQuantityInPrevious_TimeStep = \
       common.totalConsumptionInQuantityInA_TimeStep

common.totalConsumptionInQuantityInA_TimeStep = 0
 # list of all the transaction prices in a cycle of the
# Hayekian market
common.HayekianMarketTransactionPriceList_inACycle=[]
# v6 reset block ends here
```

4.3.2 planConsumptionInValueV6

- The method (or command) `planConsumptionInValueV6`,[21] sent to `workers` or `entrepreneurs`, plans the consumptions in value for the current cycle and its sub-steps, using the parameters detailed in `commonVar.py` file.

  The method is equal to `planConsumptionInValueV5` one, but the conclusion. Indeed, no update of `totalPlannedConsumptionInValueInA_TimeStep` *common* value is made after the conclusion of the *pre-Hayekian* period.

  Consumption behavior of the agent i at time t is defined as:

$$C_{i,t} = a_k + b_k Y_{i,t} + u_{i,t} \tag{2}$$

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

  The individual i can be:

  1. an entrepreneur, with $Y_{i,t} = profit_{i,t-1} + wage$;
  2. an employed worker, with $Y_{i,t} = wage$ and the special[22] case $Y_{i,t} = wc^i_t$, with wc^i_t defined in eq.28;

[21] Related to Version 6
[22] Activated if the *common* value *wageCutForWorkTroubles* is *true*

3. an unemployed workers[23], with $Y_{i,t} = socialWelfareCompensation$.

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`.

Finally, a quota of the unspent consumption capability coming from the past is added to the result. The quota, with a value in the interval $[0, 1]$, has value `reUseUnspentConsumptionCapability`[24] or q^{res} and it is applied to $unspentConsumptionCapability$, giving:

$$C^{+}_{i,t} = C_{i,t} + q^{res} \cdot unspentConsumptionCapability \tag{3}$$

The code in `Agent.py` is:

```
# consumptions
def planConsumptionInValueV6(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 + \
```

[23] In this case, if the random component exceeds the consumption coming from social welfare compensation, we can have negative consumption; in case, in version 6, consumption are set to 0.

[24] Defined in *commonVar.py*.

```
        gauss(0, common.consumptionRandomComponentSD)

# reuse unspent consumption capability
#if self.number==1:
#    print("reuse unspent consumption capability", \
#                         self.unspentConsumptionCapability)
self.consumption += common.reUseUnspentConsumptionCapability * \
                    self.unspentConsumptionCapability

if self.consumption < 0:
    #print('************************************',self.employed, \
    #        self.consumption)
    self.consumption=0


# max cons. in each step of a cycles of the Hayekian phase
self.maxConsumptionInAStep=self.consumption*common.consumptionQuota

# update totalPlannedConsumptionInValueInA_TimeStep
if common.cycle < common.startHayekianMarket or \
   (common.cycle == common.startHayekianMarket and \
   common.startHayekianMarket == 1):
   # the 'or' condition is necessary In the Hayekian perspective,
   # when the start is a cyce 1; the value of
   # common.totalPlannedConsumptionInValueInA_TimeStep is necessary
   # in the warming phase: look at the 'else' within
   # the second block in setInitialPricesHM

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

self.consumptionPlanningInCycleNumber=common.cycle
```

Referring to `totalPlannedConsumptionInValueInA_TimeStep`, the method updates that *common* value only until the starting point of the Hayekian market; on the contrary, the V5 version of the method updates it in any case.

In the *Hayekian* phase, in each step of a cycle we establish a max quantity of consumptions, via a quota introduced interactively in the starting session of the program. This action mimics the action of the households over time in a period.

When the Hayekian market operates, the consumption total amount comes from summing up all the consumption actions in `actOnMarketPlace` method.

4.3.3 setInitialPricesHM, Hayekian Market

- The method (or command) `setInitialPricesHM`,[25], sent to `all`, operates only in the *Hayekian* phase of each run.

[25] Related to Version 6

- The method performs several actions.

  1. In each cycle, while the Hayekian market is working, the method calculates the ratio $\frac{number\ of\ sellers}{number\ of\ buyers}$ between sellers and buyers as requested in adopting the *full* Hayekian paradigm (as described at p.24). This proportion is used to modify sellers' price corrections in the *full* Hayekian paradigm case, in `actOnMarketPlace` method (Section 4.3.6).

     The code in `Agent.py` (setting the value of the `ratioSellersBuyers` is:

```
# set initial sell and buy prices in Hayekian market
def setInitialPricesHM(self):

    # 1 -----------------------------------------
    if common.cycle >= common.startHayekianMarket:
      if not common.ratioSellersBuyersAlreadySet:
          nEntrepreneurs = 0
          for ag in self.agentList:
            if ag.agType == "entrepreneurs":
                      nEntrepreneurs += 1
          nSellers=nEntrepreneurs

          nBuyers=len(self.agentList)

          common.ratioSellersBuyersAlreadySet=True
          common.ratioSellersBuyers=nSellers/nBuyers
          print("Ratio sellers/buyers =",common.ratioSellersBuyers)
          # in setNewCycleValues common.ratioSellersBuyersAlreadySet=False
          # at the beginning of each cycle
```

  2. In the first step of the Hayekian phase, the method states the initial common price: being at time k, we use as starting point the price at time $k-1$ or, if $k>2$, at $k-2$.

     If $k=1$, we use the same structure of the non-Hayekian market, calculating the price as the equilibrium price that would have been created at $t=1$ in the non-Hayekian execution, but outside `WorldState` `setMarketPriceV3` method (Sections 4.6.3 and 4.16.3), to avoid here random shocks.

     The code, always in `Agent.py`, is:

```
    # 2 -----------------------------------------
    if common.cycle == common.startHayekianMarket and \
                    not common.priceWarmingDone:
       # setting the basic price uniquely before the first Hayekian cycle
       common.sellPrice=1000
       common.buyPrice=-1000
       if common.startHayekianMarket>1:
        if len(common.ts_df.price.values) == 1:
          common.buyPrice  = common.sellPrice = \
                  common.ts_df.price.values[-1] # the last price
          #print("Ag.", self.number,"buying at", self.buyPrice,
          #                         "selling at",self.sellPrice)
          # NB the code above can act only if t>1
        if len(common.ts_df.price.values) > 1:
```

```
    common.buyPrice  = common.sellPrice = \
            common.ts_df.price.values[-2] # the second last price
    #print("Ag.", self.number,"buying at", self.buyPrice,
    #                         "selling at",self.sellPrice)
    # NB the code above can act only if t>2
    # NB NB  we set the sellPrice also for workers but we do not
    #        use it
    #        when a worker becomes an entreprenuer she copies the
    #        sell price of the firm she is coming from

else: # case t==1 being common.startHayekianMarket==1
       # look at the equilibrium price that would have been created
       # at t==1 in the non-Hayekian execution

       # in the common.startHayekianMarket == 1 case, when
       # actOnMaketPlace is activated
       # we already have
       # common.totalPlannedConsumptionInValueInA_TimeStep and
       # common.totalProductionInA_TimeStep
       # so, we can calculate
    common.buyPrice  = common.sellPrice = \
                 common.totalPlannedConsumptionInValueInA_TimeStep \
                 / common.totalProductionInA_TimeStep
       # outside WorldState setMarketPriceV3 method, to avoid here
       # random shocks
       # NB NB  we set the sellPrice also for workers but we do not
       #        use it
       #        when a worker becomes an entreprenuer she copies the
       #        sell price of the firm she is coming from

#startingHayekianCommonPrice
print("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
print("starting Hayekian common price",common.buyPrice)
print("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")

common.priceWarmingDone = True
```

3. Calculation of the individual starting prices. The parameter in use here are `initShock` and `initShift`. Here we calculate both `self.buyPrice` and `self.sellPrice`, because each agent can act both as entrepreneur-seller and as buyer.

   – We add a random correction to both the initial prices, to start with two distributions of values. Let `initShock` the internal variable containing the relative value ι as the range of the correction of the individual starting prices in a random uniform way, but with the internal variable `initShift`, having value ν, reasonably[26] in the interval $[-1, 1]$, e.g., 0.10, shifting the range of the whole correction. Our goal here is that of initializing (i) buyer prices mostly *below/above* the starting price and (ii) seller prices mostly *above/below* the starting price, to induce more or less intensively the necessity of individual price corrections to start the initial exchanges.[27]

[26] With very different effects.

[27] At creation time, for each agent *sellPrice* is set to 1000 and *buyPrice* to -1000, so both to

With $p_{S,i}$ as initial sell price of agent i, $p_{B,i}$ as initial buy price of agent i and p_H as starting Hayekian price, we have the eqs.:

∗ if $\varepsilon_{S,i} \geq 0$

$$p_{S,i} = p_H(1 + \varepsilon_{S,i}) \tag{4}$$

∗ if $\varepsilon_{S,i} < 0$

$$p_{S,i} = p_H/(1 + |\varepsilon_{S,i}|) \tag{5}$$

with $\varepsilon_{S,i} \sim \mathcal{U}(-\nu\ \iota, (1-\nu)\ \iota)$.
and

∗ if $\varepsilon_{B,i} \geq 0$

$$p_{B,i} = p_H(1 + \varepsilon_{B,i}) \tag{6}$$

∗ if $\varepsilon_{B,i} < 0$

$$p_{B,i} = p_H(1 + |\varepsilon_{B,i}|) \tag{7}$$

with $\varepsilon_{B,i} \sim \mathcal{U}(-(1-\nu)\ \iota, \nu\ \iota)$

i) If we adopt $\nu = 0.10$ and $\iota = 0.10$, we only have a quite small overlapping of the initial prices correction (correction are never too rational, with someone moving in the wrong direction).
ε corrections will be uniformly distributed between -0.01 and 0.09 for the sellers and between -0.09 and 0.01 for the buyers and, considering the individual prices as reservation prices, we have an offer curve and a demand one crossing on the left side of their graphic representation, as you can see in Figure 5.

ii) If we want not overlapping curves, we can use $\nu < 0$, e.g., $\nu = -0.1$, with $\iota = 0.10$.
ε corrections will be uniformly distributed between 0.01 and 0.11 for the sellers and between -0.11 and -0.01 for the buyers, as in Figure 6.

iii) We can also desire to start in a well balancing situation, with $\nu = 0.5$ and, e.g., $\iota = 0.10|0.20$. ε corrections will be uniformly distributed
(a) between -0.05 and 0.05 for the sellers and between -0.05 and 0.05 for the buyers, as in Figure 7 for $\iota = 0.10$,
or (b) between -0.10 and 0.10 for the sellers and between -0.10 and 0.10 for the buyers, as in Figure 8 for $\iota = 0.20$, doubling the y scale.

---

implausible values, not operational.

In any case, the sellers (entrepreneur) cannot operate into the Hayekian market until their starting price is defined; this information is reported setting to (True) the agent's variable *sellPriceDefined*.

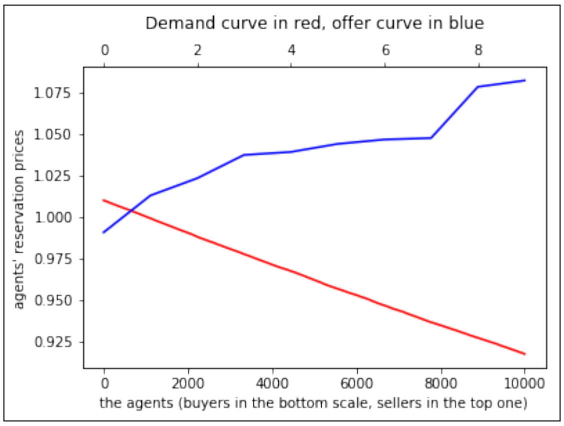

Figure 5: Case i): demand curve with 10,000 agents, ε_B in $[-0.09, 0.01)$; offer curve with 10 agents, ε_S in $[-0.01, 0.09)$

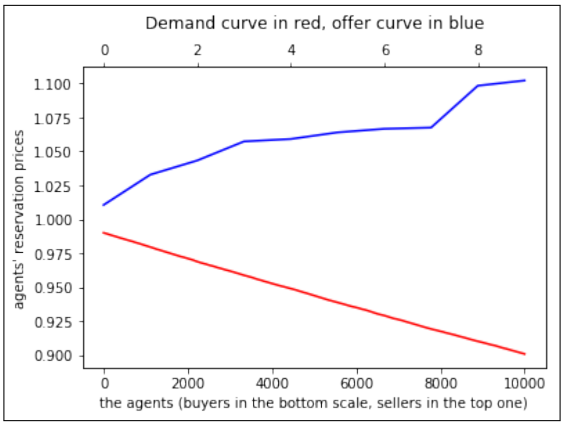

Figure 6: Case ii): demand curve with 10,000 agents, ε_B in $[-0.11, -0.01)$; offer curve with 10 agents, ε_S in $[0.01, 0.11)$

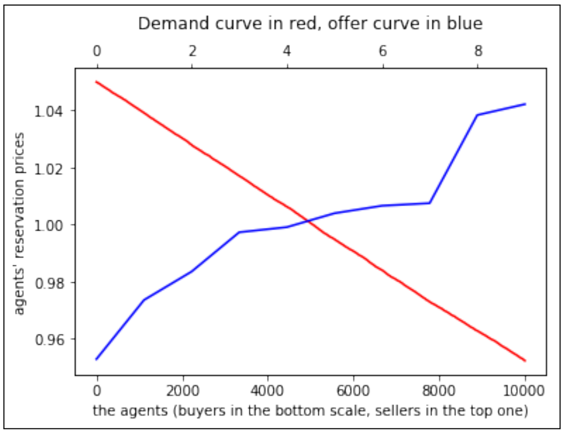

Figure 7: Case iii): (a) demand curve with 10,000 agents, ε_B in $[-0.05, 0.05)$; offer curve with 10 agents, ε_S in $[-0.05, 0.05)$

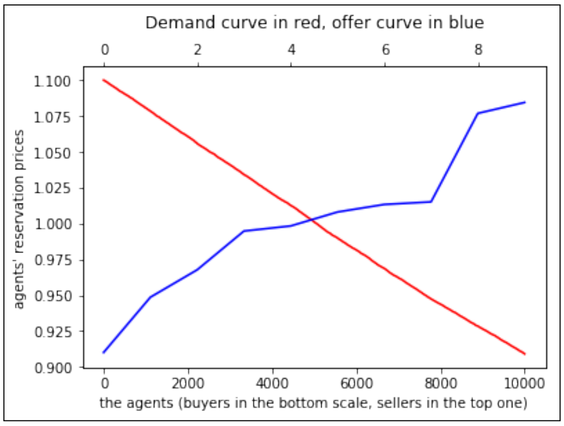

Figure 8: Case iii): (b) demand curve with 10,000 agents, ε_B in $[-0.10, 0.10)$; offer curve with 10 agents, ε_S in $[-0.10, 0.10)$

The code, always in `Agent.py`, is:

```
# 3 -----------------------------------------
# individual starting prices
if common.cycle == common.startHayekianMarket:

   #starting sell price
   self.sellPrice = \
             applyRationallyTheRateOfChange(common.sellPrice,\
                uniform(-common.initShift*common.initShock, \
                        (1-common.initShift)*common.initShock))
   if self.agType=="entrepreneurs":
       print("entrepreneur", self.number, "has initial sell price",\
                             self.sellPrice)
   self.sellPriceDefined=True

   # starting individual buy price
   self.buyPrice = \
             applyRationallyTheRateOfChange(common.buyPrice,\
                uniform((common.initShift-1)*common.initShock, \
                        common.initShift*common.initShock))
```

To generate symmetric effect both for positive and negative rates of change, we use the `applyRationallyTheRateOfChange(base,rate)` function of `Agent.py`. The code is:

```
def applyRationallyTheRateOfChange(base,rate):
        if rate >= 0:
            return base*(1+rate)
        if rate <  0:
            return base/(1+abs(rate))
```

4.3.4 nextSellPriceJumpFHM, Full Hayekian Market

- The method (or command) `nextSellPriceJumpFHM`,[28], sent to `entrepreneurs`, operates only in the *Hayekian* phase of each run.

  It works in the FHM, or Full Hayekian Market, case, i.e., with the *full* Hayekian market paradigm (as described at p.24), modifying the sell price of a specific *entrepreneur-seller* with a jump on the side of the up corrections of her reservation prices. The method operates with a given probability.

  In `commonVar.py` we have the settings of the `jump` size and the `pSize` probability.

  `jump` is a relative value, so $p^J = p(1 + jump)$.

  The probability of jumping operates as a switch, moving the agent to be a jumper and viceversa.

  If a worker of a jumping up entrepreneur moves to be an entrepreneur herself, she starts as a jumper.

[28]Related to Version 6

An entrepreneur making a jump continues anyway to modify her price within the actOnMaketPlaceMethod in each substep.

- The code is:

```
# modify a specific sell price with a jump on the side of the up
# corrections, in full hayekian market
# NB we are at the end of each cycle
def nextSellPriceJumpFHM(self):
    if self.agType != "entrepreneurs": return
    if common.hParadigm=="quasi": return

    if common.pJump != -1 and npr.uniform(0,1)<=common.pJump:
        if self.jump == 0:
            self.jump=common.jump
            self.sellPrice *= 1 + self.jump
            print("entrepreur # ", self.number, \
                  "raises the sell price with a jump")
        else:
            self.sellPrice /= 1 + self.jump
            self.jump=0
            print("entrepreur # ", self.number, \
                  "reduces the sell price with a jump back")
```

4.3.5 nextSellPricesQHM, Quasi Hayekian Market

- The method (or command) `nextSellPricesQHM`,[29], sent to `entrepreneurs`, operates only in the *Hayekian* phase of each run.

  It works in the QHM, or Quasi Hayekian Market, case, i.e., with the *quasi* Hayekian market paradigm (as described at p.24).

  In this case, with the price correction switch `hParadigm` set to `quasi`, the correction of the entrepreneurs' prices occurs only at the end of each cycle.[30]

  We have several possible choices. To select an option we use the variable `quasiHchoice` with with the values: `unsold`, `profit`, and `randomUp`.

  i) Case `quasiHchoice` set to `unsold`. We compare $\frac{sold\ production}{production}$, as sold-ratio, with two thresholds:

     – `soldThreshold1` (e.g., 0.90) and
     – `soldThreshold2` (e.g., 0.99);
     – if the result is less or equal than `soldThreshold1`, a random correction is applied, dividing the price by $1 + |u_1|$ with u_1 drawn from a flat distribution $u_1 \sim \mathcal{U}(decreasingRateRange, 0)$; e.g., $decreasingRateRange = -0.10$;

[29] Related to Version 6

[30] In this case, could be important to start with a well balancing initial situation, as in solution iii) at p. 34.

- if the result is greater or equal than `soldThreshold2`, a random correction is applied, multiplying the price by $1+u_2$ with u_2 drawn from a flat distribution $u_2 \sim \mathcal{U}(0, increasingRateRange)$; e.g., $increasingRateRange = 0.01$;
- between the `soldThreshold1` and `soldThreshold2` values, no correction occurs;
- if `entrepreneursMindIfPlannedProductionFalls` is *True* and the global `plannedProduction` is falling (more than a given threshold `thresholdToDecreaseThePriceIfTotalPlannedPFalls`, e.g., 0.05), entrepreneurs (individually) reduce their prices, with the same mechanism above; this condition supersede the comparison of the sold-ratio with the two thresholds.

ii) Case `quasiHchoice` set to `randomUp`. The logical scheme is the same of the jump in Section 4.3.4 for the *full* implementation but here the huge difference is that we have ho correction within the specific cycle, with the jumped or un-jumped price kept constant.

We use the same parameters of Section 4.3.4. In `commonVar.py` we have the settings of the `jump` size and the `pJump` probability.

`jump` is a relative value, so $p^J = p(1 + jump)$. If `jump`pJump| is set to -1 the random number generation is avoided (for retro-compatibility problems).

The probability of jumping operates as a switch, moving the agent to be a jumper and viceversa.

If a worker of a jumping up entrepreneur moves to be an entrepreneur herself, she starts as a jumper.

An entrepreneur making a jump will keep unchanged the price for a whole cycle (or more, if the opposite action is not turning up).

iii) Case `quasiHchoice` set to `profit`. The entrepreneur decides to raise or to lower her price if the profit is negative. `priceSwitchIfProfitFalls` is a switch with values *"raise"* or *"lower"*. To synthesize the actual difficulty of knowing the demand elasticity, the actual choice between raising o lowering the price is a random one, with 60% of probability to the first choice ("raise") and 40% to the other one if `priceSwitchIfProfitFalls` is set to "raise"; vice-versa, if it set to "lower".

The correction is made with probability `pJump` as above and size `jump`, always as above, calculating:

$p^P = p(1 + jump)$ if the switch is on *"raise"* or $p^P = p/(1 + jump)$ if on *"lower"*.

This action works from $t = 1$.

We have also the parameter `profitStrategyReverseAfterN`.

(a) As default, if positive, it determines the time for a reverse action, lowering the price if raised and viceversa. In the while, no other actions on price are allowed.

(b) If the parameter `profitStrategyReverseAfterN` is greater than the length of the run, the reverse action will never took place and so a unique price correction is possible.

(c) If the parameter is 0, no reverse actions are possible and the *raise* or *lower* actions can be repeated if the profit is negative.

If a worker of a profit-jumping up entrepreneur moves to be an entrepreneur herself, she starts with her `profitStrategyReverseAfterN` counter set to the value of her former company.

Being here the price corrections for the entrepreneurs uniquely at the end of each period, the *full* Hayekian market paradigm (as described at p.24) anyway operates in the first period Hayekian period. The goal of this exception is that of avoiding that the sellers are using the initial random prices for the full length of the first cycle. As we indicated,[30] above, it is important here to start with a well balancing structure of demand and offer curves.

- The resulting code is:

```
# modify sell prices in quasi hayekian market
# NB we are at the end of each cycle
def nextSellPricesQHM(self):
    if self.agType != "entrepreneurs": return
    if common.hParadigm=="full": return

    # hayekian period, "quasi" hayekian paradigm

    # i) considering relative unsold quantity
    if common.hParadigm=="quasi" and common.quasiHchoice=="unsold":
     if common.cycle >= common.startHayekianMarket:

      oldP=self.sellPrice
      if common.cycle >1 and \
       common.entrepreneursMindIfPlannedProductionFalls and \
       common.ts_df.iloc[-1, 3] / common.totalPlannedProduction - 1 >= \
               common.thresholdToDecreaseThePriceIfTotalPlannedPFalls:
       # indexing Python style, pos. -1 is the last one
        self.sellPrice = applyRationallyTheRateOfChange(self.sellPrice,\
                                 uniform(common.decreasingRateRange, 0))
        print(("end of t = %d entrepreneur %d initial production"+\
               " %.2f sold  %.3f \nold price %.3f new price %.3f as "+\
               " total plannedProduction falls") %\
               (common.cycle,self.number,self.production,\
                self.soldProduction,oldP,self.sellPrice))

      else:
        if self.soldProduction/self.production <= common.soldThreshold1:
            self.sellPrice = applyRationallyTheRateOfChange(self.sellPrice,\
```

```
                                uniform(common.decreasingRateRange, 0))
        if self.production/self.production>=common.soldThreshold2:
            self.sellPrice = applyRationallyTheRateOfChange(self.sellPrice,\
                                uniform(0, common.increasingRateRange))

        print(("end of t = %d entrepreneur %d initial production"+\
               " %.2f sold  %.3f \nold price %.3f new price %.3f") %\
               (common.cycle,self.number,self.production,\
                self.soldProduction,oldP,self.sellPrice))
      return

    # ii) considering randomUp
    if common.hParadigm=="quasi" and common.quasiHchoice=="randomUp":
        if common.pJump != -1 and npr.uniform(0,1)<=common.pJump:
                    if self.jump == 0:
                        self.jump=common.jump
                        self.sellPrice *= 1 + self.jump
                        print("entrepreur # ", self.number, \
                              "raises the sell price with a jump")
                    else:
                        self.sellPrice /= 1 + self.jump
                        self.jump=0
                        print("entrepreur # ", self.number, \
                              "reduces the sell price with a jump back")

        return

    # iii) consideirng profit falls to act on price
    if common.hParadigm=="quasi" and common.quasiHchoice=="profit":
      if common.cycle >= common.startHayekianMarket:
           if self.profitStrategyReverseAfterN==0:
               if common.priceSwitchIfProfitFalls=="raise":

                if npr.uniform(0,1)<=0.6:
                    self.priceSwitchIfProfitFalls="raise"
                else:
                    self.priceSwitchIfProfitFalls="lower"

               if common.priceSwitchIfProfitFalls=="lower":

                if npr.uniform(0,1)<=0.4:
                    self.priceSwitchIfProfitFalls="raise"
                else:
                    self.priceSwitchIfProfitFalls="lower"

               if common.pJump != -1 and self.profit <0 and \
                             npr.uniform(0,1)<=common.pJump:
                 if self.priceSwitchIfProfitFalls=="raise":
                    self.sellPrice *= 1 + common.jump
                    print("entrepreur # ", self.number, \
                    "with profit<0, is raising the sell price")
                    self.profitStrategyReverseAfterN=\
                              common.profitStrategyReverseAfterN
                    #  0 means: acting again always possible
                    #  a value > the number of cycles means:
                    #            acting again never possible
                 if self.priceSwitchIfProfitFalls=="lower":
                    self.sellPrice /= 1 + common.jump
                    print("entrepreur # ", self.number, \
                    "with profit<0, is lowering the sell price")
                    self.profitStrategyReverseAfterN=\
                              common.profitStrategyReverseAfterN
```

```
            else:
                self.profitStrategyReverseAfterN-=1
                if self.profitStrategyReverseAfterN==0:
                  if self.priceSwitchIfProfitFalls=="raise":
                     self.sellPrice /= 1 + common.jump
                     print("entrepreur # ", self.number, \
                     "lowering back the sell price")
                  if self.priceSwitchIfProfitFalls=="lower":
                     self.sellPrice *= 1 + common.jump
                     print("entrepreur # ", self.number, \
                     "raising back the sell price")

       return

     # here in error
     print("Using the 'quasi' option in hayekian market:\n",\
           "the",common.quasiHchoice, "value is not one of the\n",
           "valid option (unsold, randomUp, profit)")
     os.sys.exit(1)
```

4.3.6 actOnMarketPlace

- The method (or command) `actOnMarketPlace`,[31] operates only in the *Hayekian* phase of each run.

  Both the `entrepreneurs` and the `workers` are *buyers*, while the *sellers* are uniquely the `entrepreneurs`.

- Search for sellers. In the first step of each multiple call of `actOnMarketPlace` in each cycle, and for each buyer, a seller is chosen from a temporary list locally generated.

- The method `actOnMarketPlace` is included in a macro,.The number of repetitions for each buyer in each cycle (the number of buy attempts) is regulated by the number of repetition into the macro. In each repetition the program shuffles the agents. Each agent has in memory the effect of the action done in the previous repetition within a cycle.

- Sold products are recorded in quantity in `self.soldProduction`, always reset to 0 in any new cycle `adaptProductionPlanV6` step.

  The deal between a buyer and a seller is based on the confrontation of the prices. If $p_{B,i}$, as buy price of agent i, is $\geqslant$ of $p_{S,j}$, as sell price of agent j, the agents exchange, at the seller price, the *min* quantity among the buying

[31] Related to Version 6

capacity of the buyer at that price, its max spending value in each step of a cycle at that price[32] and the actual production residual quantity of the seller.

The revenue of the seller (one of the `entrepreneurs`) is accounted for profit calculations.

- Prices: what price in each action for each agent?
  - The best is *their* price, i.e., the price at which they have concluded a deal (the last one, if more than a deal was made in a same time interval)[33] or that of the last bid (if buyers) or ask (if sellers) they made without concluding a deal. This price is corrected with the rules below.
  - To generate symmetric effect both for positive and negative rates of change, we use the `applyRationallyTheRateOfChange(base,rate)` function as described at p.37.
  - The continuous process of correction of the individual prices, applied to the transactions within each cycle, works in the following way.
    1. We modify the buy price (a random correction) if the last agent negotiation, also if unsuccessful, was on the buy side.
    2. The same on the sell price, but here only in case of `hParadigm` set to `full` (see p.24 and p.32); if this is the case, the sellers' price correction is limited by the factor Z reporting the value of `ratioSellersBuyers`, to account for the different (extremely higher) number of corrections they made in a cycle.
    Let `runningShock` be the internal variable containing the relative value $\tilde{\iota}$ (e.g., 0.05) as the range of the correction or the individual running prices in a random uniform way (but shifted, with the internal variable `runningShift` containing the value of $\tilde{\nu}$ defined in the interval $[0, 0.5]$ e.g., 0.10, with a very low superimposition), with $p_{B,i}$ as buy price of agent i and $p_{S,i}$ as sell price of agent i.
    `runningShock` and `runningShift` are differentiate for buyers and sellers with `runningShockB` `runningShiftB` and `runningShockS` `runningShiftS`.

  The equations are the same as 6, 7, 4 and 5, with $\tilde{\nu}_B$ or $\tilde{\nu}_S$ substituting ν and $\tilde{\iota}_B$ or $\tilde{\iota}_S$ substituting ι. Remember Z as ratio sellers to buyers.

[32]As stated in Section 4.3.2, in each step of a cycle we establish a max quantity of consumptions, via a quota introduced interactively in the starting session of the program.

[33]The event is possible if the agents buy or sell only fractions of their buying or selling quantities, so repeating their actions

- After the starting cycle of the *Hayekian* phase, the agents use two *status* variable (technically: `self.statusB` and `self.statusS`), initialized to 0 and reporting their short term history both as buyers (workers or entrepreneurs) and as sellers (entrepreneurs)..

  Conventions:

  - `self.statusB`
    - 0 means never used;
    - 1 if previous action was a successful buy attempt;
    - -1 if previous action was an unsuccessful buy attempt;
  - `self.statusS`
    - 0 means never used;
    - 1 if previous action was a successful sell attempt;
    - -1 if previous action was an unsuccessful sell attempt.
  - Using the variables introduced above, we adopt $\tilde{\nu}_B = \tilde{\nu}_S = 0.10$ and $\tilde{\imath}_B = \tilde{\imath}_S = 0.05$ as values for the initial experiment.
  - *Buyer* case:
    - if the last transaction succeeded (*statusB* is 1) and $\varepsilon_{B^{down},i} \geq 0$:

      $$p_{B,i_t} = p_{B,i_{t-1}}(1 + \varepsilon_{B^{down},i}) \tag{8}$$

    - if the last transaction succeeded (*statusB* is 1) and $\varepsilon_{B^{down},i} < 0$:

      $$p_{B,i_t} = p_{B,i_{t-1}}/(1 + |\varepsilon_{B^{down},i}|) \tag{9}$$

      with $\varepsilon_{B^{down},i} \sim \mathcal{U}(-(1-\tilde{\nu}_B)\ \tilde{\imath}_B, \tilde{\nu}_B\ \tilde{\imath}_B)$, so mostly negative (the agent tries to decrease its reservation price);
    - if the last transaction failed (*statusB* is -1) and $\varepsilon_{B^{up},i} \geq 0$:

      $$p_{B,i_t} = p_{B,i_{t-1}}(1 + \varepsilon_{B^{up},i}) \tag{10}$$

    - if the last transaction failed (*statusB* is -1) and $\varepsilon_{B^{up},i} < 0$:

      $$p_{B,i_t} = p_{B,i_{t-1}}/(1 + |\varepsilon_{B^{up},i}|) \tag{11}$$

      with $\varepsilon_{B^{up},i} \sim \mathcal{U}(-\tilde{\nu}_B\ \tilde{\imath}_B, (1-\tilde{\nu}_B)\ \tilde{\imath}_B)$, so mostly positive (the agent tries to increase its reservation price).

- *Seller* case:[34]
  - if the last transaction succeeded ($statusS$ is 1) and $\varepsilon_{S^{up},i} \geq 0$:
  $$p_{S,i_t} = p_{S,i_{t-1}}(1 + Z\varepsilon_{S^{up},i}) \tag{12}$$
  - if the last transaction succeeded ($statusS$ is 1) and $\varepsilon_{S^{up},i} < 0$:
  $$p_{S,i_t} = p_{S,i_{t-1}}/(1 + Z|\varepsilon_{S^{up},i}|) \tag{13}$$
  with $\varepsilon_{S^{up},i} \sim \mathcal{U}(-\tilde{\nu}_S\ \tilde{\imath}_S, (1-\tilde{\nu}_S)\ \tilde{\imath}_S)$, so mostly positive (the agent tries to increase its reservation price);
  - if the last transaction failed ($statusS$ is -1) and $\varepsilon_{S^{down},i} \geq 0$:
  $$p_{S,i_t} = p_{S,i_{t-1}}(1 + Z\varepsilon_{S^{down},i}) \tag{14}$$
  - if the last transaction failed ($statusS$ is -1) and $\varepsilon_{S^{down},i} < 0$:
  $$p_{S,i_t} = p_{S,i_{t-1}}/(1 + Z|\varepsilon_{S^{down},i}|) \tag{15}$$
  with $\varepsilon_{S^{down},i} \sim \mathcal{U}(-(1-\tilde{\nu}_S)\ \tilde{\imath}_S, \tilde{\nu}_S\ \tilde{\imath}_S)$, so mostly negative (the agent tries to decrease its reservation price).
- The function `applyRationallyTheRateOfChange(base,rate)`, introduced above, performs in a unique code the two alternative calculations of eqs.: 8 or 9; 10 or 11; 12 or 13; 14 or 15.
- The method also has a tool to generate an optional report about residual consumption in value and unsold production in quantity, at the beginning of each *substep* in a cycle.

  To obtain the report, the variable `checkResConsUnsoldProd` has to be *true* in `commonVar.py`. The report is printed in the regular output flow of the model.

  You can find this tool within the method `actOnMarketPlace`, under the comment *# in each sub step ...* The code is a bit tricky (in each call, the elaboration jumps from an instance of agent to another one).

- The resulting code of the whole method is:

[34] Operating in case of *hParadigm* set to *full*; if set to *quasi* the correction is made once per cycle; if the parameter is set to a value which is neither *full* nor *quasi* we have a side effect, because no corrections will operate on sell prices. In case of the *quasi* scheme, in the first cycle the *full* one is anyway operating, look at p.38.

```
    # all acting as consumers on the market place
    def actOnMarketPlace(self):
        if common.cycle < common.startHayekianMarket: return

        # in each sub step, we show residual consumption and production; the
        # code operates on different agents, but consistently (in each call,
        # the elaboration jumps from an instance of agent to another one)
        if common.checkResConsUnsoldProd:

         #print(self.number)
         if common.withinASubstep:
            common.internalSubStepAgentCounter+=1
            #print('*',common.internalSubStepAgentCounter)
            if common.internalSubStepAgentCounter==len(self.agentList):
                common.withinASubstep=False

         else: # not withinASubstep
            common.withinASubstep=True
            common.internalSubStepAgentCounter=1
            if common.currentCycle != common.cycle:
                  common.currentCycle = common.cycle
                  common.subStepCounter=0
                  common.readySellerList=False
                  print()
            common.subStepCounter+=1
            residualConsumptionCapabilityInValue=0
            residualUnsoldProduction=0
            for anAgent in self.agentList:
                residualConsumptionCapabilityInValue += anAgent.consumption
                if anAgent.agType=="entrepreneurs":
                    residualUnsoldProduction+= \
                    anAgent.production - anAgent.soldProduction
            print(\
"subc. %2d.%3d starts with cons. capab. (v) %.1f and uns. p. (q) %.1f"\
% (common.cycle, common.subStepCounter, residualConsumptionCapabilityInValue,\
                                        residualUnsoldProduction))



        try: common.wr.writerow
        except:
            print("The file firstStepOutputInHayekianMarket.csv was not"+\
                  " created in mActions.py")
            os.sys.exit(1)


        # first call in each cycle, preparing action (only once per cycle)
        #if self.currentCycle != common.cycle:
        if not common.readySellerList:
               #self.currentCycle = common.cycle

           common.readySellerList=True
           # we check that the planning of the consumption has been
           # made for the current cycle
           if self.consumptionPlanningInCycleNumber != common.cycle:
               print('Attempt of using actOnMarketPlace method before'+\
                     ' consumption planning')
               os.sys.exit(1) # to stop the execution, in the calling module
                           # we have multiple except, with 'SystemExit' case

           # create a temporary list of sellers, starting each step (cycle)
           common.sellerList=[]
```

```
    for anAg in self.agentList:
        if anAg.getType() == "entrepreneurs":
            if not anAg.sellPriceDefined:
                print("Inconsistent situation, an active selles"\
                +" has no sell price defined.")
                os.sys.exit(1)
            else: common.sellerList.append(anAg)


# acting (NB self.consumption comes from planConsumptionInValueV6)
# if buying action is possible
#print("cycle",common.cycle,"ag",self.number,"cons val",self.consumption)

if self.consumption > 0:
  if common.sellerList != []:
    # chose a seller
    mySeller=common.sellerList[randint(0,len(common.sellerList)-1)]
    sellerQ=mySeller.production - mySeller.soldProduction
    if sellerQ>0:
      # try a deal
      if self.buyPrice <  mySeller.sellPrice:
         self.statusB=mySeller.statusS=-1
      if self.buyPrice >= mySeller.sellPrice:
         self.statusB=mySeller.statusS= 1

         #print(common.cycle,"entr.",mySeller.number,\
         #  mySeller.production,mySeller.soldProduction,\
         #  mySeller.sellPrice)

         # NB production can be < plannedProduction due to lack of workers

         # consumption in value cannot exceed self.maxConsumptionInAStep
         buyerQ=min(self.consumption/mySeller.sellPrice, sellerQ,\
                    self.maxConsumptionInAStep/mySeller.sellPrice)

         mySeller.soldProduction+=buyerQ
         mySeller.revenue+=buyerQ*mySeller.sellPrice
         self.consumption-=buyerQ*mySeller.sellPrice
         self.unspentConsumptionCapability=self.consumption
         #print("cycle",common.cycle,"ag",self.number,"deal: cons val",\
         #        buyerQ*mySeller.sellPrice,"price",mySeller.sellPrice)
         # saving the price of the transaction
         common.hayekianMarketTransactionPriceList_inACycle.\
                append(mySeller.sellPrice)

         common.totalConsumptionInQuantityInA_TimeStep += buyerQ


    #ouput -  seller has no goods to sell
    elif common.cycle==common.startHayekianMarket:
             common.wr.writerow\
             (["nogoods", "buy", numpy.nan, self.consumption, self.number,\
             "sell", numpy.nan,mySeller.number])

    #output - deal vs. nodeal
    if common.cycle==common.startHayekianMarket:
        if mySeller.statusS==1:
            common.wr.writerow\
            (["deal", "buy", self.buyPrice, self.consumption, self.number,\
            "sell", mySeller.sellPrice,mySeller.number])
        if mySeller.statusS==-1 and mySeller.sellPriceDefined:
            common.wr.writerow\
```

```
            (["nodeal", "buy", self.buyPrice, self.consumption, self.number,\
            "sell", mySeller.sellPrice,mySeller.number])


    # correct running prices

    # if the status is != 0 the agent has already been acting
    if self.statusB == 1:  # buyer case (statusB 1, successful buy attempt,
                       # acting mostly to decrease the reservation price)
      self.buyPrice = applyRationallyTheRateOfChange(self.buyPrice,\
                           uniform(-(1-common.runningShiftB)* \
                                 common.runningShockB, \
                                 common.runningShiftB* \
                                 common.runningShockB))

    if self.statusB == -1: # buyer case (statusB -1, unsuccessful buy attempt,
                       # acting mostly to increase the reservation price)
      self.buyPrice = applyRationallyTheRateOfChange(self.buyPrice,\
                           uniform(-common.runningShiftB* \
                                 common.runningShockB, \
                                 (1-common.runningShiftB)* \
                                 common.runningShockB))

    if mySeller.statusS == 1 and common.hParadigm=="full" or \
                             (common.hParadigm=="quasi" and \
                             common.cycle==common.startHayekianMarket):
                       # seller case (statusS 1, successful sell attempt,
      mySeller.sellPrice = applyRationallyTheRateOfChange(mySeller.sellPrice,\
                                common.ratioSellersBuyers*\
                                uniform(-common.runningShiftS* \
                                 common.runningShockS,
                                 (1-common.runningShiftS)* \
                                 common.runningShockS))

    if mySeller.statusS == -1 and common.hParadigm=="full" or \
                             (common.hParadigm=="quasi" and \
                             common.cycle==common.startHayekianMarket):
                       # seller case (statusS -1, unsuccess. s. attempt,
                       # acting mostly to decrease the reservation price)
      mySeller.sellPrice = applyRationallyTheRateOfChange(mySeller.sellPrice,\
                                common.ratioSellersBuyers*\
                                uniform(-(1-common.runningShiftS)* \
                                 common.runningShockS, \
                                 common.runningShiftS* \
                                 common.runningShockS))

    #print("ag.", self.number, "new prices", self.buyPrice, mySeller.sellPrice)

    # cleaning the situation (redundant)\\
    self.statusB=mySeller.statusS=0

  #output - common.sellerList==[]
  elif common.cycle==common.startHayekianMarket:
       common.wr.writerow\
        (["nosellers", "buy", self.buyPrice, self.consumption, self.number,\
        "sell", numpy.nan,numpy.nan])

#output - self.consumption<=0
elif common.cycle==common.startHayekianMarket:
     common.wr.writerow\
       (["noconsumption", "buy", numpy.nan, self.consumption, self.number,\
       "sell", numpy.nan,numpy.nan])
```

```
#output close
if common.cycle==common.startHayekianMarket+1 and not common.closed:
    common.csvf.close()
    common.closed=True
```

4.3.7 evaluateProfitV6

- The method (or command) `evaluateProfitV6`,[35] 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 61, and for the limits in hiring, if any, as in `hireFireWithProduction`, page 55. The sum of all the actual productions of each entrepreneur is used, as at page 82, 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.

  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 4.16.

  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 4.7.1) and
  - to the creation of barriers against new entrants (Subsection 4.7.3).

  C are extra costs for new entrant firms. They are calibrated to assure the effectivness of the action described in Subsection 4.7.3, 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 4.8.1) the determination of the clearing price, as at page 82, can signal an increase in the equilibrium price, due to the lacking production.

[35]Related to Version 6.

The (relative) shock $\psi_{i,t} > 0$ due to work troubles is defined in subsection 4.8.1.

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 4.4.1).

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.

Here we have the difference with V5. Eqs. 16 and 17 are used in the pre-*hakyean* phase, while eqs. 18 and 19 work in the *Hayekian* one.

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

$$\Pi_t^i = p_t(1 - pv_t^1)P_t^i - (w - \psi_{i,t})(L_t^i - 1) - 1w - C \tag{16}$$

being $1w$ the wage of the entrepreneur.

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

$$\Pi_t^i = p_t(1 - pv_t^i)P_t^i - wL_t^i - C \tag{17}$$

In the *Hayekian* phase we have, with REV_t^i reporting the revenue of firm i ant time t:

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

$$\Pi_t^i = REV_t^i - (w - \psi_{i,t})(L_t^i - 1) - 1w - C \tag{18}$$

being $1w$ the wage of the entrepreneur.

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

$$\Pi_t^i = REV_t^i - wL_t^i - C \tag{19}$$

The experiments run in April 2017 (with V5 of this method) 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 evaluateProfitV6(self):

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

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

    # how many workers, not via productvity due to possible troubles
    # in production

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

    # 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.costs = (common.wage - self.hasTroubles) \
            * (laborForce - 1) \
            + common.wage * 1 +  \
            XC
        # above, common.wage * 1 is for the entrepreur herself

    else:
        self.costs = common.wage * laborForce + \
            XC
    # print "I'm entrepreur", self.number, "costs are",self.costs

    # penalty Value
    pv = 0
    if self.hasTroubles > 0:
        pv = common.penaltyValue

    # V6 - before Hayekian phase
    if common.cycle < common.startHayekianMarket:
       # the entrepreur sells her production, which is contributing - via
       # totalActualProductionInA_TimeStep, to price formation
       self.profit = common.price * (1. - pv) * self.production - self.costs
       print("I'm entrepreur", self.number, "my price is ",
          common.price * (1. - pv))

    # V6 - into the Hayekian phase
    else:
```

```
            self.profit = self.revenue - self.costs
            print("I'm entrepreneur", self.number, "my individual price is ",
               self.sellPrice)


        # individual data collection
        # creating the dataframe
        try:
            common.dataCounter
        except BaseException:
            common.dataCounter=-1

        try:
            common.firm_df
        except BaseException:
            common.firm_df = pd.DataFrame(
                    columns=[
                        'production',
                        'profit'])
            print("\nCreation of fhe dataframe of the firms (individual data)\n")

        common.dataCounter+=1
        #common.firm_df.set_value(common.dataCounter,\ deprecated since pandas 0.21
        col=common.firm_df.columns.get_loc('production')
        common.firm_df.at[common.dataCounter,\
                             col]=self.production
        #common.firm_df.set_value(common.dataCounter,\ deprecated since pandas 0.21
        col=common.firm_df.columns.get_loc('profit')
        common.firm_df.ix[common.dataCounter,\
                             col]=self.profit


        common.totalProfit += self.profit

    # consumptions
    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
```

4.3.8 setMarketPriceV6

- The method (or command) `setMarketPriceV6`,[36] sent to the `WorldState`, orders it to evaluate the market clearing prices in the pre-*Hayekian* phase and successively to record the mean and the standard deviation to the *Hayekian* prices in each cycle. See below Section 4.16 and specifically Section 4.16.4.

4.3.9 toEntrepreneurV6

- The method (or command) `toEntrepreneurV6`,[37] sent to `workers`, the agent, being a worker, decides to become 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`.

  In real world, this kind of decision is a quite rare one, so we have to pass a second more severe 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.

  About the details, look at the method `toEntrepreneurV3` in Section 4.5.1.

  The novelty, in V6, is that—if the Hayekian market is already activated—the hew entrepreneur copies the selling price of the previous firm where she was a worker.

  If the previous firm made a jump in price, also the new born entrepreneur has the info of the jump status.
  The code is:

```
    # to entrepreneurV6
    def toEntrepreneurV6(self):
        if self.agType != "workers" or not self.employed:
            return
        # print float(common.absoluteBarrierToBecomeEntrepreneur)/ \
        #                len(self.agentList)
        if random() <= float(common.absoluteBarrierToBecomeEntrepreneur) / \
                len(self.agentList):
            #myEntrepreneur = gvf.nx.neighbors(common.g, self)[0] with nx 2.0
            myEntrepreneur = list(common.g.neighbors(self))[0]
```

[36] Related to Version 6, jumping from 3 to 6 in numbering the method.
[37] Related to Version 6, jumping from 3 to 6 in numbering the method.

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

    if common.cycle >=common.startHayekianMarket:
      if myEntrepreneur.sellPriceDefined:
        self.sellPrice=myEntrepreneur.sellPrice
        self.jump=myEntrepreneur.jump
        print("with the same sell price of the the previous firm",\
              self.sellPrice)
        self.profitStrategyReverseAfterN=\
                         myEntrepreneur.profitStrategyReverseAfterN
      else:
        print("New entrepreneur cannot copy the price of previous firm")
        os.sys.exit(1)
```

4.4 Methods used in Versions 1, 2, 3, 4, 5, 5b, 5bPy3, 5c, 5c_fd, 6

4.4.1 makeProductionPlan

- The method (or command) `makeProductionPlan`,[38] 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).

  As a definition, the production plan is:

$$\widehat{P}_t^i \sim \ Pois(\nu) \tag{20}$$

  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

[38] Related to Versions 1, 2; in the 3, 4, 5, 5b, 5bPy3 and 5c, 5c_fd and 6 cases, only at time=1

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:

$$\nu = \rho \frac{(N_{workers} + N_{entrepreneurs})}{N_{entrepreneurs}} \tag{21}$$

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
```

4.4.2 hireFireWithProduction

- The method (or command) `hireFireWithProduction`,[39] sent to the `entrepreneurs`, orders them to hire or fire comparing the labor forces required for the pro-

[39] Related to Versions 1, 2, 3, 4, 5, 5b, 5bPy3, 5c, 5c_fd, 6.

duction plan $\widehat{P}_t^i$ and the labor productivity π; we have the required labor force (L_t^i is the current one):

$$\widehat{L}_t^i = \widehat{P}_t^i/\pi \tag{22}$$

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))
            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.[40]

[40] With version 3.6, as we can see at https://docs.python.org/3.6/whatsnew/3.6.html#new-dict-implementation, within the "CPython implementation improvements", the *dict* type has been reimplemented. Specifically, at least in CPython (which is the more diffused Python implementation):

> The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new dict implementation in the language for a few releases be-

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.[41]

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
```

4.5 Methods used in Version 3, 4, 5, 5b, 5bPy3, 5c 5c_fd, 6

4.5.1 toEntrepreneurV3

- With the method (or command) `toEntrepreneurV3`,[42] sent to `workers`, the agent, being a worker, decides to become 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`.

> fore changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5).

So *dict* structures, with Python 3.6 are order-preserving, but *this new implementation is considered an implementation detail and should not be relied upon.*

[41] 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.

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

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

4.5.2 toWorkerV3

- With the method (or command) `toWorkerV3`,[43] an entrepreneur moves to be an unemployed worker if her relative profit (reported to the total of the

[43] Related to Version 3, 4, 5, 6.

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
        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
```

4.6 Methods used in Version 3, 4, 5, 5b, 5bPy3, 5c 5c_fd

4.6.1 adaptProductionPlan until Version 5

- The method (or command) `adaptProductionPlan`,[44] sent to `entrepreneurs`, orders to the i^{th} 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 $u_t^i \geq 0$
  $$\widehat{P}_t^i = \frac{D_{t-1}}{N_{entrepreneurs}}(1 + u_t^i) \tag{23}$$

  – if $u_t^i < 0$
  $$\widehat{P}_t^i = \frac{D_{t-1}}{N_{entrepreneurs}}/(1 + |u_t^i|) \tag{24}$$

  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)

      if shock < 0:
       shock *= -1.
```

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

```
 self.plannedProduction /= (1.+shock)
#print self.number, self.plannedProduction

common.totalPlannedProduction+=self.plannedProduction
```

4.6.2 adaptProductionPlan with Version 5b, 5bPy3, 5c, 5c_fd correction

- The method (or command) `adaptProductionPlan`,[45] sent to `entrepreneurs`, orders to the i^{th} 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)[46]—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 $u_t^i \geq 0$

  $$\widehat{P}_t^i = \frac{\frac{D_{t-1}}{p_{t-2}}}{N_{entrepreneurs}}(1 + u_t^i) \tag{25}$$

  – if $u_t^i < 0$

  $$\widehat{P}_t^i = \frac{\frac{D_{t-1}}{p_{t-2}}}{N_{entrepreneurs}}/(1 + |u_t^i|) \tag{26}$$

  with $u_t^i \sim \mathcal{U}(-v, v)$ and p_{t-2} the lagged price.[47]

  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

[45] Related to Version 5b, 5bPy3 and 5c, 5c_fd.

[46] The missing part until Version 5b/5bPy3/5c/5c_fd 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.

[47] 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/5bPy3/5c/5c_fd 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
```

4.6.3 setMarketPriceV3

- The method (or command) `setMarketPriceV3`,[48] sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 4.16 and specifically Section 4.16.3.

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

4.7 Methods used in Version 4, 5, 5b, 5bPy3, 5c, 5c_fd, 6

4.7.1 fullEmploymentEffectOnWages

The method (or command) `fullEmploymentEffectOnWages`,[49] sent to the `WorldState`, orders it to modify wages accordingly to full employment situation, in a reversible way. See below Section 4.16 and specifically Section 4.16.6.

4.7.2 randomShockToWages

The method (or command) `randomShockToWages`. sent to the `WorldState`, orders it to randomly modify wages. See below Section 4.16 and specifically Section 4.16.5.

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

4.7.3 incumbentActionOnWages

The method (or command) `incumbentActionOnWages`,[50] sent to the `WorldState`, orders it to modify wages for one period, accordingly to the attempt of creating 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 4.16 and specifically Section 4.16.7.

4.8 Methods used in Version 5, 5b, 5bPy3, 5c, 5c_fd, 6

4.8.1 workTroubles

- The method (or command) `workTroubles`,[51] is sent to the `entrepreneurs`, 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

[49] Related to Version 4, 5, 6.
[50] Related to Version 4, 5, 6.
[51] Related to Version 5, 6.

the `schedule.txt` file)[52] 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:

$$Pc_t^i = P_t^i(1 - \psi_{i,t}) \tag{27}$$

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).

$$cw_t^i = w(1 - \psi_{i,t}) \tag{28}$$

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:

```
#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"
```

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

4.8.2 produceV5

- The method (or command) `produceV5`,[53] 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:
  $$P_t^i = \pi(L_t^i + 1) \tag{29}$$

  The production is corrected for work troubles (as in section 4.8.1) calculating the corrected value Pc_t^i with:

  $$Pc_t^i = P_t^i(1 - \psi_{i,t}) \tag{30}$$

  The production (corrected or not) 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:

```
# 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
```

[53] Related to Version 5, 6.

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 ...

4.9 Methods used in Version 5, 5b, 5bPy3, 5c, 5c_fd

4.9.1 planConsumptionInValueV5

- The method (or command) `planConsumptionInValueV5`,[54] sent to `entrepreneurs` or `workers`, produces the following evaluations, detailed in `commonVar.py` file.

  Consumption behavior with

$$C_i = a_k + b_k Y_i + u \tag{31}$$

  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 [55] case $Y_i = wc_t^i$, with wc_t^i defined in eq.28;
  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)
```

[54] Related to Version 5

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

```
        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 + \
                           gauss(0,common.consumptionRandomComponentSD)

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

    self.consumptionPlanningInCycleNumber=common.cycle
```

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 82.

4.9.2 evaluateProfitV5

- The method (or command) `evaluateProfitV5`,[56] 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 61, and for the limits in hiring, if any, as in `hireFireWithProduction`, page 55. The

[56] Related to Version 5.

sum of all the actual productions of each entrepreneur is used, as at page 82, 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.

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 4.16.

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 4.7.1) and
- to the creation of barriers against new entrants (Subsection 4.7.3).

C are extra costs for new entrant firms. They are calibrated to assure the effectivness of the action described in Subsection 4.7.3, 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 4.8.1) the determination of the clearing price, as at page 82, 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 4.8.1.

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 4.4.1).

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:

$$\Pi_t^i = p_t(1 - pv_t^1)P_t^i - (w - \psi_{i,t})(L_t^i - 1) - 1w - C \tag{32}$$

being $1w$ the wage of the entrepreneur.

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

$$\Pi_t^i = p_t(1 - pv_t^i)P_t^i - wL_t^i - C \tag{33}$$

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 evaluateProfitV5(self):

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

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

    # how many workers, not via productvity due to possible troubles
    # in production

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

    # 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.costs = (common.wage - self.hasTroubles) \
            * (laborForce - 1) \
            + common.wage * 1 +  \
            XC
        # above, common.wage * 1 is for the entrepreur herself

    else:
        self.costs = common.wage * laborForce + \
            XC
```

```
# print "I'm entrepreur", self.number, "costs are",self.costs

# penalty Value
pv = 0
if self.hasTroubles > 0:
    pv = common.penaltyValue

# the entrepreur sells her production, which is contributing - via
# totalActualProductionInA_TimeStep, to price formation
self.profit = common.price * (1. - pv) * self.production - self.costs
print("I'm entrepreur", self.number, "my price is ",
      common.price * (1. - pv))


# individual data collection
# creating the dataframe
try:
    common.dataCounter
except BaseException:
    common.dataCounter=-1

try:
    common.firm_df
except BaseException:
    common.firm_df = pd.DataFrame(
            columns=[
                'production',
                'profit'])
    print("\nCreation of fhe dataframe of the firms (individual data)\n")

common.dataCounter+=1
#common.firm_df.set_value(common.dataCounter,\ deprecated since pandas 0.21
col=common.firm_df.columns.get_loc('production')
common.firm_df.at[common.dataCounter,\
                         col]=self.production
#common.firm_df.set_value(common.dataCounter,\ deprecated since pandas 0.21
col=common.firm_df.columns.get_loc('profit')
common.firm_df.ix[common.dataCounter,\
                         col]=self.profit



common.totalProfit += self.profit
```

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

4.10.1 produce

- The method (or command) `produce`,[57] 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.

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

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:

$$P_t^i = \pi(L_t^i + 1) \tag{34}$$

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:

```
# 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 ...

4.11 Methods used in Versions 0, 1, 2, 3, 4, 5, 5b, 5bPy3, 5c, 5c_fd

4.11.1 fireIfProfit

- The method (or command) `fireIfProfit`,[58] 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[59]), 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):

$$\Pi_t^i < firingThreshold \rightarrow fire \tag{35}$$

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

[58] Used in Versions 0, 1, 2, (temporary) 3, 4 and 5, 5b, 5bPy3, 5c, 5c_fd.

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

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

4.12.1 evaluateProfit

- The method (or command) `evaluateProfit`,[60] 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 61, and for the limits in hiring, if any, as in `hireFireWithProduction`, page 55. The sum of all the actual productions of each entrepreneur is used, as at page 82, 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.

  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 4.16.

  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:

$$\Pi_t^i = p_t P_t^i - wL_t^i - C \tag{36}$$

  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
```

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

```
# 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 contributing - via
# totalActualProductionInA_TimeStep, to price formation
self.profit=common.price * self.production - self.costs

common.totalProfit+=self.profit
```

4.12.2 planConsumptionInValue

- The method (or command) `planConsumptionInValue`,[61] 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, as an example, into Table 1 of Mazzoli *et al.* (2017). The description below is unique for both the cases.

  The resulting consumption behavior if the agent i at time t is:

  $$C_t^i = a_j + b_j Y_t^i + u \tag{37}$$

  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 $Y_t^i = P_{t-1}^i + w_t$;
  - case $j = 2$: an employed worker at time t, with $Y_i = w$ and the special [62] case $Y_t^i = wc_t^i$, with wc_t^i defined in eq. 28;
  - 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:

[61] Related to Version 2, 3, 4

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

```
# 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 82.

4.13 Methods used in Version 2 only

4.13.1 toEntrepreneur

- With the method (or command) `toEntrepreneur`,[63] sent to `workers`, the agent, being a worker, decides if to become 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

[63] Related to Version 2.

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
```

4.13.2 toWorker

- With the method (or command) `toWorker`,[64] an entrepreneur moves to be an unemployed worker if its profit at time t is $\leq$ a given *threshold* in t. The threshold is retrieved from the variable `thresholdToWorker`.

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

  The code in `Agent.py` is:

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



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

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

            common.g.remove_edge(self, aWorker)
```

[64] Related to Version 2.

```
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
```

4.13.3 setMarketPriceV2

- The method (or command) `setMarketPriceV2`,[65] 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 4.16 and specifically Section 4.16.2.

4.14 Methods used in Version 1 only

4.14.1 setMaketPriceV1

- The method (or command) `setMarketPriceV1`,[66] sent to the `WorldState`, orders it to evaluate the market clearing price. See below Section 4.16 and specifically Section 4.16.1.

4.15 Methods used in Version 0 only

4.15.1 evaluateProfitV0

- The method (or command) `evaluateProfitV0`,[67] 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 $L_t^i = P_t^i/\pi$

[65] Related to Version 2.
[66] Related to Version 1.
[67] Related to Version 0.

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:

$$\Pi_t^i = L_t^i(R - w) + u_t^i \tag{38}$$

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)
```

4.15.2 hireIfProfit

- The method (or command) `hireIfProfit`,[68] 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):

$$\Pi_t^i > hiringThreshold \rightarrow hire \tag{39}$$

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
```

[68] Used in Version 0.

```
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"
```

4.16 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.

  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 WorlState 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.[69]

If in Col. `B` we have the expression `computationalUse`[70], the content of Col. `C` is a special method making *world calculations*.

A few examples, with their code, are below.

4.16.1 setMarketPriceV1 as in WorldState, with details

- The method (or command) `setMarketPriceV1`,[71] sent to the `WorldState`, orders it to evaluate the market clearing price.

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

$$D_t = \sum_i P_t^i \tag{40}$$

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

$$p_t = a + bD_t \tag{41}$$

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

$$p_t = 1.4 - 0.02D_t \tag{42}$$

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
```

[69]These methods have to be implemented by the user, see the example in the *basic* project.
[70]the expression *specialUse* is still working, but it is deprecated.
[71]Introduced above as related to Version 1 only.

4.16.2 setMarketPriceV2, as in WorldState, with details

- The method (or command) `setMarketPriceV2`,[72] sent to the `WorldState`, orders it to evaluate the market clearing price considering each agent behavior.

  Having:

  $$p_t = D_t/O_t \tag{43}$$

  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. 78), 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
```

4.16.3 setMarketPriceV3, as in WorldState, with details

- The method (or command) `setMarketPriceV3`,[73] 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):

  $$p_t = \frac{D_t(1+\Xi)}{O_t} \tag{44}$$

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

if the shock Ξ is (< 0):

$$p_t = \frac{D_t/(1+\Xi)}{O_t} \tag{45}$$

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. 78), 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:
        totalDemand = \
            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)
        # common.totalDemandInPrevious_TimeStep is necessary for
        # adaptProductionPlan and adaptProductionPlanV6
        common.totalDemandInPrevious_TimeStep=totalDemand

    if shock < 0:
        shock *= -1.  # always positive, being added to the denominator
        totalDemand = \
            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)
        # common.totalDemandInPrevious_TimeStep is necessary for
        # adaptProductionPlan and adaptProductionPlanV6
        common.totalDemandInPrevious_TimeStep=totalDemand

    print("-------------------------------------\n")
```

4.16.4 setMarketPriceV6, as in WorldState, with details

- The method (or command) `setMarketPriceV6`,[74] sent to the `WorldState`, orders it to evaluate the market clearing price considering each agent behavior and *an external shock, potentially large* in the pre-*Hayekian* phase and successively to record the mean and the standard deviation to the *Hayekian* prices in each cycle.

  About the pre-*Hayekian* phase look at Section 4.16.3. In the code reported here, the new part uses the function coded as:

```
def checkHayekianPrices(a):
    # list a not empty
    if a!=[]: m = statistics.mean(a)
    else: m = -100 # -100 will not appear in graphs
    # and with at least one element
    if len(a)>1: sd = statistics.stdev(a)
    else: sd=-100 # -100 will not appear in graphs
    return (m,sd)
```

  strictly related to the code of the method `actOnMarketPlace` of `agent.py`.

  The resulting code in `WorldState.py` is:

```
    # set market price V6
    def setMarketPriceV6(self):

        print("\n-------------------------------------")

        if common.cycle < common.startHayekianMarket:

           shock0 = random.uniform(-common.maxDemandRelativeRandomShock,
                                   common.maxDemandRelativeRandomShock)
           shock = shock0

           if shock >= 0:
               totalDemand = \
                   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)
               # common.totalDemandInPrevious_TimeStep is necessary for
               # adaptProductionPlan and adaptProductionPlanV6
               common.totalDemandInPrevious_TimeStep=totalDemand

           if shock < 0:
               shock *= -1.  # always positive, being added to the denominator
               totalDemand = \
                   common.totalPlannedConsumptionInValueInA_TimeStep / \
                   (1 + shock)
               common.price=(common.totalPlannedConsumptionInValueInA_TimeStep \
```

[74] Introduced above as related to Version 6, jumping in numbering from 3 to 6.

```
                        /(1 + shock))  \
                        / common.totalProductionInA_TimeStep
        print("Relative shock (symmetric) ", shock0)
        print("Set market price to ", common.price)
        # common.totalDemandInPrevious_TimeStep is necessary for
        # adaptProductionPlan and adaptProductionPlanV6
        common.totalDemandInPrevious_TimeStep=totalDemand

# Hayekian phase
else:
    (common.price, common.hPriceSd)=checkHayekianPrices(\
           common.HayekianMarketTransactionPriceList_inACycle)

    print("Hayekian phase (NA as not available values)")
    if common.price != -100:    print("Mean price ",common.price)
    else:                       print("Mean price NA")
    if common.hPriceSd != -100: print("Mean price s.d.",common.hPriceSd)
    else:                       print("Mean price s.d. NA")

print("------------------------------------\n")
```

4.16.5 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 $u_t \geq 0$
  $$w_t = w_{t-1}(1 + ut) \tag{46}$$
  - if $u_t < 0$
  $$w_t = w_{t-1}/(1 + |u_t|) \tag{47}$$

  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)
```

4.16.6 fullEmploymentEffectOnWages, as in WorldState, with details

As a first step, the wage level is reset to its base value, but the case of a wage raise in this same cycle (coming from another procedure). In the same cycle we can have different wage raises, with cumulative effects.

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:

$$\begin{cases} w_t = w_0(1+s) \text{ if } U_t \leq \zeta \\ w_t = w_0 \text{ if } U_t > \zeta \end{cases} \tag{48}$$

The code in `WorldState.py` is:

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

    # wages: reset wage addendum, if any
    # excluding the case of a raise made in this cycle by another procedure
    if  common.wageCorrectionInCycle != common.cycle:
        common.wage = common.wageBase

    # 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 unemploymentRate <= common.fullEmploymentThreshold:
        common.wage *= (1 + common.wageStepInFullEmployment)
        common.wageCorrectionInCycle=common.cycle
```

4.16.7 incumbentActionOnWages, as in WorldState, with details

As a first step, the wage level is reset to its base value, but the case of a wage raise in this same cycle (coming from another procedure). In the same cycle we can have different wage raises, with cumulative effects.

The current number of entrepreneurs H_t^E is calculated from the network at the end (so the superscript ^{E}) of a cycle and the previous values H_i^B are extracted from the structural dataframe, containing the data at the beginning of each period (see `collectStructuralData` at page 13). In H_i^B the superscript B means: at the beginning of the cycle i. Pay attention, $H_i^B = H_{i-1}^E$. Consistently, $\frac{H_t^E}{H_t^B} - 1$ measures the relative increment/decrement of the number of the entrepreneurs in cycle t.

The wage level has two components, mutually exclusive:

1. the effects of full employment on wages, as in Section 4.16.6;
2. the effect described in this Section about the actions of the incumbent oligopolists, which are 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.

   How to measure the increment in the number of the entrepreneurs?

   (a) If we compare H_t^E and H_t^B, we have a simple direct measure,

   (b) but if we have a continuous series of small increments—all with $\frac{H_t}{H_t^B} - 1 \leq K$, so under the threshold—the overall effect is invisible.

In case 2a:

$$\begin{cases} w_t = w_0(1+k) \text{ if } \dfrac{H_t^E}{H_t^B} - 1 > K \\ w_t = w_0 \text{ if } \dfrac{H_t^E}{H_t^B} - 1 \leq K \end{cases} \tag{49}$$

In case 2b:

first of all, we define the *reference level*, or R^L, as a dynamic value, calculating, at any time t:[75]

$$\begin{cases} R_t^L = H_t^B \text{ if } \dfrac{H_{t-1}^E}{R_{t-1}^L} - 1 > K \\ R_t^L = R_{t-1}^L \text{ otherwise} \end{cases} \tag{50}$$

[75] We could consider to have the first condition in the form: $R_t^L = H_t^B$ if $H_t^B - H_{t-1}^B \leq 0$ *or* $\frac{H_{t-1}^E}{R_{t-1}^L} - 1 > K$.

remembering that in $t = 1$ (starting time), $R^L_0 = H^B_1$ and $H^B_0 = H^E_0 = H^B_1$. As a consequence, always in case 2b:

$$\begin{cases} w_t = w_0(1+k) \text{ if } \dfrac{H^E_t}{R^L_t} - 1 > K \\ w_t = w_0 \text{ if } \dfrac{H^E_t}{R^L_t} - 1 \leq K \end{cases} \tag{51}$$

The code in `WorldState.py` is:

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

    # wages: reset wage addendum, if any
    # excluding the case of a raise made in this cycle (by another procedure)
    if  common.wageCorrectionInCycle != common.cycle:
        common.wage = common.wageBase
        common.wageAddendum=0 # for the final print if in use


    # E and B final letters in the name are consistent with the symbols
    # in Section "incumbentActionOnWages, as in WorldState, with details"
    # current number of entrepreneurs
    peopleList = common.g.nodes()
    nEntrepreneursE = 0
    for p in peopleList:
        if p.agType == "entrepreneurs":
            nEntrepreneursE += 1
    nEntrepreneursE = float(nEntrepreneursE)

    # no cumulative measure
    # as in the Section incumbentActionOnWages, as in WorldState, with details
    # in the Reference
    if not common.cumulativelyMeasuringNewEntrantNumber:
      # previous number of entrepreneurs
      # values in str_df at the beginning of each cycle (B as beginning)
      nEntrepreneursB = common.str_df.iloc[-1, 0]  # indexing Python style
                                                   # pos. -1 is the last one

      # print nEntrepreneurs, nEntrepreneurs0

      # wages: set
      if nEntrepreneursB >= 1:
          if nEntrepreneursE / nEntrepreneursB - 1 > \
             common.maxAcceptableOligopolistRelativeIncrement:
              common.wageAddendum = common.wage *\
                   common.temporaryRelativeWageIncrementAsBarrier
              common.wage += common.wageAddendum
              common.wageCorrectionInCycle=common.cycle
    # cumulative measure
    # as in the Section incumbentActionOnWages, as in WorldState, with details
    # in the Reference
    if common.cumulativelyMeasuringNewEntrantNumber:
      #print("///////// ","common.cycle",common.cycle)
      if common.cycle == 1:
            # values in str_df at the beginning of each cycle
```

```
      nEntrepreneursB_1     = common.str_df.iloc[-1, 0]#indexing Py. style
      nEntrepreneursB       = common.str_df.iloc[-1, 0]# pos. -1 is
      nEntrepreneursE_1     = common.str_df.iloc[-1, 0]
      ReferenceLevel_1      = common.str_df.iloc[-1, 0]# the last one
      common.ReferenceLevel = common.str_df.iloc[-1, 0]
                            # common to avoid a reference error
else:
      nEntrepreneursB_1 = common.str_df.iloc[-2, 0]#indexing Py. style
      nEntrepreneursB   = common.str_df.iloc[-1, 0]
      nEntrepreneursE_1 = common.str_df.iloc[-1, 0]
      ReferenceLevel_1  = common.ReferenceLevel

#if nEntrepreneursB - nEntrepreneursB_1 <= 0 or \
if  nEntrepreneursE_1 / ReferenceLevel_1 - 1 > \
         common.maxAcceptableOligopolistRelativeIncrement:
            common.ReferenceLevel = nEntrepreneursB
else:
            common.ReferenceLevel = ReferenceLevel_1

# wages: set
if common.ReferenceLevel >= 1:
    if nEntrepreneursE / common.ReferenceLevel - 1 > \
       common.maxAcceptableOligopolistRelativeIncrement:
        common.wageAddendum = common.wage *\
               common.temporaryRelativeWageIncrementAsBarrier
        common.wage += common.wageAddendum
        common.wageCorrectionInCycle=common.cycle

"""
print("/// ","nEntrepreneursE",nEntrepreneursE)
print("/// ","nEntrepreneursE_1",nEntrepreneursE_1)
print("/// ","nEntrepreneursB",nEntrepreneursB)
print("/// ","nEntrepreneursB_1",nEntrepreneursB_1)
print("/// ","ReferenceLevel",common.ReferenceLevel)
print("/// ","ReferenceLevel_1",ReferenceLevel_1)
print("/// ","wageAddendum",common.wageAddendum)
"""
```

4.16.8 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 https://www.palgrave.com/gp/book/9781137339805

Boettke, P. J. (1990). *The theory of spontaneous order and cultural evolution in the social theory of FA Hayek*. In «Cultural Dynamics», vol. 3(1), pp. 61–83.

Bowles, S., Kirman, A. and Sethi, R. (2017). *Retrospectives: Friedrich Hayek and the Market Algorithm*. In «Journal of Economic Perspectives», vol. 31(3), pp. 215-30.
URL http://www.aeaweb.org/articles?id=10.1257/jep.31.3.215

Hayek, F. A. (1994). *Hayek on Hayek: an autobiographical dialogue*, vol. edited by Stephen Kresge and Leif Wenar. University of Chicago Press:.

Lewis, P. (2014). *Hayek: from Economics as Equilibrium Analysis to Economics as Social Theory*. In R. Garrison and N. Barry, eds., *Elgar Companion to Hayekian Economics*. Edward Elgar Publishing, pp. 195–223.
URL https://ssrn.com/abstract=2546259

Mazzoli, M., Morini, M. and Terna, P. (2017). *Business Cycle in a Macromodel with Oligopoly and Agents' Heterogeneity: An Agent-Based Approach*. In «Italian Economic Journal», pp. 1–29.
URL http://rdcu.be/tlE6

Index