DOI: https://doi.org/10.14393/REE-v37nesp. ago.a2022-66702
Instituto de Economia e Relações Internacionais – Universidade Federal de Uberlândia

economia
ensaios

An implementation on Python of the Classical Nelson and Winter Models

Uma implementação em Python dos modelos clássicos de Nelson e Winter

José Bruno do Nascimento Clementino [a]
Martin Harry Vargas Barrenechea [b]

Resumo: Este trabalho trata da implementação computacional em Python da família de modelos clássicos de Concorrência Schumpeteriana desenvolvidos por Nelson e Winter. Apresentamos como ponto de partida um modelo estático simples duma indústria como semente do desenvolvimento de modelos dinâmicos evolucionários, em conjunto com o desenvolvimento paralelo entre o desenvolvimento teórico e a implementação computacional utilizando as capacidades de herança presentes no Python, de modo a contemplar inovação, dinâmicas de sistema, políticas adaptativas de inovação e patentes. Os resultados do trabalho original são comparados aos resultados encontrados ao longo do desenvolvimento desse trabalho.

Palavras-chave: Dinâmica industrial, Modelagem baseada em agentes, Economia evolucionáriam, Python
Classificação JEL: B52, C89, L10

Abstract: This paper addresses the computational implementation in Python of the family of classical Schumpeterian Competition models developed by Nelson and Winter. A simple one-shot model is presented as a seed for the development of evolutionary dynamic models. Then, a parallel development between theory and computational implementation using the inheritance capabilities present in Python is done, in order to contemplate innovation, system dynamics, adaptive innovation policies and patents. The results of the original work are compared to the results found throughout the development of this work.

Keywords: Industrial Dynamics, Agent Based Models, Evolutionary Economics, Python
JEL Classification: B52, C89, L10

[a] Mestrando em Economia aplicada, Programa de Pós-Graduação em Economia Aplicada (PPEA), UFOP. E-mail: jose.clementino@aluno.ufop.edu.br
[b] Professor do Programa de Pós-Graduação em Economia Aplicada (PPEA), UFOP. E-mail: mbarrenechea@ufop.br

1. Introduction

One of the most cited models in evolutionary economics is the model of Nelson and Winter that is described in chapter 12 of the seminal Nelson and Winter (1982) book, named here as NW82. Several models were developed later to take into consideration critics and extend the generality of the model. Such modifications are described in Winter (1984), which is named NW84, and in Winter (1993), named NW93. In the first model, the investment policy in R&D is adaptive and allows the entrance and exit of firms. The second model incorporates the analysis of patent systems.

There have been several efforts to systematize the evolutionary models developed by Nelson and Winter NW, one of the most important described in Andersen et al. (1996) and Andersen (2001) where several models are algorithmically developed, such efforts were condensed in the development of a software called LSD. Valente and Andersen (2002) show the implementation of the NW82 model in this software. However, there is no available version of the more complex models NW84 and NW93.

Between the descriptions of a model and its implementation a gap that must be explored exists. A full implementation of all models while also explaining the process to the reader, describing its inner workings and assumptions and showing its representation in code, is the product of this work. We intend to show that the tools provided by Object Oriented Programming can be used to develop the NW models constructively. To do so, we implement these models in Python, inspired by the bottom-up procedure (see Isaac (2008)). A very simple model is gradually extended through inheritance until the most complex model NW93 is developed. By using inheritance, we show that such programming device describes in a very natural way the developments of the NW Evolutionary models.

We have chosen Python, a general-purpose programming language. One if its advantages is its readability, since it uses whitespace and indentation in the definition of its functionalities: declaring functions, control flow (defining if and else clauses), loops, Oriented-Object Programming (OOP) etc. Because of this characteristic of the language, reading code is almost like reading plain English. Also, readability is associated with an ease of use: if code is easy to read and, consequently, to understand, then the implementation of an extension tends to be easy as well. Thus, the implementation on Python of these models is not only a thorough explanation of the choices made for its implementation and explanation of its inner workings, but also an invitation to the reader to comprehend it, improve it (if possible), and to extend it in order to comprehend other relevant extensions that the reader might see fit.

Our implementation of the models is mostly based on pure Python, except for two widely known modules: **NumPy**, for numerical computation and array manipulation; random, for random value generation; and **Matplotlib**, for plotting. While we are aware that there are dedicated platforms for the implementation of agent-based models, such as

NetLogo, LSD and HASH (a promising new platform by the StackOverflow founder), we chose Python due to its ease of use and ease of read without having mastered the software. Another advantage is the number of external modules that can be used to extend the model as well. For example, analysis of generated data can be done without the use of external programs and platforms, such as R, SPSS or SAS. The implementation in dedicated agent-based Python modules (such as MESA, ABCE, SPADE and AgentPy, to name a few) is possible, but it demands not only that the reader is familiarized with the programming language, but also to a meta-layer that comes with the usage of such methods. Finally, another aspect that favors Python is the fact that it can be used to speed up simulations that are done in NetLogo using a package developed by Gunaratne and Garibay (2018).

Since the intent of this work is to build from scratch all models developed by Nelson and Winter, the values used as parameters for the models are the same as in the original work. Choosing to do so was necessary to guarantee the reproduction of qualitative results. However, an exception occurs in the NW84 model, in which one of the parameter values is not available. To measure the relevance of this parameter and how it might affect the model, we conduct a sensitivity analysis. Our results point that the model is robust to changes in that parameter, that is, results are the same as long as the parameter is bigger than zero.

The manuscript is divided into four sections that describe distinct implementations of the NW models. The first model is the simplest version, taking into account only supply and demand; the second model introduces investments in capital and the possibility of innovation; the third model includes the entrance and exit of firms, and the fourth model introduces patents. Every section is divided into subsections that describe the dynamics of the model and its implementation in Python.

2. A simple one step market with homogeneous firms

In this section we implement a very simple model that describes a part of what is to come. There is no time evolution and, therefore, no changing environment. Its utility comes from the fact that it serves as a skeleton of all other implementations. The subsection model describes the system's equations. The implementation subsection shows how Python's class features can be used to implement such system.

2.1. The model

The simplest model has n homogeneous firms, each firm has the same technology and produces q_i units of a homogeneous product, their output is described by the equation

$$q_i = A_i K_i \qquad (1)$$

where K_i is the stock of capital and A_i is the technology. All firms have an average cost of c. The firms face a market with an inverse demand curve

$$P = \frac{D}{Q^{\frac{1}{\eta}}} \qquad (2)$$

where Q is the sum of individual firm outputs; η is the price elasticity of demand, which is typically one; P is the market clearing price; and D is the demand parameter.

2.2. Implementation

We now proceed to the implementation of firms and industry. To do such a thing, we will use Python's object-oriented programming features. First, we construct a class responsible for representing firms. Once a firm instance is initialized, the values of K, A and c are assigned. The only method, besides the constructor, of our constructed class is used to calculate the individual output

```
class Firm:
    def __init__(self, A, K):
        self.A, self.K = A, K
        self.c = 0.16
    def output(self):
        self.qi = self.A*self.K
```

Next, we construct a new class that defines the industry, which is composed by a collection of firms. When an industry is initialized the parameters n, A_0, K_0 and D are assigned and an empty list of firms is created. Three class methods are then defined. The create method initializes n instances of firms and appends then to a list; next, the price method calculates the individual output of each firm in the generated list and then calculates the total output; last, the reset method provides a convenient way to reset an instance's firms attribute.

The Industry class is a base class because it does not depend on anything defined previously. Its constructor method has the same parameters as the Firm one, except for the number of firms that will be part of the collection and the demand parameter D. Besides the initial parameters, a new attribute called firms is created as a list and it will be used to represent the collection of firms in the system.

```
class Industry:
    def __init__(self, n=2, A0=0.16, K0=139.58, D = 67):
        self.n = n
        self.A0 = A0
        self.K0 = K0
```

```
        self.D = D
        self.firms = []
```

The reset method is used to recreate an empty list of firms. This may be not so useful in this simple implementation, but it will be as the program increases in size. Many attributes will change dynamically and to start a new run of an industry they need to be set back to their default values.

```
    def reset(self):
        self.firms = []
```

The create method populates the system with firms. The loop structure goes over a range defined by the parameter n and a firm is appended to the firms attribute for every element in the iterator.

```
    def create(self):
        for i in range(self.n):
            firm = Firm(self.A0, self.K0)
            self.firms.append(firm)
```

The last method is price, it loops over the list of firms and sum their productions to calculate the market price defined as the p attribute. We define total production as Qtot.

```
    def price(self):
        self.Qtot = 0
        for firm in self.firms:
            firm.output()
            self.Qtot += firm.qi
        self.p = self.D/self.Qtot
```

Once classes are defined, results can be generated. First, we initialize an industry object with default parameters. The following step is to populate the industry by using the create method. With the list of firms populated, the final step is to calculate the product's price. Generated results of this implementation are the product's price and the industry's output. Printing the attributes p and Qtot shows them.

```
industry = Industry()
industry.create()
industry.price()
print("Price={}, Quantity={}".format(industry.p, industry.Qtot))
```

Therefore, an industry containing $n = 2$ firms, whose initial technology $A_0 = 0.16$, capital $K_0 = 139.58$ and $D = 64$ will have a $Price = 1.50004$ and $Quantity = 100.50268$.

3. The NW82

The objective in this section is to briefly describe and implement the Schumpeterian Competition model presented in Nelson and Winter (1982). In this model, firms have more possibilities such as the investment in research and development. Such practice allows firms to innovate, which leads to the development of new production techniques, increasing overall productivity; or to imitate the prevailing best practice. Alongside with randomness, the amount of stock of capital invested R&D will define if a firm is allowed to innovate or imitate at a given period. We first describe the model in the subsection model, then we proceed to the implementation, where we describe how firms, industry and a simulation environment are implemented.

3.1. The model

The production and demand characteristics are the same as described in the simplest model. The distinction now is that firms are able to change their stock of capital through investment decisions. Profits will define most of the decisions made by firms, at a given period t they are defined as

$$\pi_{i,t} = P_t A_{i,t} - \left(c + r^{im} + r^{in}\right) \qquad (3)$$

where P is price, $A_{i,t}$ is the current production technique (or technology), c denotes production costs, r^{im} is the R&D imitative cost and r^{in} is the innovative cost. Innovative and imitative costs are fixed at the firm level and do not change over time.

At each period, R&D activity will generate new productivity levels for each firm. A firm will innovate successfully if $d^n_{i,t} = 1$ and if not $d^n_{i,t} = 0$. The probability of innovation is

$$Pr\left(d^{in}_{i,t} = 1\right) = a_n r_{in} K_{i,t} \qquad (4)$$

and imitation's

$$Pr\left(d^{im}_{i,t} = 1\right) = a_m r_{im} K_{i,t} \qquad (5)$$

a_n and a_m are sufficiently small parameters that won't let the probability of innovation and imitation be equal to one. Such process is described by a uniform random variable, but has been described as a poisson random variable by Andersen et al. (1996) and Valente and Andersen (2002).

Once a firm gets the chance to innovate, i.e. $d^{in}_{i,t} = 1$, a new technology level will be sampled from a Lognormal(L(t), \sigma) distribution. $L(t)$ is called latent productivity and describes the evolution of fields of knowledge related to the industry's endeavors, but unrelated to the industry's own research, to describe such function, we use the definition in Winter (1993):

$$L(t) = L_0 + \gamma t \qquad (6)$$

where L_0 is the initial amount of latent productivity and γ is a parameter that describes the latent productivity growth at every period t .

If a firm imitates, then it has the option to copy the best technological practice. There's also the possibility that a firm both innovates and imitates. If that is the case, future technology level will be given by

$$A_{i,(t+1)} = \max\left(A_{i,t}, \widehat{A_{i,t}}, \overline{A_{i,t}}\right) \quad (7)$$

where $\hat{A}$ is the best practice among the industry and $\bar{A}$ is a value sampled from the lognormal distribution.

The following step is to define the evolution of a firm's stock of capital K. This investment decision is determined by its percentage margin divided by the average cost, its market share (individual output divided by total output), its profitability and a physical depreciation rate. The function that describes this decision will be defined as

$$K_{i,(t+1)} = I\left(\frac{P_t A_{i,(t+1)}}{c}, \frac{q_{i,t}}{Q_t}, \pi_{i,t}, \delta\right) K_{i,t} + (1-\delta)K_{i,t} \quad (8)$$

where δ is the physical depreciation rate and I is the gross investment function. The gross investment function depends on two other functions: desired and financeable investment. The former depends on the conjecture of an optimal margin. Firms will only invest while they can still obtain profits, such conjecture is called $\mu(s)$

$$\mu(s) = \frac{\eta + (1-s)\psi}{\eta + (1-s)\psi - s} \quad (8)$$

where η is the price elasticity of demand and ψ is the price elasticity of supply. If a firm has a market share, s , equal to one

$$\mu(s) = \frac{P_t - c}{P_t} \quad (9)$$

The desired investment will be then

$$I_D = \delta + 1 - \mu(s) \cdot \frac{c}{P_t A_{i,t(+1)}} \quad (10)$$

The conjecture $\mu(s)$ is relevant because it sets an upper boundary to the system's development. Once a firm concludes its percentage margin over cost is equal to the conjecture, there's no incentive to keep investing more than the depreciation rate.

Next, the financeable investment describes the amount of investment a firm is able to obtain given its profit at a period t

$$I_F = \begin{cases} \delta + \pi_{i,t} & \pi_{i,t} \leq 0 \\ \delta + b\pi_{i,t} & \pi_{i,t} > 0 \end{cases} \quad (11)$$

where b is a parameter which describes the amount that a firm can borrow if its profit rate is positive.

Finally, the gross investment function will be:

$$I = max\{0, min\{I_D, I_F\}\} \quad (12)$$

3.2. Implementation

3.2.1. Firm

To implement firms with the aforementioned characteristics, we use Python's class inheritance features. This is convenient because the new child class will inherit all attributes and methods from the parent class. In our case, a NW82Firm inherits all methods from a Firm. New methods created are as follows: profit calculates a firm's profit; innovation determines the probability of innovation and the amount of technology obtained from it; imitation determines the probability of imitation; finally, update determines the investment decision of the firm. A brief description of each method along with its code is provided below.

First, a widely known module, NumPy, is imported for generating random numbers, array construction and manipulation and also for useful mathematical functions, such as log and exp, which will be used for innovative draws. The new class, NW82Firm, is created using the Firm as a base class. Then, the constructor method is changed to include other necessary parameters for the simulation. Besides the initial production technique and stock of capital, it now includes the amount of research and development spent in innovation and imitation respectively. Two new attributes are initialized, Ain and Aim, they describe the productivity of innovation and imitation from random draws at a period t.

```
class NW82Firm(Firm):
    def __init__(self, A0, K0, rin, rim):
        super().__init__(A0, K0)
        self.rim = rim
        self.rin = rin
        self.Ain = 0
        self.Aim = 0
```

The method profit describes the amount of profit a firm has at a price p. Notice that it takes a parameter generated by Industry. Its result is saved to the attribute pi.

```
def profit(self, p):
        self.pi = p*self.A - (self.c + self.rim +
        self.rin)
```

Another implemented method, innovation, describes the innovative draws from a given firm. The probability of innovation is calculated by using the amount invested in innovation, rin; the parameter that balances the probability[1] an; and the stock of capital. Then, if a firm innovates, it samples a new productivity technique. For that sampling, two variables are needed, mu is the latent productivity of the distribution and sigma is the variance of the distribution. The result is saved to the attribute Ain.

```
def innovation(self, an, mu, sigma):
        prob_inn = an*self.rin*self.K
        if np.random.uniform(0,1) < prob_inn:
            Ain = np.exp(np.random.normal(np.log(mu), sigma))
        else:
            Ain = self.A
        self.Ain = Ain
```

The method imitation is very similar to innovation and describes the imitation draw from a given firm. Instead of sampling from all available techniques, imitation allows the firm to learn the best available, bestA, production technique. Its result changes the attribute Aim.

```
    def imitation(self, am, bestA):
        prob_imi = am*self.rim*self.K
        if np.random.uniform(0,1) < prob_imi:
            Aim = bestA
        else:
            Aim = self.A
        self.Aim = Aim
```

Finally, update is a method that describes the firm's stock of capital evolution and its production technique choice. It takes as parameters the total production in the system, Qtot; the product's price, p; the amount a firm can borrow, bank; and the depreciation rate, delta.

```
def update(self, Qtot, p, bank, delta):
        self.A = max(self.A, self.Aim, self.Ain)
        s = self.qi/float(Qtot)
        rho = p*self.A/float(self.c)

        if s < 1:
```

[1] In the sense that it won't let the probability of innovation be equal to one.

```
            I_d = 1+delta-float(2-s)/float(rho*(2-2*s))
        else:
            I_d = (p-self.c)/p

        if self.pi <= 0:
            b = 0
        else:
            b=bank
        I_p =  delta + (b+1)*self.pi
        self.K = (1-delta+max(0, min(I_d, I_p)))*self.K
```

The process goes as follows: first, the new production technique is defined through the comparison between the technique being used and the results from random draws. Then, two variables are calculated, s, which is calculated using the parameter Qtot; and the variable rho, which is the percentage margin over cost. The conjecture is used to estimate desired investment only if a firm's market share is less than one[2], otherwise a monopolist mark-up formula is used. The financeable investment is determined. If profit is less or equal than zero, the amount a firm can borrow will be zero; otherwise, a firm will be able to borrow its own profit plus its profit multiplied by bank. The results are used to calculate a firm's stock of capital at the end of a period t.

3.2.2. Industry

The next step is to implement an industry that will contain a collection of new firms. Once again class inheritance is going to be used while also including new methods and attributes. Methods such as reset and create are redefined due to inclusion of new features. The former will not only be able to reset the list of firms, but also lists that describe production techniques and stocks of capital for all firms in the system. The latter will be able to create two different types of firms: innovators and imitators3. The newly added method, update_industry, describes the evolution of an industry while also calculating the Herfindahl–Hirschman index[4] at each period t , but it could also be used to calculate other relevant statistics. Descriptions of each methods and their code are below.

We start by describing the new class, NW82Industry and its constructor. The new class has Industry as a base class, which was earlier defined. The constructor takes as

[2] Although only explicitly defined in Winter (1984), we assume the parameters $\eta = 0$ and $\phi = 2$ for the conjecture.

[3] The difference is given by the cost of innovation and imitation: an innovator firm has an imitative cost set to zero.

[4] An index used to measure the amount of output concentration in the industry.

parameters the same set of parameters described in NW82Firm, except for n, which describes the amount of firms the industry will have. Many relevant parameters are set as attributes: an and am, responsible for balancing random draws; mu, the initial level of latent productivity, and sigma, the variance of random innovative draws; delta, the depreciation rate of capital; Bank, a rate which describes the amount firms can borrow; time, describing at which iteration the system is; b_A, describing the best available technique; IHH, the reciprocal of the HH index, describing the amount of "equivalent firms"; and finally, latent, which describes the growth rate of the latent productivity.

```
class NW82Industry(Industry):
    def __init__(self, n=2, K0=139.58, A0=0.16, rin=0.0287,
rim=0.00143):
        super().__init__(n, A0, K0)
        self.rin = rin
        self.rim = rim
        self.an, self.am = .125, 1.25
        self.mu, self.sigma = 0.16, 0.05
        self.delta = 0.03
        self.Bank = 1
        self.time = 0
        self.b_A = 0
        self.IHH = self.n
        self.latent = 1.01
```

Next, we choose to override the reset method, which sets parameters that change over iterations to their initial values.

```
def reset(self):
        self.firms = list()
        self.time = 0
        self.b_A = 0
        self.mu = 0.16
```

The method create is also overridden. It is now composed by two loop structures instead of a single one. These two loops are very similar to the first one defined in the simple implementation of the model, but they now divide the collection of firms into two distinct groups. The first one is defined with all earlier given parameters. However, the second one isn't able to innovate, so their attribute rin is set to zero.

```
def create(self):
        for i in range(int(self.n/2)):
            firm =  NW82Firm(self.A0, self.K0, self.rin, self.rim)
            self.firms.append(firm)
        for i in range(int((self.n+1)/2)):
            firm = NW82Firm(self.A0, self.K0, 0, self.rim)
```

```
            self.firms.append(firm)
```

In this class, the final method is update_industry. In a certain sense, this method is a composition of all other methods and it is going to change relevant parameters and attributes of the industry dynamically. Its first three lines describe the evolution of latent productivity and assigns their values to variables that are going to be used for innovation draws. We use list comprehension to create two lists and each of their values are representing individual production techniques and stocks of capital respectively. These lists are used for calculating two statistics: the HH index and the best available production technique. A loop structure wraps up all NW82Firm methods and calculates their values for every firm in the system. Finally, one is added to the time attribute so that time evolution can be described in the system.

```
def update_industry(self):
        self.mu *= self.latent
        mux = self.mu
        sigmax = self.sigma
        self.Alist = [self.firms[i].A for i in
range(len(self.firms))]
        self.Klist = [self.firms[i].K for i in
range(len(self.firms))]
        HH = sum([k**2 for k in
self.Klist])/float(sum(self.Klist)**2)
        self.IHH = 1/float(HH)
        self.b_A = max(self.Alist)
        for firm in self.firms:
            firm.profit(self.p)
            firm.innovation(self.an, mux, sigmax)
            firm.imitation(self.am, self.b_A)
            firm.update(self.Qtot, self.p, self.Bank, self.delta)
        self.time += 1
```

3.2.3. Simulation Environment

As the program grows in complexity, a dedicated environment is necessary to generate and collect data at every period and for multiple trials. To do this, a simulation class, which we will call SimNW82, is built such that it will run N distinct simulations, each consisting of T periods. This class has two methods, one is the constructor, which contains the relevant values for the industry and the initialization of arrays that are going to collect data for each simulation and period. The other one is the simulate, which will contain steps for each simulation and period.

The constructor method is, in a sense, self-explaining. It takes all the necessary parameters for the functioning of the system, such as the number of firms, their initial stock of capital and production technique, and their investment decisions. We define attributes that are responsible for collecting data and their dimensions are of T rows and N columns. In our simulation, we are collecting data related to (prices), the number of "equivalent" firms and the best available technology at a given time period. Other values could be collected as well, such as the number of firms or the mean market share in the system.

```
class SimNW82:
    def __init__(self,T =100, N= 20, n=2, K0=139.58, A0=0.16,
            rin=0.0287, rim=0.00143):
        self.T, self.N = T, N
        self.n = n
        self.K0, self.A0 = K0, A0
        self.rin = rin
        self.rim = rim
        self.ntprices = np.zeros((self.T, self.N))
        self.ntIHH = np.zeros((self.T, self.N))
        self.ntb_A= np.zeros((self.T, self.N))
```

To generate runs of simulations, we define the simulate method. It creates an instance of an industry using the NW82Industry class and pre-defined parameters in the constructor method that are given by the user. After defining an instance, the process of time evolution and data collection are described by two nested loop structures. In the first loop, for every simulation run, industry parameters that change dynamically will be reset, and new firms will populate it right after. The second loop changes the industry and collects data for every period T.

```
    def simulate(self):
        indus = NW82Industry(self.n, self.K0, self.A0,
                self.rin, self.rim)
        for i in range(self.N):
            indus.reset()
            indus.create()
            for t in range(self.T):
                indus.price()
                indus.update_industry()
                self.ntprices[t,i] = indus.p
                self.ntIHH[t,i] = indus.IHH
                self.ntb_A[t,i] = indus.b_A
```

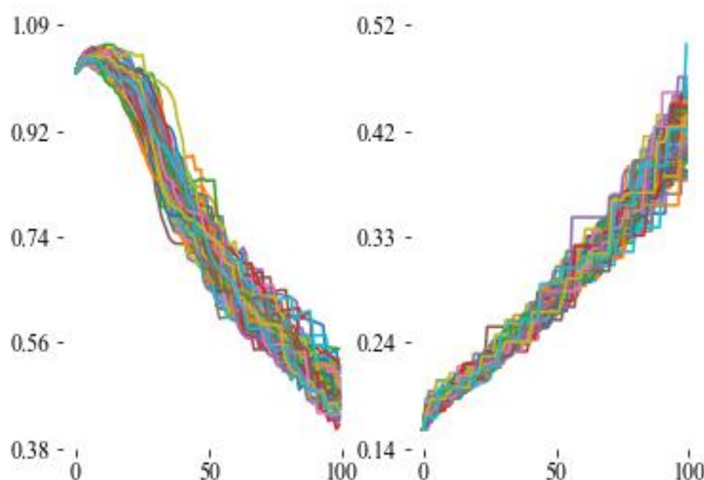
3.3. Results

We now proceed by running N = 100 simulations each having a time span of $T = 100$ periods. There are $n = 32$ firms, with initial technology $A_0 = 0.16$, initial stock of capital $K_0 = 12.89$ and costs of innovation and imitation of 0.194 and 0.00097 respectively. From the results, we have generated figures using the matplotlib module.

```
sim = SimNW82(N = 100, n=32, K0= 12.89, rin = 0.194, rim =
0.00097)
sim.simulate()
```

The left panel of figure 1 shows price (y-axis) evolution as time (x-axis) progresses. Initially, prices are increasing due to the fact that firms lose part of their stock of capital to depreciation. As time progresses, and firms start to innovate or imitate, prices start to get lower. The right panel shows that the best available production technique (y-axis) also increases as time evolves.

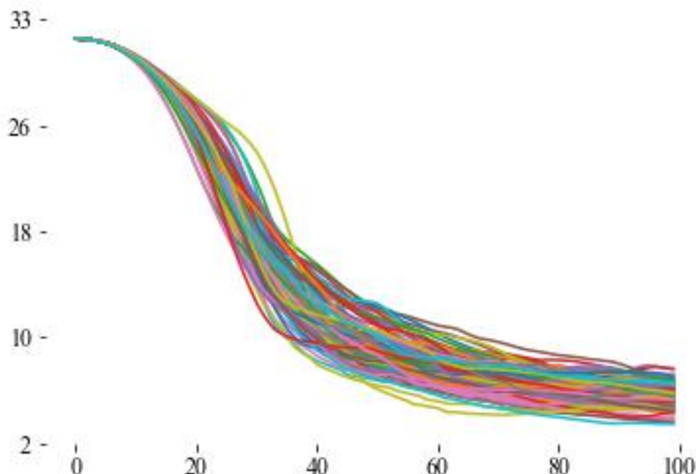
Figure 1: Price and technology evolution in the NW82 model



Source: own elaboration.

Figure 2 shows how the inverse of the HH index (y-axis) evolves over time (x-axis). When output concentration is low, the number of equivalent firms is very close to the number of firms in the system. As firms discover new production techniques, some start to grow more rapidly than others. Some firms do not innovate or imitate and their size might get negligible when compared to the bigger competitors.

Figure 2: Number of equal firms evolution in the NW82



Source: own elaboration.

A few problems arise in this model. If a firm is too small when compared to those that progressed rapidly, why would it still be in the industry? Another interesting question is this: if a firm isn't evolving as rapidly as its competitors, why wouldn't it invest more aggressively in R&D to obtain better results? Also, when prices are sufficiently high, why wouldn't other firms enter in the system? The next section tackles the implementation of such possibilities.

4. The NW84

This section's objective is to implement the model described in Winter (1984). This is an extension to the NW82 model. Instead of fixed imitative and innovative costs at the firm level, firms will adjust their R&D policies depending on a level of performance. Another addition to the NW82 model is the possibility of entry and exit of new firms. We focus in the entrepreneurial regime, where innovative entry is favored, but our implementation is general in the sense that the routinized regime could be simulated with a change in parameters.

First, we are going to start by implementing a model with adaptive changes in R&D policies while also adding the possibility that a firm leaves the industry when it isn't performing as well as it should. Finally, we will implement the possibility that new firms enter the market, provided that it is still possible to earn profits.

4.1 The Model

A new relevant variable is introduced to the model: $X_{i,t}$. This variable describes the performance of the i-th firm at period t. In general, the bigger the profits a firm has, the bigger is its performance. If at a given period t a firm has a lower profit than usual, performance isn't changed immediately, it makes a weighted sum of its past performance and its current profit. The equation that describes it is given as

$$X_{i,t} = \theta X_{i,(t-1)} + (1-\theta)\pi_{i,t} \qquad (12)$$

where θ is a value between 0 and 1. The bigger it is, the smaller is the effect of recent profits in a firm's performance.

The next step is to describe how firms change their R&D policies. Winter (1984) describes this search in the investment space as an adaptive rule. Firms with performances lower than the mean of their competitors' profits will tend to change their policies. This change will depend on the mean of the competitor's investments. Such adaptive rule is given as

$$r^{m}{}_{i,(t+1)} = \begin{cases} r^{m}_{i,t} & X_{i,t} \geq \overline{\pi}_t \\ 1 - \beta r^{m}_{i,t} + \beta \overline{r}^{m}_{t} + u^{m}_{i,t} & \pi_{i,t} > 0 \end{cases} \qquad (13)$$

$$r^{n}{}_{i,(t+1)} = \begin{cases} r^{n}_{i,t} & X_{i,t} \geq \overline{\pi}_t \\ 1 - \beta r^{n}_{i,t} + \beta \overline{r}^{n}_{t} + u^{n}_{i,t} & \pi_{i,t} > 0 \end{cases}$$

where β is a parameter that describes how a policy will change when compared to competitors' policies. Bar describes mean, and u is a random normal variable truncated at 0 with specific variance for each type of investment. Although low performance is a primary condition for policy change, the author suggests that such changes will happen only with a 50% chance, which we will describe as a uniform random variable. Finally, in order to guarantee that firms will test their new policies for a while, whenever a firm changes its policies, performance will be updated by a value Δ.

Now, we have to define how firms leave the industry. Two conditions were proposed by Winter: when investment in capital is below a minimum or when performance is below a minimum. Such rule is given as

$$\left(I\left(\frac{P_t A_{i,(t+1)}}{c}, \frac{q_{i,t}}{Q_t}, \pi_{i,t}, \delta\right) + (1-\delta) \right) K_{i,t} < K^{min} \text{ or} \qquad (14)$$

$$X_{i,t} < X^{min}$$

once a firm leaves the industry, its capital is destroyed.

The final step is to describe how an entrant decides to enter the industry. A fixed amount of potential entrants is generated at every period. This amount depends on the assumption that there is some external form of investment in R&D and it is responsible for determining the mean amount of entrants. These means, M_t for imitators and N_t for innovators, are given by the equations

$$\begin{aligned} M_t &= A_m E_m \\ N_t &= a_n E_n \end{aligned} \qquad (15)$$

where E_m and E_n are representations of external expenditure in imitative and innovative R&D. These calculated means are used in a poisson process.

Finally, a potential entrant decides to join the industry when it finds out that the technology available to it will generate a percent margin minus cost that is superior to an entry barrier rate summed up with an error term

$$P_t A_t - c > r_e + u_{e,t} \qquad (16)$$

where the error term, $u_{e,t}$ is a normal random variable truncated at zero and has its own fixed variance. Entering firms have to draw their capital from a normal random variable also truncated at zero and a specific variance.

4.2. Implementation

4.2.1. Firm

The new class firm is able to update its R&D policies and for that it also needs to calculate its performance. To do so, we define the NW84FirmE class and use the NW82Firm as its base class. First, we change its constructor method to include three new attributes: a beta value, a performance list and a trajectory list.

```
class NW84FirmE(NW82Firm):
    def __init__(self, A0, K0, rin, rim, beta, X0 = 0.001):
        super().__init__(A0, K0, rin, rim)
        self.beta = beta
        self.X = [X0]
        self.trajectory = [K0]
```

Although inefficient, because this is the unique time that has to be done, we choose to override the update method for a change of parameters and for including a firm's stock of capital's trajectory. Parameters, such as η and ψ that are defined as in Winter (1984).

```
def update(self, Qtot, p, bank, delta ):
        self.A = max(self.A, self.Aim, self.Ain)
```

```
        s = self.qi/float(Qtot)
        rho = p*self.A/float(self.c)
        if s < 1:
            I_d = 1+delta-float(3-2*s)/float(rho*(3-3*s))
        else:
            I_d = (p-self.c)/p

        if self.pi <= 0:
            b = 0
        else:
            b=bank
        I_p =  delta + (b+1)*self.pi
        self.K = (1-delta+max(0, min(I_d, I_p)))*self.K
        self.trajectory.append(self.K)
```

A very simple method for performance calculation, calculate_performance is included in this new class and it is self-explaining.

```
def calculate_performance(self, theta):

        self.X.append(theta*self.X[-1] + (1-theta)*self.pi)
```

The last method included is called update_policy. It uses the means of investment in innovation, imitation and profits from all competitors in the industry to calculate if a firm should change its policies. Notice that if a firm changes its policies, a constant will be added to its performance variable, this will let firms establish their new policies. The original work does not determine a value for this parameter and in response to this, a fixed value is chosen for this update: 0.001. In appendix A, a sensitivity analysis is conducted considering different values for this parameter. Our results show that this parameter does not exercise a great influence over the qualitative results of the model as long as it is bigger than zero.

```
def update_policy(self, mrin, mrim, mpi):
        if self.X[-1] < mpi and np.random.uniform(0, 1) > 0.5:
            value_one = np.random.normal(0, 0.0004)
            value_two = np.random.normal(0, 0.002)
            uin = value_one if value_one > 0 else 0
            uim = value_two if value_two > 0 else 0
            self.rin = (
                (1-self.beta) * self.rin + self.beta * mrin + uin
                )
            self.rim = (
                (1-self.beta) * self.rim + self.beta * mrim + uim
                )
            self.X[-1] += 0.001
```

4.2.2. Industry

For industries, we define two distinct new classes. In the first class, NW84IndustryE, we include in it the possibility of exit. The second class, NW84IndustryEE, is built upon NW84IndustryE and it has the possibility of entry. The constructor method of the former class is changed to include new parameters and attributes, such as the minimum amount of stock of capital and performance, new initial latent productivity and variance values, a new demand parameter, and most notably, a new attribute named history, which will contain a list of trajectories for firms that leave the market.

```
class NW84IndustryE(NW82Industry):
    def __init__(
            self, n, K0, A0, rin, rim,
            Kmin = 10, Xmin = -0.051, theta = 0.25, beta = 0.167
            ):
        super().__init__(n, K0, A0, rin, rim)
        self.Kmin, self.Xmin = Kmin, Xmin
        self.theta, self.beta = theta, beta
        self.mu, self.sigma = 0.135, 0.1177
        self.an, self.am = 0.025, 2.5
        self.D = 64
        self.history = []
```

This time we chose not to override reset completely; everything from the last class is used, but we also include the fact that history has to be reset.

```
def reset(self):
        super().reset()
        self.history = []
```

The create method is completely overridden. There is no such thing as imitators and innovators, all firms are able to innovate or imitate.

```
    def create(self):
        for i in range(self.n):
            firm = NW84FirmE(self.A0, self.K0, self.rin, self.rim,
self.beta)
            self.firms.append(firm)
```

Another method that suffered a change was price it is now defined as a piecewise function as in Winter (1984).

```
def price(self):
        self.Qtot = 0
        for firm in self.firms:
            firm.output()
            self.Qtot += firm.qi
        if self.Qtot < 53.33:
            self.p = 1.20
        else:
            self.p = self.D/self.Qtot
```

The method that updates the industry is now changed to include a list that has the mean of production techniques at every period t.

```
def update_industry(self):
        super().update_industry()
        self.meanA = sum(self.Alist)/len(self.firms)
```

Although this could have been done in the update_industry, a new method is created for changing all policies across firms. It calculates the means in investments and profits, calculates the performance and then changes policies accordingly.

```
def update_policies(self):
        self.mrin = sum(
            [self.firms[i].rin for i in
range(len(self.firms))])/len(self.firms

)
        self.mrim = sum(
            [self.firms[i].rim for i in
range(len(self.firms))])/len(self.firms

)
        self.mpi = sum(
            [self.firms[i].pi for i in
range(len(self.firms))])/len(self.firms

)
for firm in self.firms:
            firm.calculate_performance(self.theta)
            firm.update_policy(self.mrin, self.mrim, self.mpi)
```

The last included method in this version of the class is remove_firms. It is a loop structure that verifies if a firm has a sufficient amount of stock of capital and performance to stay in the industry. If a firm leaves, it is completely deleted, but its stock of capital trajectory is included in the history attribute.

```
def remove_firms(self):
```

```
        for index, firm in enumerate(self.firms):
            if firm.K < self.Kmin or firm.X[-1] < self.Xmin:
                self.history.append((self.time, firm.trajectory))
                    del self.firms[index]
```

The second class NW84IndustryEE is very similar to its base class NW84IndustryE. Its constructor method includes the amount of external investments in R&D. Counters for the number of potential entrants are initialized as well. Although not included, the reset method has been changed to take into account these new counters.

```
class NW84IndustryEE(NW84IndustryE):
    def __init__(self, n, K0, A0, rin, rim, En= 2, Em = 0.2):
        super().__init__(n, K0, A0, rin, rim)
        self.En, self.Em = En, Em
        self.ncount = 0
        self.mcount = 0
```

Also, a new method is added to the class. It defines the process of entry of new firms in the industry. The amount of possible imitators and innovators is calculated and then two loop structures are defined. Imitators get access to the best available technology and innovators have to draw their initial production technique. A new entering firm has its β set to one as in Winter (1984) and we assume that its initial investments are the same as other competitors.

```
def entry(self):
        imitators = np.random.poisson(self.Em * self.am)
        innovators = np.random.poisson(self.En * self.an)
        for i in range(imitators):
            value = np.random.normal(0, 0.014)
            entry_error = value if value >= 0 else 0
            if self.p * self.b_A - 0.16 > 0.007 + entry_error:
                value = np.random.normal(25, 7.5)
                init_K = value if value >= self.Kmin else
self.Kmin
                firm = NW84FirmE(self.b_A, init_K, self.rin,
self.rim, beta = 1)
                self.firms.append(firm)
                self.mcount += 1
        for i in range(innovators):
            value = np.random.normal(0, 0.014)
            entry_error = value if value >= 0 else 0
            init_A = np.exp(np.random.normal(np.log(self.mu),
self.sigma))
            if self.p * init_A- 0.16 > 0.007 + entry_error:
                value = np.random.normal(25, 7.5)
```

```
                init_K = value if value >= self.Kmin else
self.Kmin
                firm = NW84FirmE(init_A, init_K, self.rin,
self.rim, beta = 1)
                self.firms.append(firm)
                self.ncount += 1
```

4.2.3. Industry

The simulation enviroment changes very little between industries, but two new classes are defined for simulations. The first one is SimNW84E and it now includes an attribute that collects data for the mean production technique. It also includes a list for industry histories and an array for the number of firms in every period t.

```
class SimNW84EE(SimNW84E):
    def __init__(self, T =100, N= 20, n=2, K0=139.58, A0=0.16,
            rin=0.0287, rim=0.00143):
        super().__init__(T, N, n, K0, A0)
        self.nentrants = np.zeros(self.N)
        self.mentrants = np.zeros(self.N)
        self.mprofits = np.zeros((self.T, self.N))
        self.mrin = np.zeros((self.T, self.N))
        self.mrim = np.zeros((self.T, self.N))
        self.nentrants = np.zeros(self.N)
        self.mentrants = np.zeros(self.N)
    def simulate(self):
        indus = NW84IndustryEE(self.n, self.K0, self.A0,
                self.rin, self.rim)
        for i in range(self.N):
            indus.reset()
            indus.create()
            for t in range(self.T):
                    indus.price()
                    indus.update_industry()
                    indus.update_policies()
                    indus.remove_firms()
                    indus.entry()
                    self.ntprices[t,i] = indus.p
                    self.ntb_A[t,i] = indus.b_A
                    self.ntIHH[t,i] = indus.IHH
                    self.meanA[t,i] = indus.meanA
                    self.firm_amount[t, i] = len(indus.firms)
                    self.mprofits[t, i] = indus.mpi
                    self.mrin[t, i] = indus.mrin
                    self.mrim[t, i] = indus.mrim
            self.histories.append((i, indus.history))
```

```
            self.nentrants[i] = indus.ncount
            self.mentrants[i] = indus.mcount
```

4.3. Results

4.3.1. Model with exit only

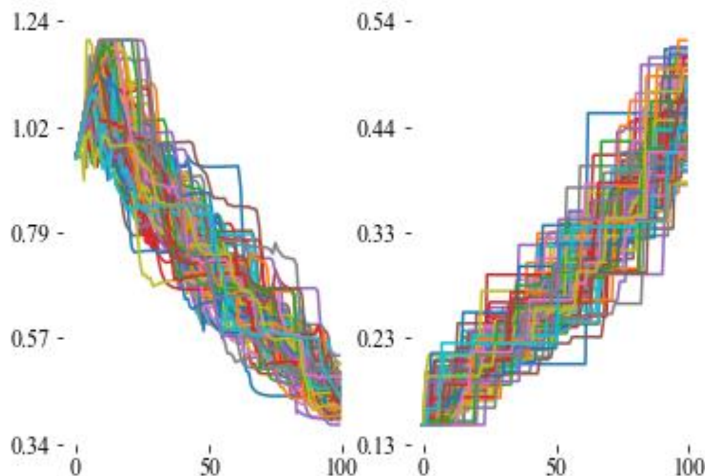
The first simulation analyses results from an industry which only exit is allowed. To do so, we instantiate a SimNW84E object. We chose a firm amount of $n = 18$ so that the industry does not die out instantly[5]. Figure 3 shows results from simulations. The right panel There's more variability in the model due to the fact that firms can exit and also because they are able to change their investment policies. When a firm exits the market and its capital is destroyed, prices jump because the total output in industry is diminished. Best available production technique is increasing in time but also has a lot of volatility due to the fact of changing policies.

Figure 4 right panel shows that the number of equivalent firms (y-axis) might fall rapidly due to early exit of firms. Its trajectory is smoother than the number of actual firms (y-axis) in industry presented in the left panel and in all of the simulations there were no cases which all firms died out. Results suggest that the number of equivalent firms is bigger than the number of actual firms, as expected, but variability is too large.

Figure 5 left panel shows a histogram of exit times. About 70% of exits occur at the 80th period. The right panel shows a histogram of the stock of capital at exit, suggesting that a large spike of exits occurs when firms have a stock of capital around 10 units. In fact, about 48% of firms leave when their stock of capital is equal to or less than 10. Notice, though, that the stock of capital is seldom above 30 units.

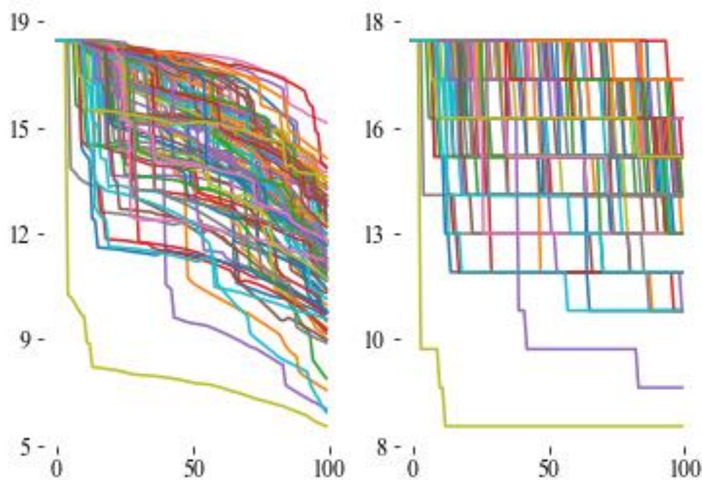
[5] At higher values, all firms exit the industry at the third or fourth period due to low performances.

Figure 3: Price and technology evolution in the NW84e model



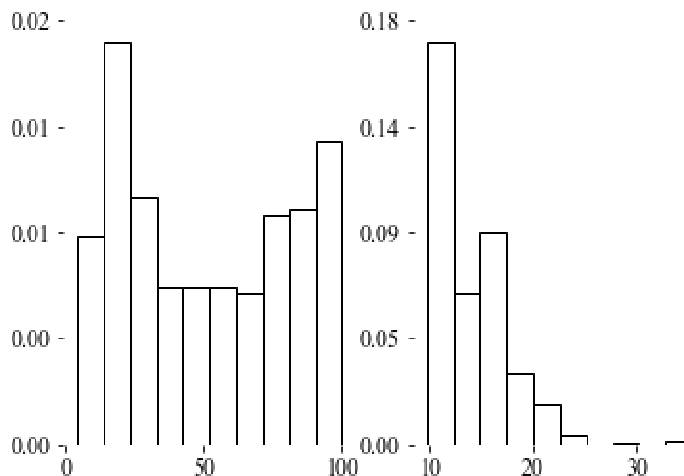
Source: own elaboration.

Figure 4: Equivalent and actual firms in the NW84e model



Source: own elaboration.

Figure 5: Exit patterns in the NW84E model



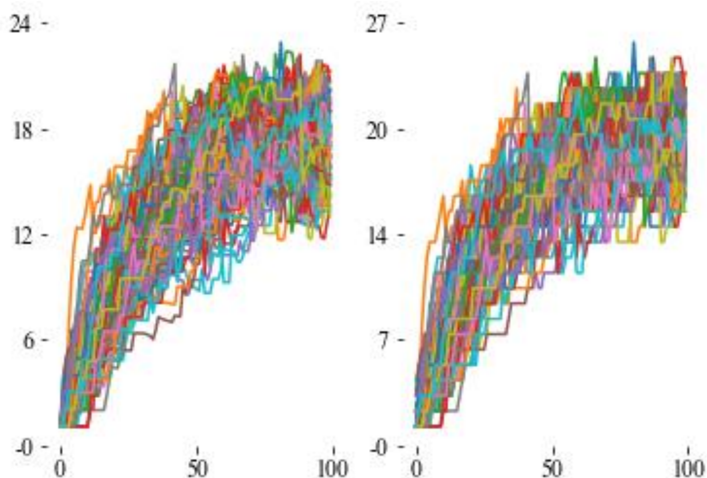
Source: own elaboration.

4.3.2. Model with exit and entry

To generate results for the model with entry, we run the SimNW84EE with a single firm in the industry. This will allow us to understand how the industry accommodates new entrants. Results related to convergence in prices and technology are very similar to the aforementioned model. The distinction here is in the amount of firms at the end of T (total amount of periods) and the distribution of exits and stock of capital at period t. Results in figure 6 show that the number of equivalent firms is around 10 to 20 firms, while the actual number of firms is around 15 to 25, which is greater than in the model without possibility of entry.

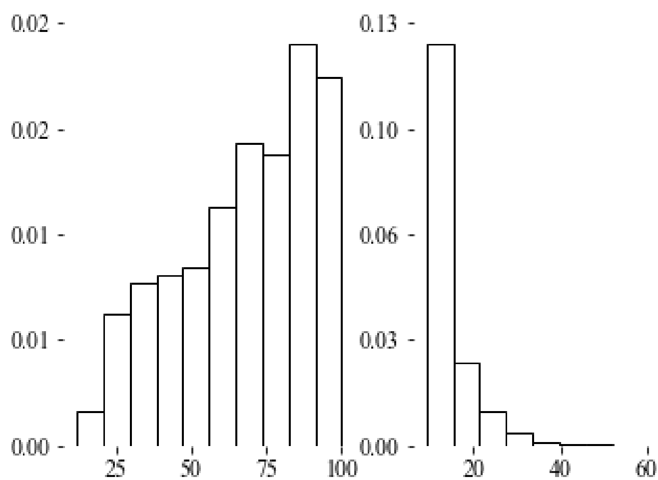
When it comes to the distribution of exits, shown in figure 7, two distinct patterns emerge. The number of exits increase as time increases. As new entrants join the competition, they are smaller than the already existing firms and innovate or imitate less frequently than their competitors. This leads to a smaller amount of technological progress and, consequently, a smaller increase of the stock of capital. The second histogram shows a more distinct pattern when it comes to stock of capital at exit, about 51% of firms leave the industry when their stock of capital is equal to or less than 10.

Figure 6: Equivalent and actual firms in the NW84EE model



Source: own elaboration.

Figure 7: Exit patterns in the NW84EE model



Source: own elaboration.

5. The NW93

This section's objective is to implement the model in Winter (1993). It is very similar to the model in Winter (1984). Its most notable distinction is the fact that innovative firms have access to patents which affect the diffusion of new production techniques in the

industry. Because the NW93 model is identical to the NW84 model when it comes to production, adaptive policies, and exit and entry of new firms, we skip the model subsection and write about its implementation directly.

5.1. Implementation

5.1.1. Industry

The behavior of firms does not change at all. What changes is the behavior of the industry. We create the NW93Industry class, which is built upon the NW84IndustryEE. The constructor method is similar to the last one, but it has two new attributes. A dictionary of patents, which has as a key the period t of the production technique discovery; and a list of techniques discovered at that period as its value. The other attribute describes the duration of a patent in time periods. The reset method is changed to include the fact that patents have to reset at a new simulation.

```
class NW93Industry(NW84IndustryEE):
    def __init__(self, n, K0, A0, rin, rim):
        super().__init__(n, K0, A0, rin, rim)
        self.patents = {}
        self.patent_duration = 12
        self.best_imitable = 0
def reset(self):
        super().reset()
        self.patents = {}
        self.best_imitable = 0
```

The update_industry method is changed to include patents. At every call, a loop structure compares the time attribute and the patents dictionary keys to verify if a patent has ended. Patents are created whenever a firm innovates.

```
    def update_industry(self):
        self.mu *= self.latent
        mux = self.mu
        sigmax = self.sigma

        if len(self.patents.keys()) > 0:
            oldest = min(self.patents.keys())
            if (self.time - oldest == self.patent_duration + 1):
                    self.best_imitable = max(self.patents[oldest])
                    del self.patents[oldest]

        self.Alist = [self.firms[i].A for i in
range(len(self.firms))]
```

```
        self.Klist = [self.firms[i].K for i in
range(len(self.firms))]
        self.b_A = max(self.Alist)
        self.meanA = np.mean(self.Alist)
        HH = sum([k**2 for k in
self.Klist])/float(sum(self.Klist)**2)
        self.IHH = 1/float(HH)

        for firm in self.firms:
            firm.profit(self.p)
            firm.innovation(self.an, mux, sigmax)
            firm.imitation(self.am, self.best_imitable)
            firm.update(self.Qtot, self.p, self.Bank, self.delta)

        self.time += 1

        for firm in self.firms:
            if firm.A == firm.Ain:
                if self.time in self.patents:
                    self.patents[self.time].append(firm.Ain)
                else:
                    self.patents[self.time] = [firm.Ain]
```

A final change has to be done to the entry method. If an innovative entrant draws a new production technique that allows it to enter the industry, the innovation will be patented.

```
def entry(self):
        imitators = np.random.poisson(self.Em * self.am)
        innovators = np.random.poisson(self.En * self.an)

        for i in range(imitators):
            value = np.random.normal(0, 0.014)
            entry_error = value if value >= 0 else 0
            if self.p * self.best_imitable - 0.16 > 0.007 +
entry_error:
                value = np.random.normal(25, 7.5)
                init_K = value if value >= self.Kmin else
self.Kmin
                firm = NW84FirmE(self.best_imitable, init_K,
self.rin, self.rim, beta = 1)
                self.firms.append(firm)
                self.mcount += 1

        for i in range(innovators):
            value = np.random.normal(0, 0.014)
            entry_error = value if value >= 0 else 0
```

```
            init_A = np.exp(np.random.normal(np.log(self.mu),
self.sigma))
            if self.p * init_A- 0.16 > 0.007 + entry_error:
                value = np.random.normal(25, 7.5)
                init_K = value if value >= self.Kmin else
self.Kmin
                firm = NW84FirmE(init_A, init_K, self.rin,
self.rim, beta = 1)
                self.firms.append(firm)
                self.ncount += 1

                if self.time in self.patents:
                    self.patents[self.time].append(init_A)
                else:
                        self.patents[self.time] = [init_A]
```

5.1.2. Simulation Environment

Changes in simulation environment are very simple. We develop a new class that takes into account the most recent industry class while also adding a new attribute. The imitable attribute takes into account the best imitable production technique at a period t.

```
def simulate(self):
        indus = NW93Industry(self.n, self.K0, self.A0,
                self.rin, self.rim)
        for i in range(self.N):
            indus.reset()
            indus.create()
            for t in range(self.T):
                    indus.price()
                    indus.update_industry()
                    indus.update_policies()
                    indus.remove_firms()
                    indus.entry()
                    self.ntprices[t,i] = indus.p
                    self.ntb_A[t,i] = indus.b_A
                    self.ntIHH[t,i] = indus.IHH
                    self.meanA[t,i] = indus.meanA
                    self.firm_amount[t, i] = len(indus.firms)
                    self.mprofits[t, i] = indus.mpi
                    self.mrin[t, i] = indus.mrin
                    self.mrim[t, i] = indus.mrim
                    self.imitable[t, i] = indus.best_imitable
            self.histories.append((i, indus.history))
            self.nentrants[i] = indus.ncount
            self.mentrants[i] = indus.mcount
```

5.2. Results

We run a simulation with a single firm and evaluate how the industry accommodates new entrants. In this simulation, imitators do not enter the market when there are no available technologies to imitate. Price convergence is the same as in the other models: as time passes, technology increases and prices fall. Other results are shown in table 1. The table compares results from models with and without patents. In the table, all values are means at the end of runs. Profits are min-max normalized, we use a normalized measure due to the fact that many firms have negative profits at the 100th period.

Table 1: Results from NW93 and NW84 models

	Entrants		**Investments**		**Profits**	
	No Patents	**Patents**	**No Patents**	**Patents**	**No Patents**	**Patents**
Imitation	38	27	0.347	0.4250	0.521	0.447
Innovation	3	4	2.914	2.957	0.521	0.447

Source: own elaboration.

In an industry without patents, the typical number of entrants is 38 for imitators and 3 for innovators. The ratio of capital invested is around 0.3472% for imitation and 2.914% for innovation at the 100th period. The normalized mean profits is around 0.5217. In line with Winter (1993), our results show that patents lower profits in an industry and the ratio of capital invested in innovation and imitation increases. Also, the number of entrants using imitation draws decreases in the presence of patents, but the number of innovators increases, a result that is not the same as in Winter (1993).

6. Concluding Remarks

In this work, we have implemented the Nelson and Winter models of Schumpeterian Competition. We used Python's object-oriented programming and the inheritance feature to build each of their classical models. An evolutionary description of the classical NW models is presented and implemented in a natural way by using inheritance in Python. Since the language resembles pseudocode, the implementation is easily understood and can be extended or implemented in other platforms.

We started from a very simple model, in which firms have no choices and the supply is based on the technological level of each firm. In this industry, the aggregate supply is obtained by cumulating the individual supply, the aggregate demand is fixed and no assumptions are made about individual consumers. Then, based on these characteristics, quantities and prices are calculated. This simple model serves as a base

class, where basic procedures for industry creation and equilibrium calculation are defined.

The next implementation is regarding to the NW82 model, in which we include two firms with fixed innovation policies. Some firms are innovators and others are imitators. Besides being able to innovate, firms are also allowed to update their stock of capital. Using arrays in our implementation allows for the collection of every kind of data that is generated by the model. Collected data shows that insertion of innovation into the model leads to market concentration associated to the progression of time, a result that points that innovation is responsible for the market structure.

Extending further the already complex NW82 model, adaptive innovation policies are included. Extending the model in such a way allowed firms to change their innovation or imitation strategies depending on the current state of the model. Entry and exit of new firms were also included, allowing that firms with low stock of capital or performance to leave the industry whenever a threshold was not met or new firms to enter whenever the opportunity arose.

A final extension included patents, allowing firms to have exclusivity over their innovations for a period. Overall, qualitative results were quite similar to those found in the original papers by Winter: innovation leads to market concentration. However, in our implementation, unlike the original work, we have found out that patents increase the amount of entering innovators in an industry. Further investigation is necessary to investigate the cause behind this phenomenon.

One possibility to extend this model further and consequently increase its complexity would be the inclusion of a second industry. In that case, both industries could exchange information (through imitation, for instance), which would help giving structure to the innovation process. This idea visits the concept of national innovation systems and is further discussed in Andersen (2005).

References

ANDERSEN, E. S. Evolutionary Economics: Post-Schumpeterian Contributions. Routledge, 2005.

ANDERSEN, E. S., JENSEN, A. K., MADSEN, L. and Jørgensen, M. The Nelson and Winter Models Revisited: Prototypes for Computer-Based Reconstruction of Schumpeterian Competition. **Danish Research Unit for Industrial Dynamics Working** Paper, n. 96-5. DRUID, Aalborg University, 1996.

DOI: http://dx.doi.org/10.2139/ssrn.54247

ANDERSEN, E. S. Toward a Multiactivity Generalisation of The Nelson-Winter Model. **Nelson and Winter Conference**. DRUID, Aalborg University, 2001.

GRIMM, Volker and RAILSBACK, Steven F. Agent-Based and Individual-Based Modeling. Princeton University Press, 2019.

GUNARATNE, Chathika and GARIBAY, Ivan. NL4Py: Agent-Based Modeling in Python with Parallelizable Netlogo Workspaces. **CoRR: Computer Research Repository**, 2018.
URL: https://arxiv.org/abs/1808.03292

ISAAC, A. G. Simulating Evolutionary Games: a Python-based introduction. **Journal of Artificial Societies and Social Simulation**, v.23, n.3, p.8, 2008.
URL: http://jasss.soc.surrey.ac.uk/11/3/8.html

HOLLANDER, Myles and WOLFE, Douglas. Nonparametric Statistical Methods. John Wiley & Sons, New York, 1973.

NELSON, R. R. and WINTER, S. G. An Evolutionary Theory of Economic Change. Harvard University Press, 1982.

VALENTE, M and ANDERSEN, E. S. A hands-on approach to evolutionary simulation: Nelson and Winter models in the Laboratory for Simulation Development. **The Electronic Journal of Evolutionary Modeling and Economic Dynamics**, 2001.

WINTER, S. G. Schumpeterian Competition in Alternative Technological Regimes. **Journal of Economic Behavior & Organization**, v.5, n.3, p.287-320, 1984.
DOI: https://doi.org/10.1016/0167-2681(84)90004-0

WINTER, S. G. Patents and Welfare in an Evolutionary Model. **Industrial and Corporate Change**, v.2, n.2, p.211-231, 1993.
DOI: https://doi.org/10.1016/0167-2681(84)90004-0

Appendix A – A sensitivity analysis for the performance adjustment parameter

As mentioned in the section regarding the NW84 model, once a firm adapts its investing policies, a constant Δ is added to its performance. Adding this constant guarantees that a firm does not change its policies at every period, which otherwise would lead to a firm changing its policies in an inefficient way. Although all other parameters are thoroughly defined, the value for Δ is not defined in Winter (1984), which suggests that a small value should suffice. To verify this, a sensitivity analysis is going to be conducted in this appendix. A single parameter sensitivity analysis will hold constant all parameters of the model, except for the parameter of interest, in order to comprehend how sensible are the results of a model when the parameter of interest varies (Railsback and Grimm, 2019, p. 301).

The single parameter sensitivity analysis goes as follows: the parameter delta is going to vary from 0 to 0.95 by steps of 0.05. For each step, 100 simulations consisting of 100 periods are going to be done. The next step is to analyze results and compare if there any statistically significant differences in the variables that are generated by the model. To test difference in means a non-parametric test is used: the Kruskal-Wallis rank sum test (see Hollander and Wolfe (1973)).

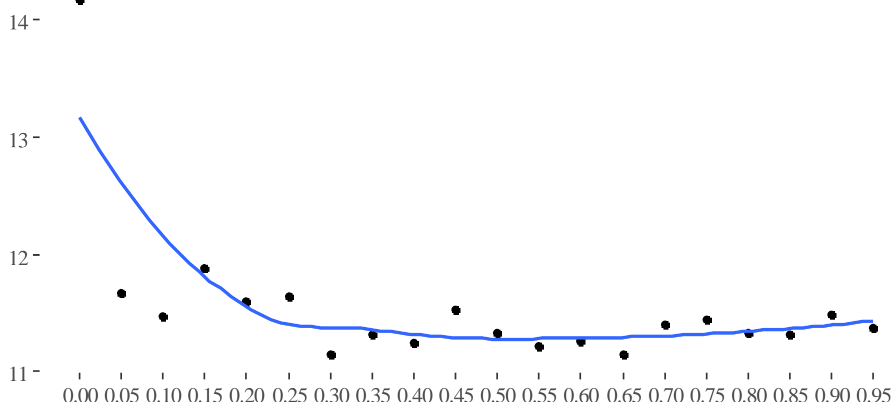
The results show that the model is robust to changes in the Δ parameter. The only variable affected, according to Table 2, is the number of firms. An explanation for this behavior is the fact that Δ is also responsible for indirectly determining if firms are going to leave the industry. As mentioned in the section regarding the NW84 model, low performance is one of the conditions for firm exit. The other condition is a capital that is lower than a threshold. For example, if Δ is zero, firms could be exiting the industry despite having a sufficient amount of capital to stay in it. Figure 8 shows the relationship between the mean amount of capital (y-axis) and the Δ value when firms are leaving the industry. The figure suggests that when Δ is equal to zero, the mean amount of capital at exit is higher than the amount of capital with other values of Δ. However, no significant change is seen in the values as Δ increases considering the set of analyzed values.

Table 2: Results from Kruskal-Wallis test

Variable	Statistic	p-value
Price	0.3938489	1
Mean Technology	0.2452635	1
Firm Number	37.165529	0.007565102
IHH	11.878954	0.890733901

Source: own elaboration.

Figure 8: Mean capital at exit and Δ value



Source: own elaboration.

To verify if the effect of increasing Δ is in fact negligible, we conduct another sensitivity analysis for the parameter. The difference is that instead of considering only values from 0 to 0.95, values from 0 to 100 are analyzed. The result from Table 3 is the estimation of a simple linear regression model of capital exit on the value of Δ. Although the value of the coefficient is negative, suggesting that increases in Δ are associated to decreases in the capital at exit, and it is statistically significant with a confidence level superior to 5%, the coefficient size turns out to be very small. Supposing that a value of capital at exit should be on average a value of 10, turning the relevance of performance in decision making of firms in an industry almost null, the Δ associated with it will be around 217.

Table 2: Simple linear regression of capital at exit on Δ

Term	Estimate	Std. Error	Statistic (t)	p-value
Intercept	10.93639	0.021643	505.3195729	0
Δ	-0.00429	0.000375	-11.42554071	0

Source: own elaboration.