

I mercati nel computer: primi passi per la costruzione di un simulatore didattico per la microeconomia

Rev. 20061205

Pietro Terna (terna@econ.unito.it)

Dipartimento di Scienze economiche e finanziarie G. Prato, corso Unione Sovietica 218bis
10134 Torino, Italia

Sommario

Si presenta una applicazione orientata alla didattica, ma con interesse anche per una discussione interna alla disciplina economica, contribuendo alla diffusione delle metodologie di simulazione. Una delle caratteristiche del modello è la esplicita utilizzazione di competenze cognitive nel comportamento dei produttori, a seconda che siano capaci o no di apprendimento. La razionalità dei consumatori è limitata o no dalla capacità di interagire solo con i produttori in raggio dato di distanza, oppure con tutti. La distribuzione dei produttori è differenziata per densità territoriale con uno schema “uno solo, alcuni, molti” a seconda delle zone. In tal modo si genera una differenziazione che costituisce una metafora delle barriere all’ingresso, avendo a disposizione gli scenari adatti per la nascita del monopolio, della concorrenza monopolistica e della concorrenza perfetta. I modelli sono sviluppati con NetLogo, <http://ccl.northwestern.edu/netlogo/>, ambiente di facile utilizzazione per chi usa il modello per l’apprendimento dell’economia. I modelli sono anche sviluppati in Python, <http://www.python.org/>, in via totalmente aperta, in quanto l’intero codice risulta abbastanza facilmente comprensibile.

Abstract

We introduce here an application directed to a didactic use, but with interest also for the discussion within the economic discipline, contributing to the spread of the simulation methodologies. One of the characteristics of the model is the explicit use of cognitive competences in the behavior of the producers, according to their learning capabilities. The rationality of the consumers is limited by the possibility of interacting with the producers only within a given distance. The distribution of the producers is differentiated by zones with the cases "one only, some, many". We obtain in this way a metaphor of the barriers to entry in a market, with the scenarios of monopolistic production, monopolistic competition and perfect competition. The models are developed with NetLogo, <http://ccl.northwestern.edu/netlogo/>, an environment with high usability for undergraduate students in economics. The models are also developed in Python, <http://www.python.org/>, in a totally open way, assuring the comprehensibility of the whole code.

Premessa di carattere economico

Un diffuso manuale introduttivo all'economia (Mankiw, 1999), dopo aver descritto i comportamenti dei consumatori, in modo realistico e comprensibile, introduce la classica descrizione degli effetti dei cambiamenti nei prezzi e nel reddito, con l'uso delle curve di indifferenza e delle equazioni di bilancio e infine

propone il quesito “le persone pensano davvero in questo modo?”

Il modo è appunto quello delle equazioni di bilancio, delle curve di indifferenza, intese come combinazioni di quantità di beni che sono equivalenti per il consumatore, e delle sottostanti caratteristiche di perfetta razionalità, conoscenza completa dei prezzi, capacità illimitata di calcolo; il tutto, fondato su una funzione di utilità più o meno complicata.

Mankiw risponde in modo negativo: “(. . .) *in fondo siamo tutti consumatori, prendiamo una decisione ogni volta che entriamo in un negozio e sappiamo benissimo che non lo facciamo confrontando vincoli di bilancio e curve di indifferenza*” e aggiunge “*Ma questa consapevolezza sul proprio modo di decidere è una prova della inesattezza della teoria? La risposta è no. La teoria delle scelte del consumatore non cerca di dare una descrizione accurata di come i singoli individui prendono le proprie decisioni: si tratta di un modello e (. . .) non è necessario che i modelli siano completamente realistici. Il modo migliore di considerare la teoria delle scelte del consumatore è come una metafora di come il consumatore prende le proprie decisioni*”.

As if e plausibilità

Ecco quindi comparire prepotentemente, in un libro destinato a studi introduttivi all’economia, il principio *as if* enunciato da Friedman (1953) sull’uso delle teorie, ancorché non realistiche, come se fossero vere, con il corollario che quanto più una teoria è irrealistica, tanto più può essere utile. In altri termini: non abbiamo bisogno della plausibilità.

Non ne abbiamo bisogno . . . se i modelli funzionano. Hann (1994) osserva che anche i fisici hanno dubbi sulla plausibilità dei modelli quantistici, ma producono, con quei modelli, risultati esatti “*fino all’ottava cifra decimale*”. Come sappiamo, i modelli degli economisti producono previsioni spesso sbagliate nel segno della variazione. Altro che (vedi sopra) non essere “completamente realistici”!

La plausibilità vuole invece che la semplicità dei comportamenti sia rispettata, secondo le indicazioni dello psicologo e premio Nobel dell’economia Herbert Simon, sui limiti alla razionalità e sull’uso di rassicuranti *routine* nei comportamenti economici e sociali. La complessità è dunque da cercare fuori degli agenti, nella loro interazione.

La complessità dell’andamento dei mercati non può quindi essere spiegata né analizzando i consumatori come singoli punti considerati a sé, né studiando la domanda quale fenomeno aggregato, ma solo tenendo conto delle azioni e delle interazioni dei e tra i consumatori che

scelgono prodotti diversi e i comportamenti e le interazioni delle singole imprese.

Il modello in NetLogo del monopolio, come esempio di simulazione

Si presenta una applicazione orientata alla didattica, ma con interesse anche per una discussione interna alla disciplina economica, contribuendo alla diffusione delle metodologie di simulazione. Una delle caratteristiche del modello è la esplicita utilizzazione di competenze cognitive nel comportamento dei produttori, a seconda che siano capaci o no di apprendimento.

Il simulatore proposto dispone, dal lato della domanda, di una popolazione di consumatori, disposti in uno spazio geografico; ogni consumatore ha uno o più prezzi di riserva al di sopra dei quali non compera; i prezzi di riserva sono stabiliti casualmente in un intervallo dato. In termini aggregati abbiamo una curva di domanda quasi standard, con la possibilità di valutare la rendita del consumatore e così via. Facendo invece riferimento ad uno spazio in cui i consumatori siano disposti casualmente, l'interpretazione possibile diventa realisticamente complessa.

La razionalità dei consumatori è limitata o no dalla capacità di interagire solo con i produttori entro una data distanza, oppure con tutti. I consumatori possono eventualmente muoversi nello spazio e comunicare tra loro.

La distribuzione dei produttori è differenziata per densità territoriale con uno schema "uno solo, alcuni, molti" a seconda delle zone. In tal modo si genera una differenziazione che costituisce una metafora delle barriere all'ingresso, avendo a disposizione gli scenari adatti per la nascita del monopolio, della concorrenza monopolistica e della concorrenza perfetta. Si può anche sviluppare, in prospettiva, il percorso di analisi dell'oligopolio, anche in parallelo rispetto alla storia del pensiero economico.

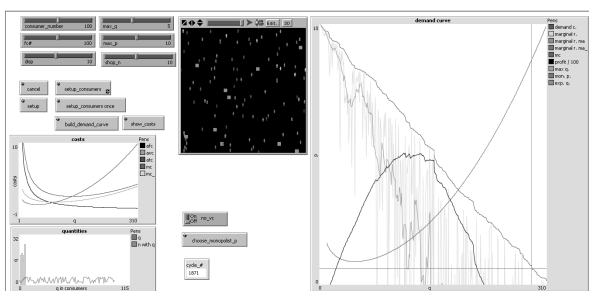


Figura 1: Vista di insieme del modello di monopolio.

I produttori si adattano o no al modello teorico atteso, nel breve e nel lungo periodo, a seconda della capacità di apprendimento di cui sono dotati.

Incrociando offerta, con e senza apprendimento, e domanda, con visione dello spazio di mercato limitata e no, si generano i diversi modelli didattici e, parzialmente, alcuni riscontri con il mondo reale.

Il modello è sviluppato con NetLogo, <http://ccl.northwestern.edu/netlogo/>, inteso come ambiente di relativamente facile utilizzazione per chi deve

programmare il modello e di facilissima utilizzazione, tramite bottoni, cursori e interruttori, per chi usa il modello per l'apprendimento dell'economia.

La struttura del modello

In figura 1 abbiamo una vista di insieme del modello in cui simultaneamente conosciamo, sulla sinistra, la struttura dei costi del monopolista, con la classica rappresentazione dei costi fissi unitari, dei costi variabili, dei costi medi e dei marginali, nonché la sequenza delle quantità di acquisto massimo di ognuno dei consumatori, anche sintetizzate nell'istogramma a sinistra.

Al centro, con l'ingrandimento in figura 2, abbiamo lo spazio di mercato con i consumatori e i "negozi", cioè i quadrati, dove i consumatori possono acquistare. I consumatori si muovono casualmente nel piano e acquistano solo se trovano un negozio; in quel caso confrontano i prezzi cui sono disponibili ad acquistare, per ciascuna delle unità di acquisto, il bene, compiendo l'operazione solo se il prezzo di vendita è inferiore a quello corrispondente alla disponibilità espressa internamente ad acquistare (ad es. ho deciso di acquistare tre unità del bene; per la prima sono disposto a spendere 10, per la seconda 8 e per la terza 6, con 10, 8 e 6 prezzi di riserva; se il prezzo è 7 acquisto due unità con una *rendita del consumatore* pari a 3 per il primo bene e a 1 per il secondo).

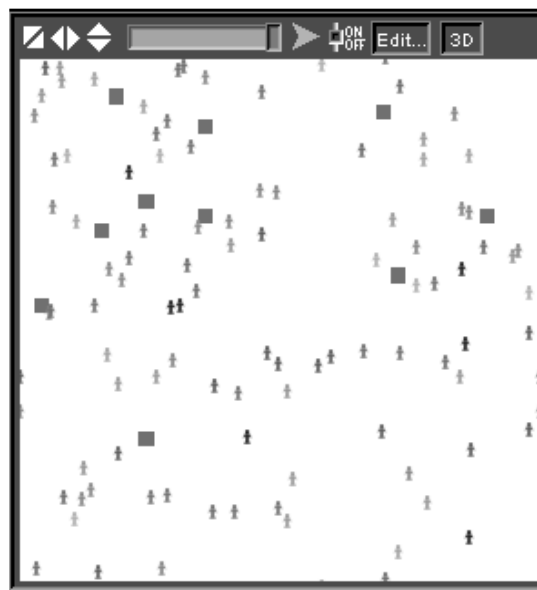


Figura 2: Lo spazio dei consumatori.

Sulla destra, in figura 3, abbiamo la curva di domanda, che si forma sulla base delle decisioni di quantità di acquisto dei consumatori e dei prezzi di riserva per ciascuna delle unità di acquisto, ordinate in sequenza decrescente di prezzo di riserva. Da questa curva di domanda molto realistica, continuamente modificata dalle scelte dei consumatori, deriva una curva di ricavo marginale molto frastagliata, perché la curva di domanda non è regolare.

Semplifichiamo con una media mobile la curva di domanda e otteniamo la curva frastagliata più marcata. In nero, a forma di parabola rivolta verso il basso abbiamo la

curva del profitto, che ovviamente risulta massima là dove curva di costo marginale e curva di ricavo marginale si incontrano.

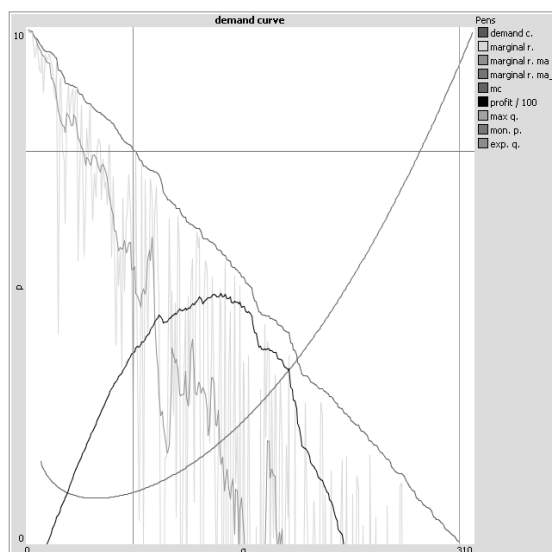


Figura 3: Curva di domanda, costi marginali, profitti.

Ma ... il monopolista non conosce la curva di domanda (la teoria pretende ciò, ma si tratta di una grande semplificazione); la curva di domanda è mutevole; i dati di acquisto si accumulano con il movimento dei consumatori nei negozi (i quadratini della figura 2) e il monopolista mantiene il prezzo stabile per intervalli di tempo predeterminati; infine modifica il prezzo, con prove ed errori, per cercare di massimizzare il profitto, inseguendo il risultato teorico.

Introducendo forme di apprendimento, come una stima della domanda sulla base dei dati via via raccolti e l'analisi dei tempi necessari ai consumatori per rispondere ad una offerta formulata, si può verificare il grado di maggiore o minore successo delle strategie del monopolista.

Aspetto altrettanto importante, dal punto di vista dello studente che usa il modello di simulazione per comprendere il tema del monopolio: si crea una interazione diretta tra le scelte del monopolista e quelle dei consumatori, senza le semplificazioni del modello teorico e con il realismo della complessità emergente dall'interazione tra gli agenti.

Evoluzione del modello

I modelli a seguire riguardano la concorrenza perfetta, la concorrenza monopolistica e, con un ambiente necessariamente diverso e più complesso, l'oligopolio.

Tutti i modelli sono o saranno disponibili all'indirizzo http://web.econ.unito.it/terna/simulazione_didattica.

In prospettiva i modelli saranno sviluppati anche con l'utilizzazione di StarLogo TNG (TNG= The Next Generation; reperibile a <http://education.mit.edu/starlogo-tng/>), ambiente che condivide, con alcune varianti, lo schema base di NetLogo (che deriva del resto da StarLogo), aggiungendo due rilevanti novità: una modalità di programmazione grafica (secondo l'affermazione *removing the syntax barrier - graphical programming*); la

capacità di sviluppare modelli che presentano punti di somiglianza con un cartone animato.

Uno schema in Python per un simulatore ad agenti generalizzato e direttamente accessibile

Il modello è sviluppato anche in Python, <http://www.python.org/>, in via totalmente aperta, in quanto l'intero codice risulta così accessibile ed abbastanza facilmente comprensibile. Dal sito: "Python® is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code."

La versione Python del simulatore didattico è preceduta dalla esemplificazione di una schematizzazione generale della programmazione per la simulazione, con un semplice esempio programmato (i) in plain Python; (ii) in Python a oggetti, mostrando così l'indipendenza del codice dalla numerosità degli agenti; (iii) infine, in Python a oggetti, con la gestione del tempo tramite un sistema di *scheduling*, riducendo la macchinosità del codice costruito con cicli di ripetizione. Si tratta dell'idea originaria sviluppata con lo costruzione di Swarm (www.swarm.org) a metà degli anni '90, secondo lo schema attribuibile a Chris Langton, riportato nel *tutorial* disponibile nel sito di Swarm (www.swarm.org); il *tutorial* è stato successivamente adattato a JavaSwarm (Staelin, 2000), versione di Swarm che consente l'accesso alla libreria delle funzioni del sistema stesso tramite Java.

Anche in questo caso, tutti i modelli sono o saranno a http://web.econ.unito.it/terna/simulazione_didattica.

Una precisazione di metodo e di terminologia

Utilizziamo una terminologia parzialmente impropria, richiamando, come nel *tutorial* citato, il C per indicare una programmazione del tutto senza oggetti; Java per introdurre l'uso degli oggetti; Swarm per indicare un uso sofisticato degli oggetti, sia con più strati (modello e osservatore) di codifica, sia con lo stratagemma dello *scheduling* per rendere più comprensibili i cicli di ripetizione della simulazione.

In tutti i modelli a seguire, uno o più agenti si spostano a caso su un piano. Si tratta di un *frame* generalissimo, in grado di contenere problemi complessi mantenendo la comprensibilità del codice. E' anche una strada diretta per comprendere la logica di Swarm, per poi sviluppare i problemi di maggiore complessità in quell'ambiente

SimpleCBug. Un solo agente, con il tempo gestito tramite un ciclo *for*, senza programmazione a oggetti.

```
#SimpleCBug.py
import random
```

```
def SimpleBug():
```

```

# the environment
worldXSize = 80
worldYSize = 80

# the bug
xPos = 40
yPos = 40

# the action
for i in range(100):
    xPos += randomMove()
    yPos += randomMove()
    xPos = (xPos + worldXSize) % worldXSize
    yPos = (yPos + worldYSize) % worldYSize
    print "I moved to X = ", xPos, " Y = ", yPos

# returns -1, 0, 1 with equal probability
def randomMove():
    return random.randint(-1, 1)

SimpleBug()

SimpleJavaBug. Il tempo è gestito con un ciclo for, con la programmazione a oggetti che crea un solo agente, come esemplare di una classe; è quindi semplice generarne altri e farli agire.

#SimpleJavaBug.py
import random

class Bug:
    def __init__(self, number, xPos, yPos, worldXSize = 80, worldYSize = 80):
        # the environment
        self.number = number
        self.worldXSize = worldXSize
        self.worldYSize = worldYSize
        # the bug
        self.xPos = xPos
        self.yPos = yPos
        print "Bug number ", self.number, \
            " has been created at ", self.xPos, ", ", \
            self.yPos

        # the action
        def randomWalk(self):
            self.xPos += randomMove()
            self.yPos += randomMove()
            self.xPos = (self.xPos + self.worldXSize) % \
                self.worldXSize
            self.yPos = (self.yPos + self.worldYSize) % \
                self.worldYSize
            # report
            def reportPosition(self):
                print "Bug number ", self.number, " moved to X = ", \
                    self.xPos, " Y = ", self.yPos

# returns -1, 0, 1 with equal probability
def randomMove():
    return random.randint(-1, 1)

nBugs = input("How many bugs? ")
bugList = [0] * nBugs
worldXSize= input("X Size of the world? ")
worldYSize= input("Y Size of the world? ")
length = input("Length of the simulation in cycles? ")

for i in range(nBugs):
    aBug = Bug(i, random.randint(0,worldXSize-1), \
        random.randint(0,worldYSize-1))
    bugList[i] = aBug

for t in range(length):
    for aBug in bugList:
        aBug.randomWalk()

```

```
aBug = Bug(1, 40, 40)
```

```

for i in range(100):
    aBug.randomWalk()
    aBug.reportPosition()

```

SimpleJavaManyBugs. Il tempo è gestito con un ciclo *for*, con la programmazione a oggetti che crea molti agenti, come esemplari di una classe; gli agenti sono inclusi in una collezione ed è quindi semplice farli agire.

```

#SimpleJavaManyBugs.py
import random

class Bug:
    def __init__(self, number, xPos, yPos, worldXSize = 80, worldYSize = 80):
        # the environment
        self.number = number
        self.worldXSize = worldXSize
        self.worldYSize = worldYSize
        # the bug
        self.xPos = xPos
        self.yPos = yPos
        print "Bug number ", self.number, \
            " has been created at ", self.xPos, ", ", \
            self.yPos

        # the action
        def randomWalk(self):
            self.xPos += randomMove()
            self.yPos += randomMove()
            self.xPos = (self.xPos + self.worldXSize) % \
                self.worldXSize
            self.yPos = (self.yPos + self.worldYSize) % \
                self.worldYSize
            # report
            def reportPosition(self):
                print "Bug number ", self.number, " moved to X = ", \
                    self.xPos, " Y = ", self.yPos

# returns -1, 0, 1 with equal probability
def randomMove():
    return random.randint(-1, 1)

nBugs = input("How many bugs? ")
bugList = [0] * nBugs
worldXSize= input("X Size of the world? ")
worldYSize= input("Y Size of the world? ")
length = input("Length of the simulation in cycles? ")

for i in range(nBugs):
    aBug = Bug(i, random.randint(0,worldXSize-1), \
        random.randint(0,worldYSize-1))
    bugList[i] = aBug

for t in range(length):
    for aBug in bugList:
        aBug.randomWalk()

```

```
aBug.reportPosition()
```

SimpleSwarmModelBugs. Il tempo è gestito con uno *schedule* applicato in modo flessibile ad un ciclo *for*, con programmazione a oggetti che crea molti agenti all'interno di un modello considerato come strato dell'applicazione; il modello e la classe sono esterni al codice di avvio della simulazione.

```
#SimpleSwarmModelBugs.py
import ModelSwarm

nBugs = input("How many bugs? ")
worldXSize= input("X Size of the world? ")
worldYSize= input("Y Size of the world? ")
nCycles = input("How many cycles? (0 = exit) ")

modelSwarm = ModelSwarm.ModelSwarm(nBugs,
nCycles, worldXSize, worldYSize)

# create objects
modelSwarm.buildObjects()

# create actions
modelSwarm.buildActions()

# run
modelSwarm.run()

print
print "Simulation stopped after ", nCycles, " cycles"

###

#ModelSwarm.py
import Bug
import ActionGroup
import random

class ModelSwarm:
    def __init__(self, nBugs, nCycles, worldXSize = 80,
worldYSize = 80):
        # the environment
        self.nBugs = nBugs
        self.bugList = []

        self.nCycles = nCycles
        self.worldXSize= worldXSize
        self.worldYSize= worldYSize

    # objects
    def buildObjects(self):
        for i in range(self.nBugs):
            aBug = Bug.Bug(i,
random.randint(0,self.worldXSize-1), \
random.randint(0,self.worldYSize-1))
            self.bugList.append(aBug)
        print

    # actions
    def buildActions(self):
```

```
self.actionGroup1 = ActionGroup.ActionGroup
("move")
def do1(self):
    random.shuffle(self.bugList)
    for aBug in self.bugList:
        aBug.randomWalk()
    self.actionGroup1.do = do1 # do is a variable linking
a method

self.actionGroup2 = ActionGroup.ActionGroup ()
def do2(self, nCycles):
    if nCycles % 3 == 0:
        print "Time = ", nCycles
        for aBug in self.bugList:
            aBug.reportPosition()
    self.actionGroup2.do = do2 # do is a variable linking
a method

# schedule
self.actionGroupList = ["move", "talk"]

# run
def run(self):
    for n in range(self.nCycles):

        for s in self.actionGroupList:

            if s == "move":
                self.actionGroup1.do(self) # self here is the
model env.

                # not added automatically
                # being do a variable

            if s == "talk":
                self.actionGroup2.do(self, n)

###

#Bug.py
import random

class Bug:
    def __init__(self, number, xPos, yPos, worldXSize =
80, worldYSize = 80):
        # the environment
        self.number = number
        self.worldXSize = worldXSize
        self.worldYSize = worldYSize
        # the bug
        self.xPos = xPos
        self.yPos = yPos
        print "Bug number ", self.number, \
" has been created at ", self.xPos, ", ",
self.yPos

    # the action
    def randomWalk(self):
        self.xPos += randomMove()
        self.yPos += randomMove()
        self.xPos = (self.xPos + self.worldXSize) %
self.worldXSize
```

```

        self.yPos = (self.yPos + self.worldYSize) %
self.worldYSize
    # report
    def reportPosition(self):
        print "Bug number ", self.number, " moved to X = ",
\
            self.xPos, " Y = ", self.yPos

# returns -1, 0, 1 with equal probability
def randomMove():
    return random.randint(-1, 1)

###

#ActionGroup.py

class ActionGroup:
    def __init__(self, groupName = " "): # the name is
optional
        self.groupName = groupName

    # reporting name
    def reportGroupName(self):
        return self.groupName

```

SimpleSwarmObserverBugs. Il tempo è gestito con uno *schedule* applicato in modo flessibile ad un ciclo *for* del modello e ad uno dell'osservatore, con programmazione a oggetti che crea molti agenti all'interno di un modello considerato come strato dell'applicazione, a sua volta contenuto in uno strato che rappresenta l'osservatore; l'osservatore, il modello e la classe sono esterni al codice di avvio della simulazione. A questa impostazione non è difficile aggiungere capacità di rappresentazione grafica (non riportate qui).

```

#SimpleSwarmObserverBugs.py
import ObserverSwarm

observerSwarm = ObserverSwarm.ObserverSwarm()

# create objects
observerSwarm.buildObjects()

# create actions
observerSwarm.buildActions()

# run
observerSwarm.run()

###

```

```

#ModelSwarm.py
import Bug
import ActionGroup
import random

class ModelSwarm:
    def __init__(self, nBugs, worldXSize = 80, worldYSize
= 80):
        # the environment
        self.nBugs = nBugs
        self.bugList = []

        self.worldXSize= worldXSize
        self.worldYSize= worldYSize

    # objects
    def buildObjects(self):
        for i in range(self.nBugs):
            aBug = Bug.Bug(i,
random.randint(0,self.worldXSize-1),
random.randint(0,self.worldYSize-1),
self.worldXSize,
self.worldYSize)
            self.bugList.append(aBug)
        print

    # actions
    def buildActions(self):

        self.actionGroup1 = ActionGroup.ActionGroup
("move")
        def do1(self):
            random.shuffle(self.bugList)
            for aBug in self.bugList:
                aBug.randomWalk()
            self.actionGroup1.do = do1 # do is a variable linking
a method

        self.actionGroup2 = ActionGroup.ActionGroup ()
        def do2(self, nCycles):
            if nCycles % 3 == 0:
                print "Model time = ", nCycles
                for aBug in self.bugList:
                    aBug.reportPosition()
            self.actionGroup2.do = do2 # do is a variable linking
a method

        # schedule
        self.actionGroupList = ["move", "talk"]

    # step
    def step(self, n):

        for s in self.actionGroupList:

            if s == "move":
                self.actionGroup1.do(self) # self here is the
model env.

                # not added automatically
                # being do a variable

```

```
if s == "talk":
    self.actionGroup2.do(self, n)
```

Futuri sviluppi

I futuri sviluppi di questo lavoro vanno in una direzione ambiziosissima: la preparazione di un manuale di microeconomia con contenuti tradizionali, affiancando la simulazione ad agenti agli argomenti tradizionali introdotti.

Opere molto importanti in questa direzione non mancano, ma tutte con un livello di ingresso non adatto ad uno studente che affronta un corso introduttivo e nessuna con una chiara adesione al paradigma della simulazione ad agenti.

References

- Friedman M. (1953). The Methodology of Positive Economics. *Essay in positive economics* (pp. 3-43). Chicago University of Chicago Press.
- Hahn F. (1994). An Intellectual Retrospective. *Quarterly Review*, 245-258.
- Mankiw N.G. (1999). *Principi di economia*. Bologna: Zanichelli, Bologna.
- Staelin C. P. (2000). *jSIMPLEBUG: a Swarm tutorial for Java*. Bozza on line, disponibile ad esempio a <http://eco83.econ.unito.it/swarm/materiale/jtutorial/JavaTutorial.zip>.