UNIVERSITA' DEGLI STUDI DI TORINO

Facoltà di Economia

Corso di Laurea in Finanza Aziendale e dei Mercati Finanziari

TESI DI LAUREA

Un simulatore di Borsa con dati reali per l'analisi quantitativa di derivative strategies

Relatore:Prof. Pietro Terna
Correlatore: Prof. Sergio Margarita

Candidato: Francesco Lovera

Anno Accademico 2009-2010

INDICE

INTRODUZIONE

La certezza della conoscenza completa di un sistema si esplica attraverso la capacità di fare previsioni, che in nessuna scienza, come nell'economia, risultano soggette all'insuccesso e all'imprevedibilità. Uno dei difetti della nostra società è quello di tendere a razionalizzare, ossia eliminare le espressioni irrazionali, con l'intento di descrivere qualsiasi fenomeno attraverso strutture prestabilite e certe. Questa tendenza facilmente ci rende soggetti ad errore. Per questa ragione, gli scienziati, che più di ogni altro sono riusciti a comprendere che una delle strade percorribili per evitare di insistere sui medesimi errori è quella di non racchiudere un fenomeno in una semplificazione di ciò che dovrebbe essere.

Con il mio progetto di tesi, non cerco di andare contro quella che è la tendenza generale, appena descritta, nè di risolvere quesiti, a cui è più probabile che mai si giunga ad una risposta certa. Desidero semplicemente raccogliere la sfida della complessità, per addentrarmi in una metodologia di studio nuova, che certamente non mi garantirà la soluzione dei sistemi complessi, ma mi aiuterà nel fare congetture, ricercando da un punto di vista diverso rispetto a quello classico.

Ho cercato di creare un percorso ragionato, che fornisse al lettore tutti gli strumenti necessari per la comprensione teorica e tecnica del lavoro, che sarà presentato al fondo della tesi.

Nel capitolo primo saranno ripresi i principi cardine della *Capital Market Theory* e i successivi studi che hanno dimostrato la loro inconsistenza. L'inefficienza, dimostrata dalla teoria classica, ci consentirà inoltre di comprendere la necessità di considerare nuovi sistemi e nuove procedure per poter catturare le dinamiche non lineari, generate nei mercati finanziari. A questo proposito, viene approfondita la visione dei mercati come *Complex Adaptive System,* dando una descrizione dei sistemi dinamici complessi, identificando proprietà chiave e attributi. Infine, sarà posto in evidenza il maggior potere descrittivo e previsionale, che la nuova teoria consente di avere rispetto alla visione classica.

Dopo aver introdotto la causa che spinge verso la necessità di approccio ad una nuova visione, per migliorare la comprensione di un fenomeno, che ad oggi risulta privo di leggi capaci di descriverlo e prevederlo, verrà approfondito il concetto di complessità nei modelli che consentono di coglierla, tentando di porre rimedio alle inefficienze degli studi classici.

Di complessità e simulazione si parlerà quindi nel capitolo secondo, cercando di esaltare i pregi e le qualità, che i modelli di simulazione ad agenti mettono a disposizione dei ricercatori, per tentare di individuare i fenomeni emergenti dalle interazioni tra agenti. Verrà effettuato un approfondimento sulle qualità e sulle caratteristiche, per dimostrare la versatilità metodologica che i modelli di simulazione forniscono al ricercatore.

Per consentire al lettore di ottenere una visione critica del lavoro svolto, priva di alcuna influenza, certamente positiva, da parte dell'autore stesso, il capitolo terzo offrirà una serie di dubbi mossi contro l'efficacia della metodologia ABM. Saranno, quindi, poste in evidenza una serie di critiche, alle quali sarà data risposta, avanzate da parte di coloro i quali considerano la nuova metodologia di non pari dignità rispetto alla tecnica standard, che da sempre ha governato la ricerca.

Nel capitolo terzo la pura teoria sarà abbandonata per dare spazio alla teoria applicata, che accompagnerà il lettore durante tutto il percorso verso la comprensione del modello di simulazione.

Tale percorso inizia nel capitolo quarto, in cui sarà spiegato nel dettaglio una parte del protocollo di programmazione Slapp, con il duplice scopo di introdurre sia il linguaggio informatico e la struttura del programma, sia di comunicare il rigore metodologico che è stato seguito dall'autore nella creazione del simulatore.

Per Slapp si è fatto riferimento al percorso, che deve essere seguito per la creazione di un programma, con un approccio generale, che richiama solamente aspetti teorici ancora lontani dal simulatore; con il capitolo quinto, invece, si entra nel cuore della programmazione tecnica, che troverà il suo apice nel capitolo sesto alla spiegazione del manuale utente.

Nel capitolo quinto si spiegheranno, quindi, i tecnicismi utilizzati nella programmazione ad oggetti che rappresenta una tecnica di rappresentazione della realtà, ricreata da semplici unità informatiche interagenti. Saranno inoltre forniti elementi teorici relativi alle proprietà di questo tipo di programmazione e saranno spiegate particolari funzioni e variabili, che di seguito troveremo nel codice del simulatore.

Il capitolo sesto sarà dedicato completamente alla spiegazione del codice, evidenziando le funzioni programmate che caratterizzano il comportamento degli agenti e regolano il funzionamento del mercato simulato. Tutte le informazioni necessarie alla comprensione dei risultati delle simulazioni e ai comportamenti degli agenti saranno quindi sicuramente acquisite attraverso la lettura di questo capitolo.

Con il capitolo settimo si arriva alla visualizzazione dei risultati generati dalle interazioni tra i vari agenti componenti il mercato. La strutturazione che è stata data agli esperimenti ha l'obiettivo di evitare di ottenere effetti che non abbiano una corrispondente spiegazione teorica relativa alle cause. Per questa ragione, un limitato numero di classi di agenti sarà fatto agire contemporaneamente. Come sarà possibile approfondire leggendo il capitolo, esso è sostanzialmente diviso in due parti.

Nella prima parte si cerca di dimostrare il reale funzionamento del modello ed in particolare di due categorie di agenti quali *random* e *variation*, che saranno sempre

inserite come base per i futuri esperimenti di negoziazione tra agenti. In particolare, di queste due classi si vuole ottenere che la prima garantisca ai prezzi simulati una distribuzione *random walk* e la seconda che abbia il potere di condizionare le serie di prezzi, avvicinandoli a dati reali. L'intelligenza di cui è stato dotato l'agente *variation,* quindi la capacità di realizzare offerte proporzionate ai dati storici ricavati dell'indice Ftse, permetterà di ricavare i risultati più significativi dalle simulazioni.

Nella seconda parte del capitolo si vedrà invece come la competizione nel mercato con altri agenti, quali, *volume*, che agiscono in base ai volumi di transazione, *covered*, che agiscono spinti dalla volontà di effettuare strategie di negoziazione tramite opzioni, *best offer*, che agiscono spinti dalla volontà di concludere un contratto a qualsiasi prezzo e *trend agent*, che agiscono seguendo il trend di mercato, condizionerà le distribuzioni delle serie di prezzo. Le strategie di negoziazione tra le varie categorie di agenti entrerà in competizione generando comportamenti aggregati complessi che produrranno risultati a cui si cercherà di dare spiegazione.

Nel capitolo 8 riporterò il codice ufficiale del programma utilizzato per le simulazioni. A questo capitolo faranno seguito le conclusioni del lavoro, in cui sarà dato spazio ad alcune idee su possibili sviluppi futuri della tesi.

Capitolo 1

RIVISITANDO LA TEORIA DEL MERCATO EFFICIENTE: IL MERCATO DEI CAPITALI COME UN *COMPLEX ADAPTIVE SYSTEM*

Lo studio delle dinamiche evolutive dei sistemi ha coinvolto da sempre studiosi provenienti da diverse discipline scientifiche. L'interesse nel dimostrare la reale efficacia di modelli tradizionali, spesso rivelatisi incapaci di cogliere adeguatamente le dinamiche comportamentali dei sistemi, e la volontà di dare una spiegazione a nuovi fenomeni osservati nelle evoluzioni degli studi, hanno spinto gli studiosi alla scoperta della complessità.

La scoperta della complessità, inoltre, rimanda la totalità delle scienze a considerare seriamente il fatto che non solo possono cambiare le domande e le risposte, ma anche possono cambiare i tipi di domande e risposte attraverso le quali si definisce l'indagine scientifica.

Ripercorrendo l'evoluzione delle scienze si è, così, passati da una visione meccanicistica[1] e lineare fondata sui principi di causa ed effetto, ispirata da Newton[2], ad una concezione non lineare e complessa, in cui la visione lineare rappresenta soltanto uno dei molteplici stati in cui può transitare un sistema.

Trattando la teoria dei mercati finanziari, l'ipotesi di mercato efficiente, secondo una visione lineare, ha iniziato ad essere posta in dubbio all'inizio degli anni '60 da sostenitori della non linearità dei mercati finanziari. Fama [1965], dopo approfondite analisi sui rendimenti giornalieri dei titoli, fu uno dei primi studiosi a contrastare fortemente il paradigma di linearità[3] dei mercati, ossia l'ipotesi di *random walk* (ritorni distribuiti normalmente e serialmente indipendenti). Egli osservò una distribuzione leptocurtica dei rendimenti (spostamenti verso destra del valore medio rispetto alla distribuzione normale, con frequenze molto più elevate e fenomeni di code spesse – *fat tails*), la quale determina, quindi, l'impossibilità di interpretare il mercato basandosi sugli assunti del mercato efficiente. Questo portò alla nascita di numerose teorie e numerosi strumenti, finalizzati allo studio di dinamiche non lineari del mercato finanziario, considerato come sistema dinamico complesso.

In questo capitolo saranno ripresi i principi cardine della *Capital Market theory* e i successivi studi che hanno dimostrato la loro non consistenza. L'obbiettivo della stesura di questo capitolo sarà raggiunto successivamente, quando saranno posti in evidenza i principi alla base del *Complex Adaptive System*. Inoltre, saranno confrontate le previsioni, elaborate con la nuova teoria, rispetto a quanto emerso dal

[1] Il meccanicismo, concetto nato nel '600, è un termine filosofico utilizzato per indicare una concezione del mondo in cui tutto accade, sia nel campo della materia, sia nel campo dello spirito, per cause meccaniche, non legate ad un fine superiore preordinato

[2] In Peters (1996:136-137) si legge: "Newtonian physics is based on linear relationships between variables. It assumes that:

- For every cause, there is a direct effect.
- All system seek an equilibrium where the system is at rest.
- Nature is orderly.

The clock is the supreme symbol of Newtonian physics. The parts integrate precisely and in perfect harmony toward a predictable outcome. Newtonian physics was the ultimate achievement of the 18th Century "Age of Reason" […] Through mathematics, we were finally able to understand how nature acted on bodies in motion, and how these bodies interacted".

[3] In Peters (1996:27-28) si legge: "The linear paradigm says, basically, that investors react to information in a linear fashion. That is, they react as information is received; they do not react in a cumulative fashion to a series of events. The linear view is built into the rational investor concept, because past information has already been discounted in security prices. Thus, the linear paradigm implies that returns should have approximately normal distributions and should be independent. The new paradigm generalizes investor reaction to accept the possibility of nonlinear reaction to information, and is therefore a natural extension of the current view".

comportamento dei mercati. Infine, saranno espresse considerazioni sul potere previsivo che i nuovi studi conferiscono, rispetto agli studi classici, ai mercati finanziari.

1.1 STANDARD CAPITAL MARKET THEORY

La maggior parte degli studi sviluppati in Economia nasce da un assunto: l'Economia come sistema in equilibrio, quindi bilancio tra domanda e offerta, rischio e rendimento, prezzo e quantità.

Articolata da Marshall [1890], questa visione deriva dall'idea che l'Economia sia una scienza consanguinea alla fisica newtoniana, la quale pone un chiaro collegamento tra causa ed effetto e implica prevedibilità. Egli sostiene che, quando un sistema in equilibrio è colpito da uno "shock esogeno", come notizie riguardanti default o un sorprendente taglio dei tassi di interesse da parte delle banche centrali, ha la tendenza ad assorbire l'impatto, ritornando in breve tempo ad uno stato di equilibrio.

Molti sistemi, sia in natura sia in economia, non sono in equilibrio ma piuttosto in continuo mutamento.

La *Capital Market theory*, largamente sviluppata negli ultimi cinquant'anni, poggia su alcune assunzioni di base, quali mercati efficienti e razionalità degli investitori.

Riguardo *l'efficient market hypothesis*, il maggior numero di studi è stato condotto sul tema dell'efficienza informativa. La definizione di efficienza informativa deriva da un famoso contributo di Fama [1970], integrato dalla successiva evoluzione teorica da parte di Leroy[4] [1976], in cui il mercato è definito efficiente, qualora i prezzi dei titoli riflettano pienamente, in ogni momento e in modo corretto, tutte le informazioni disponibili.

Questo significa che i prezzi si allineano in tempi praticamente nulli al giungere di nuove informazioni, creando una situazione di costante equilibrio; i movimenti nei

[4]Noto economista che, attraverso il suo lavoro, convinse lo stesso Fama [1976] ad integrare successivamente la definizione originaria di *efficienza dei mercati,* aggiungendo la locuzione "in modo corretto", riferita all'interpretazione da parte del mercato delle informazioni disponibili

prezzi sono, quindi, dovuti al sopraggiungere di nuove informazioni e non ad un ritardo di aggiustamento verso l'equilibrio.

Quindi, i prezzi di mercato devono essere tali da fornire, a chi utilizza tutte le informazioni disponibili, un rendimento atteso pari a quello di equilibrio; tale rendimento comprende il costo per l'acquisto di informazioni (Fama[1991]).

Dopo il celebre lavoro di Fama, una serie di lavori collegati tentarono di fornire spiegazioni più precise della natura dell'equilibrio, implicito nell'efficienza; essi sfociarono, in una definizione piuttosto curiosa data da Beaver [1981]:

"un mercato è efficiente se i suoi prezzi corrispondono a quelli di un'altra economia del tutto uguale a quella considerata, salvo per il fatto che le informazioni rilevanti sono pubbliche, quindi note a tutti".

Un'altra definizione interessante, fornita da Lathman [1986] e più restrittiva rispetto alla precedente, considera efficiente un mercato in cui il fatto di rendere pubbliche le informazioni non modifichi né i prezzi dei titoli, né i portafogli individuali.

In realtà, secondo la critica, queste ulteriori definizioni non hanno conferito alcun valore aggiunto evidente, dal punto di vista empirico, rispetto alla definizione originaria di Fama. In particolare, è difficile ravvisare quale possa essere il contributo offerto da un approccio, basato su un mondo *altrimenti identico*.

Questo nuovo approccio, inoltre, equivale ad un'indagine sulle proprietà dell'informazione, non dei mercati; ciò contraddice lo spirito del concetto originario di efficienza e del filone principale della letteratura empirica, che hanno sempre interpretato l'efficienza come una proprietà dei mercati.

Per queste ragioni, il contributo di Fama [1970] rimane ancora oggi il punto di riferimento più importante in tema di efficienza dei mercati finanziari. Si distinguono tre gradi di informazione riflessa nel prezzo:

1. efficienza in forma debole: i prezzi riflettono tutte le informazioni che si possono estrarre dall'andamento passato dei prezzi;

2. efficienza in forma semiforte: i prezzi riflettono anche le altre informazioni, pubblicamente disponibili;

3. efficienza in forma forte: i prezzi riflettono anche le altre informazioni, di tipo privato, che non sono disponibili a tutti.

Il concetto fondamentale, espresso dalla teoria, è quello che non possono essere create strategie sistematiche di *trading*, che diano rendimenti superiori a quelli di equilibrio. In pratica, qualsiasi regola operativa (*trading rule*), basata su informazioni disponibili a tutto il mercato, non potrà fornire rendimenti superiori a quelli altrimenti ottenibili con un comportamento da *cassettista*[5].

Collegata alla teoria del mercato efficiente, vi è la "*random walk theory*", la quale implica che i cambiamenti di prezzo delle attività finanziarie siano indipendenti l'uno dall'altro. In accordo con questa teoria, un cambiamento di prezzi deriva dal giungere di nuove informazioni al mercato, le quali sono, per definizione, casuali. Un'assunzione alla base dell'attività di trading è che gli investitori possono considerare i rendimenti attesi come variabili identicamente e indipendentemente distribuite.

Solitamente, i creatori di modelli finanziari assumono che il cambiamento nei prezzi degli stock sia normalmente o log-normalmente distribuito.

Altro pilastro della *Capital Market theory* è il concetto di razionalità degli investitori.

Esso implica che gli investitori cerchino, in ogni modo, di massimizzare il proprio benessere partendo dalle informazioni a loro disposizione, siano esse naturali o istituzionali, e dalla personale capacità di raggiungere determinati obiettivi. In altre parole, tali investitori perseguono il maggior numero di obiettivi,cercando di realizzarli completamente, massimizzando costi e profitti.

La razionalità attribuita ad un soggetto si fonda su tre principali aspetti:

1. il soggetto dispone di funzioni di preferenza e capacità di comprensione del bene da cui egli può ricavare maggiore soddisfazione, creando una classifica di preferenze tra i beni stessi
2. il soggetto ha capacità di massimizzare il suo profitto, relazionando le utilità con le risorse disponibili
3. il soggetto ha capacità di analisi e previsione, in funzione dell'ambiente in cui si trova e delle azioni intraprese dai suoi concorrenti

[5] Con questa espressione si vuole indicare il comportamento di un soggetto che, dopo aver acquistato un titolo finanziario, lo mantiene in portafoglio attuando una strategia a lungo termine.

La massimizzazione di costi, profitti e utilità rappresenta una delle componenti della teoria di efficienza dei mercati.

La struttura del comportamento degli investitori è riflesso nel Capital Asset Pricing Model (CAPM).

Esso è un modello di equilibrio, proposto da Sharpe, in uno storico contributo del 1964, e indipendentemente sviluppato da Lintner [1965] e Mossi [1966]. Il CAPM stabilisce una relazione tra il rendimento di un titolo e il suo grado di rischio, misurato tramite un unico fattore, detto *beta*, che misura quanto il valore del titolo si muove in sintonia con il mercato. Matematicamente, il beta è proporzionale alla covarianza tra rendimento del titolo e andamento del mercato.

Questo modello implica una relazione lineare tra rischio e rendimento, ovvero, investitori razionali ricercano le maggiori possibilità di guadagno per ciascun livello di rischio.

Una critica a questa teoria è stata fatta da Stern (Stewart [1991]), il quale sostenne che non tutti i partecipanti ad un mercato massimizzano la propria efficienza, seguendo teorie e modelli economici. Il concetto di Stern è riassunto in una sua celebre frase "*If you want to know where a herd of cattle is heading, you need not interview every steer in the herd, just the lead steer".* L'idea di base è che, nel mercato, solamente un ristretto numero di agenti comprende modelli economici di società e investe di conseguenza. Saranno i "*lead steer*" a regolare prezzi e margini di guadagno. Perciò, le società non devono temere i comuni investitori, poiché i prezzi dei titoli che garantiscono i margini di guadagno e gli investimenti, in realtà, sono settati correttamente dall'opera dei "lead steer".

La metafora dei "lead steer" rappresenta un atteggiamento mentale centralizzato: l'efficienza del mercato è garantita da pochi agenti massimizzanti.

In seguito, emergerà come, in realtà, non vi è necessità di assumere la presenza di leader per raggiungere un'efficienza di mercato. Questo sarà un grande traguardo raggiunto dalla teoria che considera il mercato dei capitali come un *Complex Adaptive System.*

1.2 TEST SULLA CLASSICAL MARKET THEORY

Da molti anni sono stati sviluppati test a dimostrazione delle teorie di efficienza dei mercati. Tuttavia, emerge una notevole difficoltà nel testare le teorie economiche. La causa di ciò è legata al fatto che gli economisti, a differenza di altri scienziati, valutano le loro teorie sulla capacità di spiegare eventi passati e di prevedere eventi futuri.

Il primo punto critico nei confronti della *Classical Market theory* riguarda **i rendimenti dei titoli di mercato.**

Alcuni studi hanno dimostrato che i rendimenti dei prezzi non si distribuiscono normalmente, contrariamente a quanto suggerisce la *Capital Market theory*. I rendimenti, infatti, esibiscono una forte *curtosi,* cioè una maggiore distribuzione nelle code (fat-tails) e medie di guadagni superiori rispetto alle aspettative. Per definizione, la *curtosi* rappresenta un allontanamento dalla normalità distributiva, quindi, dati empirici mostrano fenomeni contrari rispetto a quanto studiato nella teoria. Questo significa che periodi di modesti cambiamenti sono caratterizzati da maggiore volatilità, rispetto alle previsioni, evidenziando un'alternanza di picchi nella distribuzione, sia positivi sia negativi (vedi Figura 1 e 2).

L'osservazione legata al fatto che i rendimenti degli *stock* non seguono una distribuzione normale e che il reale andamento dei rendimenti è diverso dalle previsioni, fatte seguendo la convenzionale teoria, in realtà non rappresenta una novità. Questo fu notato già da Fama[1965], uno dei padri della *Efficient Market theory*, dicendo: «If the population of price changes is strictly normal, on the average for any stock … an observation more than five standard deviations from the mean should be observed about once every 7,000 years. In fact such observations seem to occur about once every three to four years».

Il crollo di 22.6% avvenuto il 19 Ottobre 1987 nello *stock market* fu la dimostrazione di una di queste *fat-tail,* come sostenuto da Jeans Jackwerth e Rubinstein [1996]: «In un mondo sorretto dalla teoria di distribuzioni normali, un crollo di questa entità avrebbe registrato una probabilità talmente remota da risultare impossibile». La reazione che questo crash suscitò nel mondo accademico fu significativa. In una intervista, datata

1997, Fama, ad una domanda riguardante il crash avvenuto nei mercati nel 1987, rispose: «I think the crash in ’87 was a mistake».

FIGURE 1
FREQUENCY DISTRIBUTION OF S&P 500 FIVE-DAY RETURNS: NORMAL VERSUS ACTUAL (JANUARY 1968–FEBRUARY 2002)

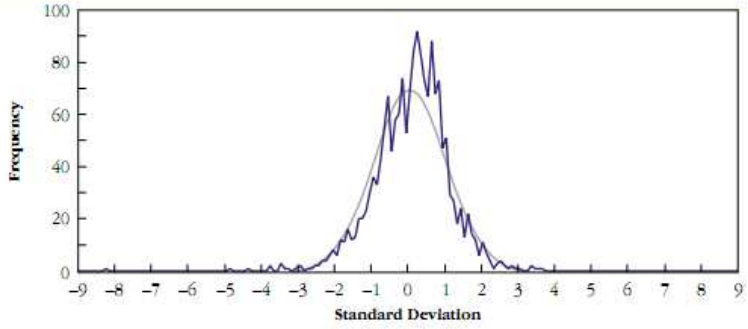
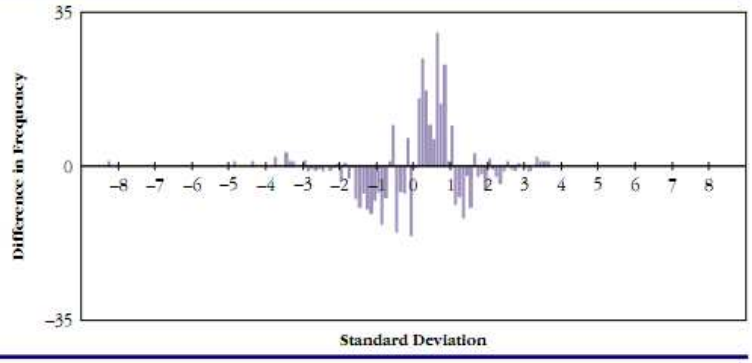


FIGURE 2
FREQUENCY DIFFERENCE: NORMAL VERSUS ACTUAL FIVE-DAY RETURNS (JANUARY 1968–FEBRUARY 2002)



Merton Miller offrì una spiegazione convincente al crash verificatosi, basandosi sulla teoria della razionalità degli investitori e citando, però, ricerche di Benoit Mandelbrot, matematico che già nel 1960 sottolineò il fatto che la volatilità dei prezzi azionari fosse troppo elevata per essere giustificata dall’uso di una distribuzione normale (Campbell, Lo, Mackinlay [1997]).

Il secondo punto critico verso la *Classical Market theory* riguarda **la distribuzione dei prezzi secondo una *random walk***.

Campbell, Lo e Mackinlay, dopo aver sviluppato test empirici, conclusero che i rendimenti finanziari, per alcuni gradi, sono prevedibili. Inoltre, altri studi sui mercati finanziari hanno dimostrato che esiste un comportamento dei mercati finanziari con memoria a lungo termine. Questo significa che le distribuzioni di rendimenti sono sia persistenti, cioè si ripetono nella storia, sia sostenuti nei trend, cioè elevati volumi.

Terzo punto critico riguarda **la relazione non lineare tra rischi e rendimenti**.

In test empirici, condotti nel 1992, riguardo al CAPM (analizzato per un periodo di tempo dal 1963 al 1990) permisero a Eugene Fama e Kenneth French di concludere che *"*i test fatti non supportano molte delle basilari previsioni del modello SLB (Sharpe-Lintner-Black), in cui la media dei rendimenti è positivamente correlata con il mercato*"* (*The Journal of Finance).*

Inoltre, Fama e French sostennero che altri due fattori, non-CAPM, quali *firm size*[6] e *market to book value*[7], fossero sistematicamente correlati con i rendimenti delle azioni, durante il periodo preso in considerazione. Tuttavia, Fama e French mantennero una "*rational asset-pricing framework*", ovvero identificarono fattori associati con vari rendimenti, assumendo che tali rendimenti fossero attribuibili al rischio.

Il quarto punto critico riguarda **la non razionalità degli investitori**.

Questa tematica ha subito nel tempo numerosi sviluppi. Il primo riguarda un elevato numero di dati, a dimostrazione del fatto che gli investitori compiono sistematici errori di giudizio (Thaler[1992]) .Uno dei migliori documenti, al riguardo, è la "*prospect theory*", la quale dimostra che le *individual risk preferences* sono profondamente influenzate dalle modalità di presentazione delle informazioni agli acquirenti "packaged"(Kahneman, Tversky [1979]).

Il secondo aspetto, messo in evidenza, riguarda il fatto che gli investitori commerciano molto più di quanto la teoria predice. Per spiegare gli effetti reali, generati dall'attività di trading, Fisher Black sviluppa la teoria del "*noise*" e del "*noise trading*". Black descrive il *noise* trading come quel comportamento assunto da determinati investitori che trattano il *noise* come se fosse informazione".

Il punto finale legato alla non razionalità degli investitori è legato al fatto che le persone solitamente operano utilizzando l'induzione[8], non la deduzione[9], per fare

---

[6] Concetto legato alle dimensioni delle società calcolato per capitalizzazione. E' stato registrato che in media titoli appartenenti a *small firm* hanno rendimenti superiori in fase di espansione economica rispetto a titoli appartenenti a *large firm.* Effetto contrario è stato registrato in situazioni di crisi economica.

[7] Rappresenta il rapporto tra la capitalizzazione complessiva di una società e il patrimonio netto della stessa società. Tale misura esprime un'idea di quanto il mercato valuta il patrimonio netto dell'azienda.

[8] Il metodo induttivo o induzione (dal latino "*inductio*") significa letteralmente "portar dentro", "chiamare a sé" e rappresenta un procedimento che partendo da singoli casi particolari cerca di stabilire una legge universale

decisioni economiche. Partendo dal presupposto che nessun individuo ha accesso a tutte le informazioni, gli investitori devono basare il loro giudizio non solamente su cosa essi conoscono, ma anche su cosa credono che gli altri pensino. Il fatto che gli investitori prendono alcune decisioni, seguendo la regola del pollice[10], comporta una fondamentale indeterminazione in economia (Arthur[1995]). Il prezzo degli *asset* è una buona approssimazione per le aspettative aggregate. Tuttavia, se un numero sufficiente di agenti adotta regole di decisione, basate sull'attività dei prezzi, generati sia casualmente sia coscientemente, il risultante trend dei prezzi può essere autorinforzato (*self-reinforcing*).

Nonostante le sue evidenti lacune, la teoria classica ha consentito di comprendere molto a riguardo alle modalità operative dei mercati dei capitali.

L'introduzione di una nuova teoria, unita alla potenza dei calcolatori, necessaria per modellare, può sancire l'inizio di una nuova era di comprensione del mercato dei capitali. Una nuova teoria, però, non si deve limitare a spiegare e dimostrare i limiti della teoria classica, bensì deve aggiungere potere previsionale.

1.3 IL MERCATO DEI CAPITALI COME UN COMPLEX ADAPTIVE SYSTEM

Questa nuova teoria è agganciata a tutto ciò che noi conosciamo delle altre scienze quali fisica, biologia e appare essere molto ben descrittiva di ciò che accade negli attuali mercati dei capitali. Come prima cosa, proseguendo con questo capitolo, sarà fornita una descrizione del *Complex Adaptive System*, identificando proprietà chiave e attributi. Inoltre, saranno confrontate le predizioni fatte con la nuova teoria rispetto al comportamento dei mercati. Infine, saranno fatte considerazione sulla teoria a riguardo delle informazioni aggiuntive per conoscere maggiormente i mercati, cercando di preservare il principi trasmessi della teoria classica.

---

[9] Il metodo deduttivo o deduzione è il contrario del metodo induttivo. Il termine significa letteralmente "condurre da" e rappresenta un processo razionale che fa derivare una certa conclusione da premesse generiche, dentro cui quella conclusione è implicita.

[10] Processo di decisione necessario in presenza di un numero elevato di dati, possibili scelte e tempo a disposizione. Queste regole empiriche, documentate scientificamente a partire dai primi anni '70, se da un lato permettono di semplificare il lavoro di ragionamento, dall'altro possono portare a conclusioni errate o troppo semplicistiche visto l'esiguo tempo e l'incompletezza del processo decisionale. Gli errori che ne derivano prendono il nome di bias.

1.4 UN NUOVO MODELLO DI INTERAZIONE TRA GLI INVESTITORI

Prendiamo due persone, poniamole in una stanza e chiediamo loro di commerciare una *commodity*: non molto accade. Aggiungendo un numero limitato di persone nella stanza noteremo come l'attività aumenta, ma l'iterazione rimane relativamente inesistente. Il sistema è molto statico, privo di vita, e riflette ciò che noi vediamo nel mercato dei capitali. Tuttavia, come aggiungiamo più persone al sistema, qualcosa degno di nota accade: esso si trasforma in un sistema, così soprannominato *Complex Adaptive System*, ricco di nuove caratteristiche. In maniera tangibile, il sistema diventa più complesso delle singole parti che esso comprende. Questa transazione, solitamente chiamata "*self-organized criticality*", avviene senza progettazione nè aiuto da parte di alcun agente esterno, ma risulta, piuttosto, dalla diretta iterazione tra agenti e sistema.[11]

Il fisico Bak si serviva di un mucchio di sabbia per illustrare le criticità dell'autorganizzazione. Appoggiando granelli di sabbia sopra una superficie piana, essi si assestano all'incirca in corrispondenza del punto su cui essi cadono. Il processo può essere modellato attraverso la fisica classica. In seguito alla formazione di un mucchio di sabbia, i granelli posti in cima ad esso iniziano a scivolare piano, fino ad instaurare una sorta di equilibrio. Questo equilibrio è precario ed è sufficiente un piccolo "disturbo" esterno per generare il crollo della montagna di sabbia. Non è possibile comprendere questi cambiamenti studiando semplicemente il comportamento di ogni singolo granello di sabbia. Il sistema, infatti, possiede delle proprietà che sono svincolate dalle quelle di ogni singolo granello.

Una caratteristica centrale del *Complex Adaptive System* è il "punto critico", cioè trovare e prevedere quel punto di rottura risultante dall'accumulo di tanti piccoli stimoli, come il peso accumulato di molti grani di sabbia provocano una valanga.

[11] Per una comprendere meglio le criticità di un sistema auto-organizzato, vedere Per Bak, How Nature Works (New York: Springer-Verlag New York, 1996). Infatti, biologi teorici tra I quali Stuart Kauffman ha teorizzato che un simile processo avvenne all'inizio della vita. Per approfondimenti vedere Stuart Kauffman, At Home in the Universe: The Search for Laws of SelfOrganization and Complexity (Oxford: Oxford University Press, 1995).

Questo implica il fatto che elevate fluttuazioni siano endogene al sistema. Cercare infatti, specifiche cause a fenomeni di vasta scala risulta essere un'operazione inutile.

Un sistema adattivo complesso esprime un numero essenziale di proprietà e meccanismi[12].

Aggregazione è l'essenza della complessità, cioè comportamenti complessi nascono dalla sommatoria di tante piccole iterazioni compiute da agenti semplici. Un esempio di questo fenomeno è quello rappresentato di una colonia di formiche. Nel caso in cui si intervistasse ogni singola formica, per cercare di comprendere il sistema osservato, si giungerebbe al risultato di un semplice elenco di compiti. Tuttavia, dall'unione dei semplici compiti di tutte le formiche crea un sistema funzionale e adattivo. Considerando il mercato dei capitali, il comportamento del mercato "emerge" dall'iterazione tra gli investitori. Questo è ciò che Adam Smith definì "*invisible hand*".

Adaptive decision rules. Gli agenti, inseriti in un sistema complesso, ricevono informazioni dall'ambiente esterno e le combinano con le informazioni che ricavano dalla loro personale esperienza, ricavandone regole di decisione. Inoltre, nuove regole di decisione competono con vecchi principi e processi di decisione. Questo processo permette l'adattamento, il quale spiega il principio di "*adaptive*" presente nella definizione di *Complex Adaptive System*. Si possono considerare regole di trading individuali e regole di investimento casuali "thumb" come regole di decisione nel mercato dei capitali.

E' importante notare come il concetto di regole di decisioni adattive è consistente con la drastica riduzione delle anomalie di mercato. La riduzione delle anomalie di mercato sono generate della continua ricerca, da parte degli investitori, di opportunità di profitto, cercando di raffinare regole di decisione per competere nei mercati per il maggior tempo possibile.

Non Linearità. In un modello lineare, il valore dell'intero equivale alla somma delle parti. In un sistema non lineare, il comportamento aggregato è molto più complesso di ciò che le singole parti dovrebbero predire. Questo punto può essere spiegato con un semplice modello preda/predatore. Date alcune variabili di base (predatori e prede in

[12] Ciò che sarà spiegato da questo punto in avanti, all'interno di questa sezione di capitolo, riguarda un lavoro di John H. Holland intitolato Hidden Order: How Adaptation Builds Complexity (Reading, MA: Helix Books, 1995)

una determinata area, tasso di iterazione tra loro e misura di "efficienza" dei predatori), il modello predatore/preda produce il risultato non lineare di periodi di carestia, che si alternano a periodi di abbondanza. Questo è dovuto ad un effetto di iterazione, ovvero le variabili di flusso e di riflusso, unite, alternano fasi di boom a fasi di crash. In relazione ai mercati finanziari, ciò significa che cause ed effetti non sono sistematicamente collegati, ma possono invece interagire per produrre risultati esagerati.

Feedback Loops. (cicli di retroazione). Un feedback system è quel tipo di sistema, in cui gli output di una iterazione diventano input dell'iterazione successiva. Cicli di Feedback possono amplificare (positive feedback) oppure smorzare (negative feedback) un effetto. Un esempio di positive feedback è rappresentato dal "*multiplier effect"*. Nel mercato dei capitali, un esempio di *Feedback Loops* è rappresentato dalla pratica degli investitori "*momentum*", i quali utilizzano il cambiamento del prezzo dei titoli come un segnale di inzio di operazioni buy/sell, permettendo un rafforzamento del comportamento.

1.5 LA TEORIA E' CONFORME ALLA REALTA'?

Il sistema che si sta analizzando, *Complex Adaptive System*, è relativamente nuovo e coerente con i progressi fatti in altre scienze e, inoltre, gode di un buon potenziale descrittivo, che tuttavia necessita di essere dimostrato nei test reali, spiegando i fatti.

In questo capitolo, sono state spiegate sia le basi della tradizionale *Capital Market theory*, sia le inconsistenze tra questa teoria e la realtà; ora indaghiamo se la nuova scienza sia in grado o meno di colmare le differenze tra teoria e realtà.

Non-normal distributions. Considerando il mercato dei capitali come *Complex Adaptive System,* dovrebbero essere visibili elevate curtosi ("*fat tails*") nelle distribuzioni dei rendimenti.

In particolare, l'alternarsi tra periodi di stabilità e rapidi cambiamenti, attribuibili ai livelli critici, è una caratteristica di molti sistemi complessi, visibili in natura, inclusa l'attività tettonica della superficie terrestre, gli alveari e le evoluzioni. Quindi, le

distribuzioni di rendimenti osservate, alternate a boom e crash, e gli elevati livelli di attività di trading dovrebbero essere tutti consistenti e prevedibili per i nuovi modelli.

Distribuzioni che approssimano il Random walk. La presenza di trend è visibile anche attraverso l'osservazione dei fenomeni naturali, di conseguenza, non dovrebbe stupire il ripetersi dello stesso fenomeno all'interno del mercato di capitali. I nuovi modelli statistici studiati dovrebbero fornire un elevato contributo all'analisi di trend. Il punto principale è che l'andamento dei prezzi nei mercati, assumendo che il mercato sia un *Complex Adaptive System*, dovrebbe essere simile ad una classica distribuzione *random walk*. Inoltre, il nuovo modello dovrebbe essere il miglior strumento per analizzare e spiegare l'andamento dei rendimenti nel tempo, in presenza di persistenza delle distribuzioni.

Portfolio manager performance. Un *Complex Adaptive System* può offrire una miglior descrizione dei modelli di mercato, ma fornisce poche informazioni riguardo alla prevedibilità. Le scarse *performance* di gestione di portafoglio sono dimostrate sia attraverso il nuovo modello, sia attraverso la teoria dei mercati efficienti. Ciò premesso, è possibile, in regime teorico, che alcuni investitori, quali Warren Buffet e Legg Mason Bill Miller, possano essere "*hard-wired*".

In questo senso, il termine "*hard-wired*" suggerisce che in queste persone vi sia la presenza di processi mentali innati che, uniti alla pratica, conferiscono sistematicamente capacità superiori nella selezione dei titoli.

Artificial models simulate market action. Alcuni ricercatori del *Santa Fe Institute* hanno creato un mercato di borsa artificiale, che imita il comportamento di un mercato reale (Arthur [1995]). Il loro modello crea agenti con multipli "*expectational models*", permettendo loro di scartare regole di basse *performance*, in favore di regole che garantiscono, invece, *performance* migliori.

Gli agenti hanno aspettative eterogenee. Il modello mostra che, quando i partecipanti al mercato utilizzano solo parzialmente le loro capacità di aspettativa, prevalgono gli effetti forniti dalla teoria classica del mercato dei capitali. Viceversa, quando i modelli di aspettativa sono adottati più attivamente, il mercato torna ad assumere le

sembianze di un *Complex Adaptive System* ed esibisce le caratteristiche di un mercato di borsa reale (*trading activity, boom and crash*).

Il modello del *Santa Fe Institute*, non essendo così complesso, apre nuove vie per la comprensione del comportamento dei mercati.

L'approccio inerente al *Complex Adaptive System* può essere utilizzato osservando fenomeni naturali. In realtà, questa comparazione è molto utile per comprendere fenomeni, osservabili nei mercati dei capitali. Un esempio riguarda le osservazioni fatte dallo scienziato informatico Mitch Resnick, in relazione al comportamento degli uccelli in natura:«molte persone credono che gli uccelli giochino ad inseguire il leader: l'uccello di fronte allo stormo guida il gruppo e gli altri lo seguono. Ma in realtà non è così. Infatti, molti stormi non hanno un leader: non c'è un "uccello leader". Piuttosto, lo stormo è un esempio di quello che alcune persone definiscono "self-organization". Ogni uccello nello stormo segue delle semplici regole, reagendo ad azioni compiute da uccelli ad essi vicini. Il modello di "stormo ordinato" deriva dalla semplice iterazione di queste regole. L'uccello in prima posizione non rappresenta l'uccello leader in nessun senso, è una casualità che esso finisca in quella determinata posizione. In pratica, lo stormo di uccelli è organizzato senza un organizzatore ed è coordinato senza un coordinatore»(Arthur[1997]).

Inoltre, questo non rappresenta il costante risultato di una leadership, ma può nascere dalla iterazione dinamica di agenti, che attuano decisioni relativamente semplici. In uno studio del 1993, Gode e Sunder testarono la possibilità di creare un semplice mercato, in cui i *traders* utilizzano regole di decisioni semplici, e non necessariamente realistiche, per avanzare le loro offerte di acquisto e di vendita. Lo studio concluse che quel mercato fosse decisamente efficiente; in altre parole, anche agenti stupidi raggiunsero i risultati perseguiti da agenti intelligenti. Secondo le parole dello studioso: « allocative efficiency of a double auction market derives largely from its structure, independent of traders' motivation, intelligence, or learning. Adam Smith's invisible hand may be more powerful than some may have thought; it can generate aggregate rationality not only from individual rationality but also from individual irrationality».

Questi risultati sono in netto contrasto con il concetto del leader. Molte persone si sentono maggiormente sicure sapendo che i prezzi sono settati da investitori razionali. Tuttavia, è evidente, in modo sempre più crescente, che l'aggregazione di molti investitori è sufficiente per creare un giusto funzionamento di mercato.

Impostando la teoria dei mercati, come *Complex Adaptive System,* si crea un ottimo strumento rispetto ai vecchi modelli, per la spiegazione della realtà (*crash, trading activity*). Il punto cruciale è che, incorporando più assunzioni realistiche, anche se più semplici, si verifica una perdita della chiarezza, ottenuta seguendo i correnti modelli economici.

Questo paradigma sposta gli equilibri, perdendo di vista la determinazione realistica e accettando l'indeterminazione, e sostituisce equazioni con un'unica soluzione per modello di equilibrio con modelli formati da equilibri multipli e soluzioni multiple.

1.6 CONSIDERAZIONI PRATICHE

Anche se i mercati dei capitali hanno molto in comune con altri sistemi naturali, cosa può significare questo nuovo paradigma per investitori e professionisti corporate? In che modo essi dovrebbero cambiare i loro comportamenti e i loro atteggiamenti per seguire la struttura dei *Complex Adaptive System*? Possono i vecchi strumenti essere applicati a nuove realtà? Ecco alcune considerazioni.

Il collegamento tra rischio e rendimento potrebbe non essere così chiaro. La teoria della finanza tradizionale assume una relazione lineare tra rischio e rendimento, con un dibattito aperto su come misurare correttamente il rischio. In un *Complex Adaptive System*, tuttavia, rischio e rendimento potrebbero non essere collegati così semplicemente. Le code nelle distribuzioni empiriche, infatti, sono più "spesse" delle previsioni fatte con modelli in cui è essenziale considerare la gestione del rischio e in cui rendimenti estremi non vengono catturati nemmeno dai più precisi modelli economici (testimone il caso di Long-Term Capital Management).

Quali sono le implicazioni pratiche? Per la maggior parte delle decisioni di investimenti corporate, il CAPM è probabilmente il miglior modello disponibile di stima

dell'*investment risk*. I *managers* devono tener conto del fatto che i prezzi degli *stock* potrebbero essere soggetti a sbalzi di volatilità, problema non evidenziato dalla teoria standard.

Non ascoltare gli agenti, seguire il mercato. La maggior parte dei gestori cerca di allocare il capitale in modo da creare valore per gli azionisti. Tuttavia, di fronte a scelte significative, essi solitamente ripongono fiducia nei consigli dispensati da individui scelti (es. analisti o investment bankers), piuttosto che nel contenuto di affidabili studi empirici di mercato. Il *Complex Adaptive System* dimostra che il mercato è più intelligente di quanto lo può essere un individuo stesso.

La maggior parte degli studi, in economia finanziaria, si sviluppa sui livelli di mercato e, quindi, usufruisce dei benefici ottenuti dall'aggregazione. Gestori che accettano i consigli forniti loro da esperti, evitando le evidenze mostrate dal mercato, hanno buona probabilità di intraprendere decisioni in perdita.

Cercare i punti di rottura. Molti *corporate managers* vedono il mercato dei capitali come un qualcosa di "sospetto". In linea di massima, questo scetticismo è infondato quando si opera diversificando e quando gli errori degli agenti sono indipendenti[20], per cui non si creano effetti domino. Tuttavia, se troppi investitori cercano di imitarsi oppure non partecipano al mercato correttamente, può nascere una fragilità e una sostanziale volatilità. I gestori dovrebbero cercare opzioni estreme quando tutti gli investitori agiscono nel medesimo modo e, in possesso di migliori informazioni, dovrebbero essere in grado di intraprendere azioni in acquisto o in vendita in modo da migliorare il valore dei titoli.

Pensare al principio di causa ed effetto è inutile, se non pericoloso. Le persone cercano di relazionare cause ed effetti e l'attività all'interno del mercato dei capitali non è diversa.

Per esempio, i politici crearono numerose commissioni dopo il crash avvenuto nei mercati, nel 1987, nel tentativo di identificare la "causa". Un approccio non lineare, tuttavia, suggerì che cambiamenti su larga scala possono provenire da *input* su piccola scala. La teoria di un *Complex Adaptive System* sostiene che un disturbo casuale, può

avere un enorme effetto e il fatto di essere propensi ad imporre una soluzione, potrebbe essere un passo avanti.

La Tradizionale analisi di sconto dei cash flow rimane la chiave per valutare.

Questo è vero per tre ragioni:

1. il discounted cash flow (DCF) spiega chiaramente principi basilari: il valore di un asset finanziario oggi è pari al valore scontato dei futuri flussi di cassa.
2. il DCF model rimane un eccellente strumento per evidenziare i problemi negli investimenti
3. non vi è forse miglior modello quantitativo basato sugli DCF per spiegare aspettative cristallizzate nei prezzi degli stock[21]?

1.6 CONCLUSIONI

In un noto *paper* del 1953, Friedman fece notare che la validità delle assunzioni di un modello non sono importanti come la precisione delle loro previsioni[13]. Noi argomentiamo che la standard *Capital Market theory* fornisce buone previsioni in molte parti, ma ci sono alcune importanti eccezioni. Per esempio, il cambiamento nell'*asset price* non è conforme alla distribuzione normale, le prove a sostegno del CAPM sono ambigue e l'attività di trading è maggiore di ciò che viene predetto dalla teoria.

Durante i passati decenni, ricercatori hanno definito alcune delle principali proprietà e caratteristiche del *Complex Adaptive System* sostenendo che esso è presente in natura e che le sue caratteristiche generali sembrano essere un buon simulatore di come il mercato dei capitali opera. Importante è notare che il *Complex Adaptive System* mostra i cambiamenti nella distribuzione dei prezzi non osservabili empiricamente, mostrando anche il motivo per cui per gli investitori è così difficile battere il mercato. Inoltre, le ipotesi che stanno alla base del *Complex Adaptive System* sono talmente

[13] Milton Friedman, Essays in Positive Economics (Chicago: The University of Chicago Press, 1953)

semplici da non richiedere assunzioni restrittive circa razionalità degli investitori o leader di mercato.

Da un punto di vista pratico, *manager* che sottoscrivono la *standard Capital Market theory* e operano assumendo *stock market efficiency* probabilmente non avranno un futuro troppo roseo. Quindi la *Complex Adaptive System* potrebbe fornire una utile alternativa in tema di *risk management* e *investor communication*.

Capitolo 2

COGLIERE LA COMPLESSITA' ATTRAVERSO MODELLI AGENT-BASED

Simulare significa riprodurre un fenomeno reale osservabile in natura, mediante lo sviluppo di modelli matematici e logici, atti a riprodurre le caratteristiche di un sistema, per acquisire informazioni utili sul manifestarsi di particolari eventi.

La simulazione tradizionale ha sempre poggiato su due grandi pilastri, quali la simulazione analitica, modelli basati su equazioni risolvibili analiticamente, e la simulazione numerica, modelli basati su equazioni non risolvibili analiticamente ma solo numericamente.

La teoria della complessità, da non confondere con la teoria del caos, studia i fondamenti di sistemi complessi per cui vi è una difficile previsione di andamenti futuri del fenomeno oggetto di studio.

L'interazione tra agenti, che possono essere simulati oppure reali, provocano fenomeni difficilmente spiegabili con regole e teoremi matematici certi e dimostrabili. La causa è legata alla continua mutazione di regole e azioni che modificano sia l'agire dell'agente

stesso sia l'ambiente esterno. Sulla base di questa affermazione che negli anni si è sviluppata la necessità di studiare ambienti per l'osservazione di fenomeni sociali.

Il simulatore programmato, oggetto di questa tesi, fonda le proprie radici teoriche nell'approccio Agent-Based. Ciò che si è voluto osservare è l'analisi di effetti aggregati che differenti comportamenti e strategie di investimento possono risultare in un ambiente virtuale. Una delle caratteristiche principali del lavoro è la condizione che alcuni agenti programmati fondano le proprie strategie su dati reali. Questo aspetto interessante consente la creazione di un collegamento diretto tra ambiente virtuale e ambiente reale. In pratica, si è cercato di ricreare un mercato simulato che generi andamenti di prezzo riscontrabili nella realtà.

In questo capitolo saranno presentate le basi teoriche necessarie a comprendere la simulazione ad agenti, le motivazioni che hanno spinto la scienza a sviluppare questo nuovo paradigma di studio, i campi di utilizzo, i benefici, gli agenti e la loro strutturazione.

Il lavoro di ricerca che sarà possibile leggere il questo secondo capitolo si basa su studi condotto da Terna[2002] e da Sakler[2001] a proposito della capacità, da parte degli ABM, di cogliere gli effetti aggregati e complessi risultanti dalle interazioni sociali.

3.1 MODELLI PER APPROSSIMARE LA REALTA'

Per modello si intende un'approssimazione, uno schema teorico che può essere elaborato in qualsiasi disciplina per rappresentare gli elementi fondamentali della realtà studiata. Gli sviluppi che devono essere seguiti da un modello, hanno l'obiettivo di raggiungere la perfetta emulazione del fenomeno originale, per poter consentire una consistenza di risultati riscontrabili nella realtà utilizzando modelli.

La necessità che ha portato alla nascita di modelli è stata l'incapacità da parte dell'uomo di comprendere e spiegare appieno fenomeni studiati. Ciò che viene modellizzato è quella parte di realtà inspiegabile, resa astratta e controllata attraverso un modello. Il paradigma è, in pratica, la semplificazione di un'astrazione. L'efficacia di un modello deriva dal risultato che, attraverso l'uso di esso, si riesce ad ottenere in

termini di comprensione, replica e previsione di un evento futuro. Nel caso in cui un fenomeno studiato fosse comprensibile senza la necessità di una semplificazione, allora il modello sarebbe inutile. Come scrivevano Rosenblueth e Wiener [1945], l'obiettivo della ricerca scientifica è la comprensione e il controllo di una parte dell'universo. Purtroppo nessuna parte dell'universo è così semplice da poter essere compresa senza l'uso dell'astrazione. Il modello ideale è quindi quello che copre l'intero universo, che ne riproduce per intero la complessità e ha una corrispondenza in scala con esso di uno a uno. Se fossimo in grado di ricostruirlo significherebbe che avremmo capito l'universo nel suo complesso; probabilmente un modello di questo tipo non può essere realizzato dalla sola mente umana. Modelli parziali e imperfetti sono quindi gli unici strumenti con cui l'uomo può cercare di capire i fenomeni dell'universo. Come ha scritto George Box [1976], «tutti i modelli sono falsi, ma alcuni sono utili».

Le modalità con cui un modello può essere costruito sono varie; esse possono variare dalla modalità fisica alla modalità letteraria (altamente flessibile ma non computabile), dalla modalità matematico-statistica (le quali conservano il pregio della possibilità di calcolo) alla modalità simulata.

La strada per sviluppare un modello maggiormente accettata e utilizzata è quella matematico-statistica, la quale unisce rigore e possibilità di calcolo, a cui si unisce di contrasto, la necessità di dover apportare un elevato numero di assunzioni e limitazioni. Partendo dalla sperimentazione tradizionale, ne è un esempio in economia l'agente rappresentativo, in cui la base di un modello era fondata sulla correlazione tra articolazione del mondo e razionalità dell'agente standard medio che popola un determinato ambiente economico. Questo comportava un'elevata difficoltà nel realizzare modelli effettivamente accettati.

Questo limite è stato superato grazie all'introduzione nelle scienze della complessità. Fondamento della complessità, in economia, è l'osservanza di un fenomeno nato dal comportamento aggregato di diversi agenti in relazione con l'ambiente esterno, annullando la ricerca classica della rigida relazione tra complicatezza dell'ambiente e complicatezza dell'agente utilizzato.

2.2 IL PARADIGMA DELLA COMPLESSITA' PER LA RICERCA IN ECONOMIA

Come già anticipato e trattato nel capitolo primo a riguardo dell'intendere l'economia come un *Complex Adaptive System*, riprendiamo il concetto per individuare come la complessità entrerà a fare parte del progetto di ricerca scopo di questa tesi. Per comprendere come applicare il paradigma di complessità alle simulazioni che andremo a fare, dovremo prima comprendere a quale tipo di simulazione dovremmo ricorrere.

Creare simulatori, come creare modelli, può essere fatto seguendo molteplici strade. Nella visione tradizionale, simulare può significare semplici esperimenti mentali, quindi congetture di cui si discutono le conseguenze. In una visione matematico-statistica, simulare può significare il ricorrere a modalità di calcolo alla ricerca di modelli matematici sia in presenza di dati matematici, ne sono un esempio le simulazioni Monte Carlo, sia in presenza di equazioni differenziali. Entrambe le modalità di simulazione spiegate, non riusciranno a soddisfare le nostre esigenze di descrivere il fenomeno sociale che guida le scelte di investitori all'interno dei mercati finanziari, oggetto di studio. Innanzitutto, perché effettuare esperimenti che coinvolgono una molteplicità di persone è molto complicato per ragioni di tempo e spazio; in secondo luogo, perché esprimere la varietà e il dinamismo dei comportamenti umani con modelli matematici può generare delle equazioni che non sono analiticamente computabili. Inoltre, nei problemi studiati dalle scienze sociali, la disposizione spaziale degli agenti può essere una variabile determinante nel risultato finale, ma è una caratteristica difficilmente trattabile per mezzo di equazioni.

L'unica strada percorribile risulta essere quella della simulazione ad agenti, che rappresenta una via del tutto innovativa rispetto ai sistemi sopra descritti. Punto di forza dei modelli ad agenti sono la calcolabilità, propria dei modelli matematici e la flessibilità dei modelli letterario-descrittivi. Inoltre, la metodologia ABM rappresenta il sistema più adatto per l'esplorazione della complessità.

2.3 MODELLI DI SIMULAZIONE AD AGENTI

Nei modelli di simulazione ad agenti, indicati come ABM (Agent-Based Models), un sistema è modellato come una collezione di entità, chiamate agenti, capaci di interagire e prendere decisioni. Ogni agente individualmente valuta la propria situazione sulla base di un *set* di regole. Gli agenti possono esprimere una serie di comportamenti appropriati per il sistema che essi stessi rappresentano, per esempio produrre, consumare oppure vendere. Interazioni competitive e ripetute tra gli agenti sono una caratteristica base degli Agent-Based models, le quali, fanno affidamento sulla potenza dei computer per portare alla luce dinamiche comportamentali non osservabili da altri tipi di approccio. Al più semplice dei livelli, un Agent-Based model è formato da un serie di agenti semplici posti in relazione tra loro. Anche in questo caso è possibile osservare complessi modelli di comportamento che forniscono preziose e valutabili informazioni circa le dinamiche del mondo reale che il sistema sta emulando. In aggiunta, gli agenti possono evolvere permettendo così l'emergere di inaspettati comportamenti. Sofisticati ABM possono incorporare inoltre reti neurali, funzioni di evoluzione o altre tecniche di apprendimento per permettere il realizzarsi di comportamenti realistici di adattamento e apprendimento.

La necessità di uno studio che approfondisse le conoscenze sulle dinamiche che regolano i comportamenti tra le parti che compongono un sistema e il sistema stesso, ha dato vita allo sviluppo dei modelli Agent-Based. Le simulazioni ad agenti, a differenza dei modelli tradizionali, rendono possibile lo studio delle reciproche interazioni tra uomo e ambiente perché permettono di rappresentare, tramite algoritmi e variabili, tutte le parti del sistema sociale: gli individui e i loro comportamenti, la struttura del fenomeno nel suo complesso e le sue proprietà.

L'obiettivo principale è ridurre al massimo le semplificazioni e le assunzioni, per evitare di ottenere risultati distorti e irreali; l'oggetto di studio è il fenomeno macro risultante dall'azione e dall'iterazione micro. Il presupposto teorico è che, ogni fenomeno delle scienze sociali non è che il risultato delle azioni e delle interazioni delle persone che vi prendono parte.

La popolarità che sta nascendo intorno agli ABM pone un interrogativo su quando e come questi sistemi debbano effettivamente essere utilizzati. L'inflazione che può nascere nell'utilizzo di questi modelli, causata dalla facilità di implementazione, può causare un uso improprio e una svalutazione verso questo tipo di approccio. La combinazione che pare più evidente, dallo studio di questi modelli è che, la semplicità tecnologica è contrapposta alla complessità dei concetti. Questa inusuale combinazione rappresenta un punto a favore degli Agent-Based models.

2.4 BENEFICI DERIVANTI DALL'UTILIZZO DEI MODELLI DI SIMULAZIONE AD AGENTI

I benefici generati dall'approccio ABM, rispetto agli altri approcci, possono essere racchiusi in tre grandi categorie:

a) ABM cattura i fenomeni emergenti
b) ABM fornisce una naturale descrizione del sistema
c) ABM è un approccio flessibile

Di seguito, per ogni beneficio, sarà fatto un breve approfondimento.

2.4.1 ABM cattura i fenomeni emergenti

I fenomeni emergenti sono il risultato dell'interazione di semplici entità. Per definizione, fenomeni complessi non possono essere ridotti a semplici parti perché l'intero risultato è superiore alla semplice somma delle parti. Come già approfondito nel capitolo primo, un fenomeno complesso non può essere studiato riducendosi all'osservazione delle semplici unità perché si trascurerebbe una parte fondamentale del risultato ottenuto, conseguenza dell'aggregazione. Un fenomeno emergente può avere proprietà che sono completamente scollegate dalle proprietà delle singole parti. Questo fa sì che i fenomeni emergenti siano spesso imprevedibili e contro intuitivi. ABM è per la sua reale natura, il canonico approccio alla modellizzazione di fenomeni

emergenti, i quali emergono non durante la programmazione del modello, bensì durante la sua simulazione.

L'utilizzo della metodologia ABM dovrebbe avvenire quando c'è la possibilità di un potenziale fenomeno emergente. I casi possibili riguardano:

- simulazioni che includono comportamenti individuali, i quali non sono mossi da regole fissate a priori, bensì evolvono con il cambiamento dovuto al modificarsi dell'ambiente esterno. Descrivere le discontinuità di questi comportamenti risulta difficile attraverso equazioni differenziali
- comportamenti individuali sono caratterizzati da memoria a lungo termine, da abitudini, correlazioni temporali non forzate, condizioni di adattamento e apprendimento
- le interazioni tra gli agenti possono generare una concatenazione di effetti. Sarebbe impossibile studiare delle equazioni che considerino i flussi aggregati, per una semplice motivazione legata al fatto che la concatenazione di effetti possono comportare significative distorsioni che possono andare ben oltre qualsiasi previsione.
- la media non è un buon strumento per approssimare la realtà. Le equazioni differenziali tendono ad appianare le fluttuazioni, al contrario degli ABM che considerano qualsiasi situazione, anche le più estreme.

L'ordine del giorno impartito dalla comunità sostenitore degli ABM è quello di sostenere il nuovo approccio di studio dei fenomeni sociali, cercando di spostare l'attenzione dai modelli tradizionali a innovativi sistemi di studio più efficaci. In accordo con Epstein J.M e Axtell R.L [1996]:«[ABM] may change the way we think about explanation in the social sciences. What constitutes an explanation of an observed social phenomenon? Perhaps one day people will interpret the question, 'Can you explain it?' as asking 'Can you grow it?'».

2.4.2 ABM fornisce una naturale descrizione del sistema

ABM è il sistema più naturale per descrivere un sistema composto da *"behavioral entities".* Quando la necessità è quella di descrivere il comportamento dei mercati finanziari, di ingorghi automobilistici, di possibili riscontri da discorsi elettorali, risulta più affidabile contare su strumenti messi a disposizione dalla tecnologia ABM. Per esempio, se si vuole studiare le dinamiche che governano il modo in cui clienti si muovono all'interno di un supermercato, risulta molto più naturale creare un modello ad agenti rispetto allo studio di equazioni differenziali legate alla dinamicità della densità di clienti nel supermercato. Questo perché, le equazioni differenziali sono studiate partendo dai risultati; gli ABM studiano i risultati rigenerando l'intero processo. E' chiaro quindi che, attraverso la seconda metodologia, il fenomeno può essere scomposto in più parti per approfondire lo studio e ottenere risultati più chiari. Inoltre, la relativa facilità di implementazione di questi modelli, permette di ricostruire situazioni simulate partendo dai dati forniti dagli stessi supermercati sulle abitudini dei loro clienti.

Un altro importante studio, che conduce a risultati maggiormente soddisfacenti utilizzando la modalità ABM, riguarda l'analisi dei "*business processes*".aziendali. Il sistema tradizionale di analisi di "*business process*" parte dal risultato aziendale per poi cercare di risalire all'attività compiuta dai singoli blocchi aziendali. Lo studio compiuto attraverso metodologia ABM, invece, consente di implementare le singole attività compiute effettivamente dai singoli agenti per ottenere simulando il risultato finale del processo aziendale. Come è possibile osservare in Figura 1, il punto di inizio dello studio non è quindi il risultato aziendale, bensì le singole attività compiute nei vari passaggi che conducono al risultato finale.

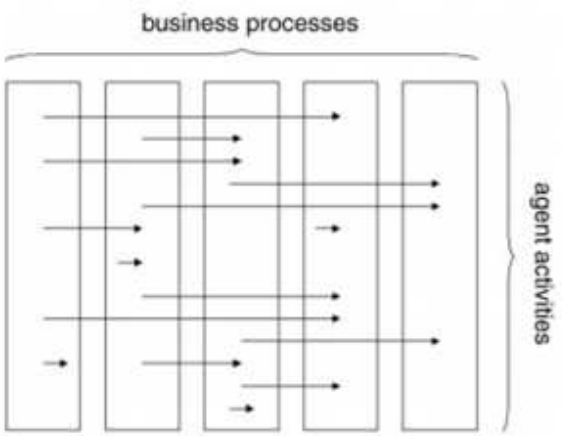


Figura 1 Rappresentazione di un business *process*. Le freccie indicano le attività dei singoli agenti duranti le fasi della produzione

La perfezione del modello, in questo caso, si avrà quando la simulazione delle singole attività compiute dagli agenti conduce ad una coincidenza di risultati effettivi e risultati simulati. Questo quindi permetterebbe di raggiungere migliori risultati nel caso in cui si volesse procedere al miglioramento dei tempi di produzione, attività, allocazione delle risorse ecc…

Riassumendo, si dovrebbero utilizzare metodologie ABM nel caso in cui si presentassero i seguenti scenari:

- I comportamenti degli individui non possono essere chiaramente descritti attraverso semplici assunzioni di tassi di aggregazione
- I comportamenti individuali sono complessi. In principio, qualsiasi cosa poteva essere descritta attraverso equazioni, ma la complessità dei modelli matematici aumentava esponenzialmente all'aumentare della complessità dei comportamenti. Descrivere quindi complessi comportamenti individuali, attraverso equazioni, diventa impossibile.
- Descrivere le singole attività rappresenta il modo più efficace per lo studio di un sistema rispetto all'analisi del risultato dato dell'intero processo.
- Applicazioni stocastiche in cui risulta necessario assumere alcune variabili casuali

2.4.3 La flessibilità degli ABM

La flessibilità garantita dagli ABM può essere osservata in molteplici dimensioni. Per esempio, è semplice in questo tipo di modelli aggiungere più agenti oppure implementare più regole di decisione. Inoltre, ABM fornisce molti strumenti che possono favorire la programmazione di sistemi per modificare la razionalità degli agenti, l'abilità ad apprendere ed evolvere in sintonia con l'ambiente esterno e inserire regole di interazione. Un'altra dimensione della flessibilità è l'abilità di cambiare livelli di descrizione e aggregazione: possono essere inseriti nello stesso modello gruppi di agenti che esprimono caratteristiche completamente diverse, anche dal punto di vista della difficoltà di comportamento. Un altro campo di utilizzo degli ABM è quando un elevato numero di componenti generanti un fenomeno non sono conosciuti, quindi, attraverso tentativi e incremento di complessità nel modello, si cerca di rigenerare il fenomeno.

2.5 GLI AGENTI

L'gente rappresenta il soggetto artificiale fondamento della medotologia di simulazione Agent-Based. La definizione che può essere data al termine agente consiste nel considerarlo un sistema computazionale posto in un certo ambiente dotato di specifiche caratteristiche. Questo concetto è stato ben argomentate da Wooldridge e Jennings[1995] i quali hanno osservato che, gli agenti, per essere considerati tali devono essere dotati di:

- autonomia: caratteristica che consente loro di operare senza la necessità di un intervento umano;
- abilità sociale: qualità che permette di interagire e comunicare con altri agenti
- reattività: caratteristica necessaria che permette agli agenti di reagine in modo tempestivo ai cambiamenti dell'ambiente circostante
- intraprendenza: capacità di intraprendere iniziative guidati da obiettivi interni

Secondo studiosi nel campo dell'intelligenza artificiale, l'agente è un sistema computerizzato che, oltre a possedere le caratteristiche sopra descritte, è concepito usando concetti che sono abitualmente applicati agli esseri umani. Ad esempio, è possibile definire gli agenti usando caratteristiche come la conoscenza, le opinioni, le interazioni, i desideri e i doveri.

Uno studio che abbia per oggetto la programmazione di agenti, deve essere in grado di soddisfare in modo approfondito tutti gli attributi sopra considerati e di dimostrare in che modo sono collegati; ad esempio, dovrà essere specificato il modo in cui l'ambiente influisce sul comportamento dell'agente e come gli obiettivi e le informazioni guidano le sue azioni.

Inoltre, è possibile effettuare una classificazione generale degli agenti usati in una simulazione in base ai loro meccanismi di decisione, alla capacità di adattamento e di apprendimento; se gli agenti sono privi di capacità di apprendimento, sono detti "senza mente"; in caso contrario "con mente". Inoltre, può effettuare una distinzione anche secondo l'ambiente in cui si trovano, che può essere neutrale o strutturato, cioè dotato di regole precise a cui gli agenti devono attenersi. Ne consegue che gli agenti possono essere raggruppati in quattro categorie secondo Terna [2006]:

1. agenti "senza mente" in un ambiente neutrale o non strutturato;
2. agenti "senza mente" in un ambiente strutturato
3. agenti "con mente" in un ambiente neutrale
4. agenti "con mente" in un ambiente strutturato

Nel primo caso, agenti senza mente in un ambiente neutrale, si osserva che dalla simulazione emerge un fenomeno complesso ma non realistico. Questo è stato dimostrato da Terna[2002], in cui si era creato una simulazione di mercato dove vigevano regole di comportamento fissate a priori, quindi senza apprendimento, e la transazione avveniva per un processo casuale di incontro tra gli agenti all'interno dell'ambiente di simulazione. L'esperimento mostra l'emergere di sequenze caotiche di prezzi in un semplice modello interattivo di compratori e venditori programmati con regole minimali. La conclusione che deriva dagli esperimenti è che, la sequenza degli eventi che comporta l'aggiornamento dei prezzi di ciascun consumatore, in un

ambiente composto di soli agenti senza mente operanti in un ambiente non strutturato, causa il generarsi di fenomeni rigidi e meccanici.

Nel secondo caso, agenti senza mente in un ambiente strutturato, si osserva che agenti molto semplici operanti con riferimento ad un meccanicismo di contrattazione telematica qual è quello di borsa "senza grida", producono risultati realistici osservabili anche attraverso serie di bolle e crash di prezzo.

In una simulazione di Terna [2002] è stato programmato un ambiente in cui gli agenti propongono offerte di acquisto e di vendita, con eguale probabilità, le quali sono memorizzate ed ordinate all'interno di un *book.* Le transazioni avvengono in sequenza ad opera del *book,* quando è possibile abbinare due contratti come avviene nel mondo reale. Non si utilizzano quindi equazioni per determinare l'equilibrio tra domanda ed offerta in un dato intervallo di tempo.

Da questi esperimenti si è osservato che i fenomeni quali bolle e crash di prezzo avvengono quando un lato del *book* è molto più corto dell'altro, per il prevalere di offerte di acquisto su vendite. L'aspetto interessante è che fenomeni di andamento reale del mercato, sono osservabili da semplici agenti che senza alcun logica operano in un ambiente simulato. Tale conclusione non era prevedibile all'inizio dei lavori; si pensava infatti, che gli andamenti osservati fossero il frutto di interazioni di agenti "intelligenti", non di semplici agenti privi di intelligenza.

Nel terzo caso, agenti con mente operanti in un ambiente non strutturato, si è osservato come agenti economici siano in grado di apprendere ed agire in maniera coerente, osservando le conseguenze che derivano da passate decisioni. Per un osservatore esterno, sembra che gli agenti operino seguendo obiettivi e strategie interne; tale effetto è in realtà generato dal modificarsi di regole di decisioni degli agenti, a conseguenza delle capacità di adattamento all'ambiente esterno.

Nel quarto ed ultimo caso, agenti con mente in un ambiente strutturato, si osserva l'emergere di fenomeni complessi. Novità riscontrabili, rispetto ai fenomeni complessi presentati nel secondo caso, sono generati dalla capacità di apprendimento e adattamento i quali determinano risultati importanti anche a livello individuale. Risultati ottenuti da Terna [2002] presentano strategie di azione da parte degli agenti

non ovvie, capaci di ottenere risultati positivi anche nel caso di bolle di prezzo e crash continue.

2.6 LA STRUTTURAZIONE DEI MODELLI

In questo paragrafo è descritto, in maniera schematica, una modalità secondo cui idee riguardanti simulazioni ad agenti nelle scienze sociali possono essere trasformate in codici programmabili ed eseguibili. Lo schema proposto da Terna[2000], che prende il nome di ERA (Environment-Rules-Agents) riportato in Figura 2, è quello di gestire quattro diversi strati nella costruzione del modello ad agenti.

I. Un primo strato rappresenta l'ambiente in cui gli agenti sono chiamati ad interagire. Come dettato dal protocollo Swarm (vedere Capitolo dedicato a Slapp), questo compito è eseguito dalla classe *ModelSwarm,* la quale si occupa della creazione degli agenti, della gestione delle azioni, gestione del tempo, strutturazione delle liste ecc..
II. Un secondo strato è rappresentato dagli agenti, i quali possiedono le caratteristiche più svariate, dotati o meno di funzioni di apprendimento, proprietà di ereditarietà, gestione di metodi e dati.
III. Il terzo strato è costituito dai gestori di regole (classi dette RuleMaster); gli agenti interrogano il gestore di regole, il quale, comunica i dati necessari e le indicazioni di azione.
IV. Il quarto strato tratta la costruzione delle regole. Nello stesso modo in cui gli agenti interrogano i RuleMaster, quest'ultimi interrogano i generatori di regole (RuleMaker) per modificare la propria linea di Azione.

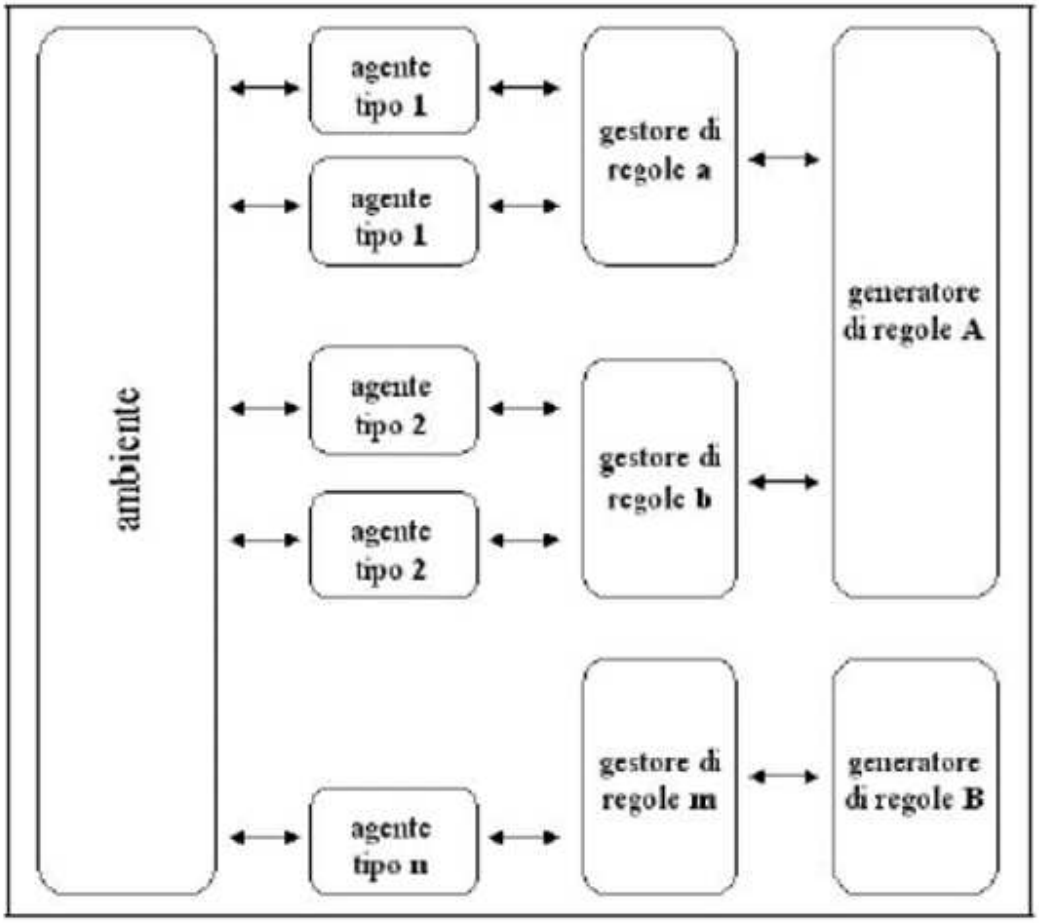


Figura 2 Schema ERA, *Environment-Rules-Agents*

Come sarà possibile leggere approfonditamente nel capitolo dedicato al protocollo di programmazione Slapp, che prende spunto da Swarm e da cui a mia volta io ho preso spunto per programmare il simulatore, si noterà che, la teoria negli anni ha fatto ulteriori passi avanti rispetto alla situazione appena presentata. Un esempio è l'addizionale stratificazione presentata dalla classe Observer che controlla la classe ModelSwarm.

La motivazione che ha spinto i creato del modello sopra citato a dettare delle precise regole di programmazione e strutturazione, trova risposta nel tentativo di rendere rigorosamente ordinato il codice informatico di una simulazione. Se il programmatore segue le direttevi dettate da questo schema, si otterrà una gestione del programma molto semplice, sia nel caso si voglia ampliare la programmazione,sia nel caso si voglia apportare modifiche sia nel caso si necessiti di un debug informatico. La spiegazione di

questo concetto non solo sarà visibile nel capitolo dedicato a Slapp, bensì sarà testato e leggibile nel manuale utente presente al fondo della tesi, in cui si vedrà il percorso da me fatto nella programmazione del simulatore.

Capitolo 3

CRITICHE VERSO LA SIMULAZIONE ABM

Dopo aver esaltato le qualità e le possibilità messe a disposizione della ricerca dai modelli ABM, è doveroso aprire una piccola parentesi relativa alle critiche e ai dubbi, mossi contro questa nuova metodologia.

LeBaron [2008] considera uno dei motivi per cui la teoria *agent-based* gode di scarsa popolarità, soprattutto negli ambienti *mainstream,* la mancanza di una «pietra miliare». Ciò di cui l'ABM necessiterebbe per poter entrare a far parte delle grandi scienze è un la creazione di modello "eccellente". Questo ipotetico modello, oltre ad essere empiricamente dimostrato e dimostrabile, dovrebbe conservare qualità di semplicità per poter essere compreso da coloro i quali non trattano con familiarità strumenti informatici.

Altro aspetto posto in risalto da LeBaron per cercare di trovare una motivazione allo scarso successo, goduto dagli ABM, riguarda l'elevata complicatezza dei modelli implementati. Egli sostiene che, per ottenere un buon risultato contro le critiche, bisogna cercare di pensare a modelli efficaci, che possano essere facilmente spiegati ad un gran numero di scienziati, provenienti da ambiti e campi metodologici differenti.

Inoltre, la facilità con cui un modello è costruito garantisce una miglior verificabilità dei risultati. Le elevate possibilità che modelli *agent-based* forniscono determinano un'elevata discrezionalità di ricerca, creando modelli altamente arbitrari, poco comprensibili e facilmente criticabili. La possibilità di operare con un calcolatore non deve forzatamente concedere spazio all'inserimento di un elevato numero di particolari, che risulterebbero difficilmente controllabili e comprensibili.

In Terna *et al.* [2006] Richiardi e Leombruni illustrano cinque principali punti avanzati a sfavore della metodologia *agent based,* criticandone i contenuti e cercando di dare una spiegazione motivata e convincente a sostegno dell'idea che esiste una stretta analogia tra modelli di simulazione e modelli tradizionali. Di seguito ne riassumerò i contenuti.

1. Le simulazioni non producono teoria.

Questa è la prima critica mossa, secondo gli autori, da un ipotetico economista scettico verso gli *agent-based models* e sostenitore della metodologia standard di ricerca. I due autori, per contro, espongono due concezioni del fare teoria. La prima riguarda la metodologia classica, quindi l'approccio che punta ad individuare una legge universale, in grado di spiegare un fenomeno osservato. La seconda riguarda la concezione alternativa, che individua la capacità di fare teoria nella possibilità di rilevare le cause di un evento. Pertanto, è sufficiente dimostrare che conoscendo la causa è possibile replicare i processi che hanno generato l'evento. Come è già stato ben argomentato nel capitolo 2, i modelli ABM non solo sono capaci di riprodurre un evento, bensì permettono, ricreando l'evento stesso, di poter approfondire la teoria, attraverso lo studio del codice che lo ha generato e di dimostrare se le reali cause sono state effettivamente trovate.

2. Le simulazioni non sono matematica.

La critica rivolta in questo punto è soprattutto legata al non rigore formale, che caratterizza la scienza esatta. Inoltre, per ben comprendere la motivazione per cui le simulazioni sono considerate una matematica minore, cito una celebre frase di Ostrom

[1988] «la simulazione non è né buona né cattiva matematica,semplicemente non è matematica». A tal proposito, gli autori definiscono le simulazioni in maniera contraria; come già citato all'interno del capitolo secondo, lo strumento che le simulazioni mettono a disposizione non solo permette di produrre calcoli computazionali, bensì permette una capacità espressiva superiore. La creazione di agenti, unita ad un elevato grado di flessibilità, ci permette di caratterizzarne diverse tipologie, pur mantenendo una certa trattabilità computazionale. Sarà poi compito del ricercatore evitare di forzare la complicatezza del modello e dell'agente, utile solamente all'incomprensione dei fenomeni generati e allo svilupparsi di critiche.

3. Impossibilità di giungere a risultati generali.

La terza critica è relativa agli output derivanti da simulazione, che sembrerebbero realizzazioni puntuali che dipendono dalle condizioni iniziali impostate, quali parametri e funzioni programmate. L'impossibilità di giungere a risultati generali e leggi universali (come la visione classica) è legata a motivi di arbitrarietà e vincolo alle decisioni del ricercatore. La replica degli autori riguarda il fatto che, se più cicli di simulazione non possono essere rappresentativi di tutti i comportamenti del sistema, lo stesso argomento può essere applicato ai dati reali, soprattutto perchè i dati reali sono solamente una realizzazione dell'evento studiato. Inoltre, la critica mossa su questo piano perde di consistenza con l'avvento di sistemi informatici sempre più potenti, che permettono un numero talmente elevato di simulazioni da poter fornire numerosi scenari, capaci di assicurare un comportamento coerente al modello con formulazioni di leggi generali.

4. Le simulazioni non sono stimabili.

Argomento che riguarda l'eccessivo numero di parametri, che rendono i modelli di simulazione non-identificati o sotto-identificati. La critica, qui enunciata, rappresenta uno dei più difficili problemi che devono essere affrontati dalla ABM. La motivazione è legata al fatto che questo problema deriva, in verità, da uno dei grandi pregi di questa nuova metodologia, cioè la flessibilità e la libertà di implementazione. L'elevato

numero di parametri programmati, come già detto più volte, aumenta la probabilità di incomprensione del fenomeno. Un'altra problematica contro cui la nuova metodologia si deve scontrare è la necessità con cui, per ricreare fenomeni reali, i modelli devono forzatamente essere resi complessi e ricchi di parametri.

5. Le simulazioni non offrono standard metodologici condivisi.

Nella modellistica tradizionale esistono protocolli consolidati che garantiscono una certa chiarezza al lavoro di ricercatore; in questo modo è semplice comprendere come i risultati siano stati ottenuti e come sia possibile replicarli. Spesso, nuove ricerche basate su modalità standard partono da progetti già esistenti, i quali vengono estesi modificandone le assunzioni. Nell'*agent based simulation* non vi è uno standard metodologico condiviso e, spesso, i nuovi modelli divergono nettamente dalla letteratura esistente. Inoltre, la scarsa trasparenza dovuta a dettagli, non resi noti in fase di implementazione, rende impossibile la comprensione. Per di più, lo spazio dedicato alla spiegazione di modelli ABM è simile agli standard utilizzati per altre discipline, i quali risultano insufficienti alla comprensione completa del modello di simulazione ad agenti. Queste condizioni rendono, quindi, difficoltoso un percorso di ricerca coerente e condiviso da tutti. Pertanto, per disciplinare il più possibile la ricerca, che si basa sugli agenti, è necessario un protocollo standard per la costruzione e la descrizione dei modelli.

A questo proposito, gli autori riassumono le misure che permettono sistematicità alla ricerca *agent based*:

I. **Collegamento con la letteratura**: fondamentale per dare allo studio un collegamento letterario sugli argomenti trattati. Inoltre, consente di evitare fenomeni di autoreferenzialità, affrontando un fenomeno già trattato
II. **Struttura del modello**: riguarda le informazioni sulla costruzione del modello, che devono sempre essere portate a conoscenza del lettore per fornirne la comprensione e la replicabilità

III. **Analisi del modello**: i dati artificiali e le relazione tra le variabili devono essere analizzate. In secondo luogo si passa al rapporto tra dati reali e risultati di simulazione
IV. **Replicabilità**:per permettere la ricreazione del modello, l'autore oltre ad osservare i punti precedentemente esposti, deve mostrare particolare attenzione nella pubblicazione di codici, librerie e linguaggi.

In questo capitolo si è voluto, quindi, mettere in evidenza le critiche che impediscono alla metodologia di ricerca ABM di ottenere pari dignità nei confronti della metodologia standard. Attraverso l'individuazione di alcuni semplici accorgimenti, quali rigore metodologico e massima trasparenza, si riuscirebbe a sfruttare l'elevata flessibilità che il nuovo metodo mette a disposizione del ricercatore, evitando di dare spazio alle critiche elencate in precedenza.

Il lavoro che sarà presentato in questa tesi cerca in tutti i modi di replicare lo standard metodologico, che ogni ricercatore ABM dovrebbe seguire, cercando di fornire al lettore tutti i dati e i sistemi, che permettano di replicare e testare le conclusioni a cui sono giunto.

Capitolo 4

SLAPP

Slapp, acrostico di *Swarm-Like Agent Protocol in Python*, è un'applicazione semplificata del protocollo Swarm[14], che utilizza il linguaggio Python come struttura orientata agli oggetti.

Le motivazioni che hanno reso necessario lo studio di un percorso per la programmazione ad agenti sono ricollegabili principalmente ad alcuni aspetti:

a) applicazione in maniera semplificata del protocollo Swarm
b) sfruttamento di tutte le potenzialità di Phyton (programma *open-source*)
c) costruzione di modelli trasparenti e di facile comprensione ai non ideatori
d) insegnamento, fornendo rigorosità ed organizzazione alla stesura di programmi di simulazione
e) creazione di un nuovo modo di concepire l'interazione tra agenti (AESOP, *Agents and Emergencies for Simulating Organizations in Python*)

[14] Swarm è un protocollo di programmazione per la simulazione ad agenti, promosso ed ideato nel 1994 da Chris Langton & Swarm development team of Santa Fe Institute (per approfondimenti vedere capitolo "Swarm").

Per comprendere le potenzialità di Slapp, avvicinandosi alla programmazione ad agenti, è stato scritto un *tutorial* (ispirato a quello originale di Swarm) con l'obiettivo di dotare l'interessato di tutte quelle conoscenze di base, che una volta acquisite lo renderanno capace di ideare programmi completi.
Cercherò di sviluppare e spiegare il *tutorial* per permettere al lettore di capire cosa si intenda per Slapp e per protocollo di programmazione.
Per ragioni di chiarezza, anticipo che i punti da sviluppare sarebbero otto, così come nella versione del *tutorial* originale. In questo capitolo, tuttavia, troverete la spiegazione dei soli primi sei, necessari per la comprensione della parte di simulazione di mercato ed applicazione della strategia di copertura, oggetto del mio lavoro.

4.1 plainPogrammingBug

Questo nome definisce il programma 1 del *tutorial*.
plainPogrammingBug è una struttura molto semplice in cui viene creato un "*bug*[15]" generico, privo di identità. A tale insetto è assegnata una modalità di spostamento casuale sull'asse X e sull'asse Y. Per visualizzare il risultato del movimento e dare avvio alla simulazione, basterà selezionare il comando "*run module*" dalla barra degli strumenti, sotto la voce "*run*". In questo modo si aprirà una nuova pagina con la stampa degli spostamenti del nostro insetto generico.
Attraverso "*SimpleBug*" sono definite le caratteristiche dell'ambiente in cui l'insetto è libero di muoversi, la posizione iniziale in cui esso si trova e l'azione che esso potrà compiere.
Il mondo all'interno del quale il "*bug*" si muoverà non è strutturato, ovvero non è creato in una funzione o in una classe esterna, bensì è definito unitamente alle caratteristiche dell'insetto.
Un'altra funzione programmata è quella legata al movimento. Attraverso "*randomMove*" si descrivono le coordinate del "*bug*" (xPos e yPos), le quali seguono

[15] Il nome "*bug*" è la traduzione inglese di insetto. Solitamente nel linguaggio informatico tale nome è associato alla presenza di un virus all'interno del sistema. Nel nostro caso esso è un nome puramente di fantasia senza alcun significato specifico.

uno spostamento casuale ("*random walk*"). A quest'ultimo è applicata la funzione modulo dei massimi valori di X e di Y, che permettono uno spostamento casuale all'interno di un mondo a forma circolare (toroidale).

```
#start 1 plainProgrammingBug.py
import random

def SimpleBug():

    # the environment
    worldXSize = 80
    worldYSize = 80

    # the bug
    xPos = 40
    yPos = 40

    # the action
    for i in range(10):
        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()

"""
you can eliminate the randomMove() function substituting
        xPos += randomMove()
        yPos += randomMove()
with
        xPos += random.randint(-1, 1)
        yPos += random.randint(-1, 1)

but the use of function allows us to use a self-explanatory name
"""
```

Figura 1 Plain Programming Bug, tratto da *tutorial* 1 Slapp

La prima istruzione del programma, detta "*import random*", consente di importare una libreria interna a Python, che in seguito viene richiamata attraverso il comando di movimento.

"Import random" restituisce un valore compreso tra 0 e 1 casuale. Tale valore potrà poi essere associato ad altre forme di casualità, come per esempio l'istruzione "*random.randint (-1, 1)*", che fornisce un numero intero compreso tra -1 ed 1 inclusi.
Per far sì che la simulazione possa avvenire è stata richiamata la funzione "*SimpleBug*".
Per quanto riguarda la funzione "*randomMove*" non è necessario richiamarla, poiché già utilizzata all'interno del "*ciclo for*".
Il "*ciclo for*", per una costante compresa tra 0 e il numero specificato all'interno del "*range*" con passo 1, ripete le istruzioni sottostanti in corrispondenza dell'*identazione*[16] esatta e rappresenta l'esecuzione del tempo.
Come è possibile notare da Figura 1, l'identazione in Python è fondamentale per un corretto funzionamento del programma.
Nel codice di questa prima parte non vi è alcun riferimento alla programmazione orientata ad oggetti e l'esecuzione del tempo è permessa attraverso l'uso di un ciclo "*for*".

4.2 basicObjectProgrammingBug

Per avere un riferimento alla programmazione orientata agli oggetti[17], dobbiamo proseguire la nostra analisi al *tutorial* numero 2.
Il codice del *tutorial 1* (riferito al bug) è stato incapsulato all'interno dell'oggetto "aBug", contenente la Classe Bug con le differenti funzioni e variabili interne, definite dal programmatore.
La figura 2, che riporta le istruzioni scritte in Python, mostra come sia relativamente semplice definire una classe.

[16] L'identazione è l'inserimento di una certa quantità di spazio vuoto all'inizio di una riga di testo. Essa viene utilizzata nella scrittura del codice sorgente allo scopo di aumentare la leggibilità, soprattutto nel contesto dei linguaggi strutturati: ogni riga viene identata di un certo numero di spazi che dipendono dalla sua posizione all'interno della struttura logica del programma.
[17] Per approfondimenti riguardanti la programmazione orientata agli oggetti vedere capitolo "*Programmazione orientata agli oggetti*".

Dopo aver definito la classe è necessario inizializzarla attraverso la procedura "__init__", la quale specifica tutte le variabili che, una volta creato l'oggetto (incapsulando la classe), saranno utilizzate all'interno della classe stessa.

Per quanto riguarda l'inizializzazione di questa classe gli argomenti sono cinque. In questo caso, tuttavia, due argomenti (*worldXSize* e *worlYsize*) sono già stati specificati nell' "__init__"; pertanto, nella creazione dell'oggetto "*aBug*" forniremo in Bug solamente tre parametri.

Nel caso in cui scrivessimo un numero di parametri diverso da tre il programma segnalerebbe un errore per incongruenza.

Tutto ciò che viene definito con "*def*" rappresenta un metodo.

In ogni metodo, il primo argomento definito si riferisce all'istanza[18] corrente della classe e, per convenzione, questo primo argomento viene sempre chiamato "*self*"[19].

Nel caso di "*__init__*", "*self*" indica l'oggetto che è appena stato creato.

Trattando i metodi, l'unica novità in questo breve programma è rappresentato da "*def reportPosition*", il quale richiamato nel ciclo "*for*" renderà visibile su interfaccia l'identità del "*bug*" e la sua relativa posizione sull'asse X e sull'asse Y.

Per rendere possibile un ciclo di simulazione è nuovamente utilizzata l'istruzione "*for*"; tuttavia, essa non è la sola che può essere impiegata. Il medesimo obiettivo può essere raggiunto con l'istruzione "*while*".

Il ciclo di simulazione richiama l'oggetto "*aBug*", costituito con parametri inseriti arbitrariamente dal programmatore, utilizzando la classe "*Bug*" ed unendo le varie funzioni desiderate.

Nel caso in cui si desiderasse creare un secondo oggetto, utilizzando gli stessi metodi della classe "*aBug*", sarebbe sufficiente nominarlo in modo diverso (es: *bBug*), assegnargli dei parametri diversi ed inserirlo nel ciclo "*for*" (es: *bBug.randomWalk()*) per vederlo in simulazione.

[18] Per istanza non si intende una richiesta, una domanda, un appello o una sollecitazione, bensì si riferisce a caso,esempio o variabile di istanza. Questo differente significato, che dovremo forzatamente attribuire alla parola istanza, è dovuto ad un'errata traduzione dall'inglese del termine "*instance*".

[19] Per approfondimenti sul linguaggio di programmazione fare riferimento all'appendice di questa tesi.

```
# start 2 basicObjectProgrammingBug.py
import random

class Bug:
    def __init__(self, number, xPos, yPos, worldXSize = 80, worldYSize =
        # 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)


aBug = Bug(2, 60, 40)
bBug = Bug(6, 30, 30)

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

Figura 2 basicObjectProgrammingBug, tratto da *tutorial* 2 Slapp ¶

Per concludere, in questo secondo file si è voluto dimostrare che, con la programmazione orientata agli oggetti, è possibile creare ed usare un unico agente come istanza di una classe; in questo modo è molto semplice aggiungere altri agenti come istanza della stessa classe.

4.3 basicObjectProgrammingManyBugs

Nel programma 3 del *tutorial* si continua a sviluppare la programmazione ad oggetti introducendo due elementi significativi:

1. la possibilità di inserire alcune variabili fondamentali per la simulazione; che consentono all'utente di interagire direttamente con il programma
2. la capacità di gestione di più "*bug*"

Il primo blocco di programma è stato ereditato dal *tutorial* numero 2.

Le sostanziali differenze sono visibili nel secondo blocco, in cui si leggono gli "*input*" che l'utente inserirà una volta avviata la simulazione, la quale opererà in base ai parametri inseriti.

Per evitare inutili ripetizioni riporterò solamente la parte di codice in cui sono presenti delle novità in termini di programmazione.

Come è possibile notare, due sono i cicli ripetuti. Il primo ciclo elenca semplicemente l'identità dei "*Bug*" e le loro posizioni di partenza; il secondo, invece, richiama l'oggetto "*aBug*", eseguendo i moduli "*randomWalk*" e "*reportPosition*".

```
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),
                            worldXSize, worldYSize)
    bugList[i] = aBug

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

Figura 3 basicObjectiveProgrammingManyBugs

L'operazione che inserisce in una lista tutti i Bug può essere eseguita anche nel modo rappresentato in Figura 3, dimostrando così la versatilità di questo programma.

```
nBugs = input("How many bugs? ")
bugList=[]

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

Figura 4 Modalità di inserimento dati in una lista

L'utilizzo di una lista ha il grande vantaggio di non dover memorizzare e gestire separatamente ogni singola istanza che viene creata, bensì di raccoglierle in un unico contenitore, trattandole in maniera più semplice. Ogni volta che un nuovo insetto viene creato, esso viene inserito nella lista con una posizione dipendente dall'ordine di creazione.

Questa parte del *tutorial* dimostra la possibilità di raggruppare tutti gli agenti in un'unica lista e di interagire con essi come insieme.

4.4 basicObjectProgrammingManyBugs_bugExternal_+_shuffle

Nel programma 4 del *tutorial* il file è suddiviso in due parti:

- file 1 (Bug.py), in cui è presente la classe che verrà richiamata
- file 2 in cui troviamo le informazioni necessarie per avviare la simulazione.

In questo passaggio è applicata una programmazione di tipo modulare, paradigma della programmazione composta da moduli separati, ognuno dei quali svolge delle specifiche funzioni richiamate dal programma principale.

Questo tipo di struttura permette di snellire il lavoro di sviluppo, verifica ed aggiornamento di programmi di grosse dimensioni.

In Figura 5 sono presenti i comandi relativi all'inserimento delle variabili e alla costruzione della lista e le istruzioni che permettono di comunicare con gli insetti, generando la simulazione vera e propria.

Un elemento di novità consiste nell'operazione di rimescolamento nella lista degli insetti.

All'inizio di ogni ciclo, infatti, è presente la funzione "*shuffle*" che, come "*randint*", appartiene alla libreria "*random*". Questo comando consente di modificare l'ordine delle istanze nella lista "*bugList*", aumentando così la casualità della simulazione.

```
# start 4 basicObjectProgrammingManyBugs_bugExternal_+_shuffle
import Bug
import random

nBugs = input("How many bugs? ")
bugList = []
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.Bug(i, random.randint(0,worldXSize-1), \
                           random.randint(0,worldYSize-1), \
                           worldXSize, worldYSize)
    bugList.append(aBug)

for t in range(length):

    random.shuffle(bugList)

    for aBug in bugList:
        aBug.randomWalk()
        aBug.reportPosition()
```

Figura 5 basicObjectProgrammingManyBugs_bugExternal_+_shuffle da *tutorial* 4 Slapp

Per creare l'oggetto, richiamando una classe da un altro file, occorre inserire il nome della cartella all'interno della quale si possono trovare le istruzioni desiderate.

Entrambi i file, per trasmettere informazioni l'un l'altro e per operare, devono essere contenuti all'interno della medesima cartella di lavoro.

In questa parte del tutorial è stata sottolineata la possibilità di poter simulare eventi gestiti da un conta tempo simulato, programmato in modo dinamico attraverso gli "input".

Una volta creata la lista dei "*Bug",* essi agiranno per il numero di cicli inserito in input; così facendo si attuerà uno dei principi fondamentali della programmazione, quale l'unicità della creazione degli agenti a inizio simulazione. Se avessimo scritto il programma diversamente, creando gli oggetti all'interno del ciclo che opera su

"*length*", avremmo ricreato la lista degli "*aBug"* per un numero di volte pari all'input dei cicli.

4.5 objectSwarmModelBugs

Particolare attenzione deve essere rivolta allo studio del *tutorial* numero 5 di Slapp, in cui è stato fatto un notevole saldo dal punto di vista della strutturazione e della difficoltà del programma.

In questa versione è stata creata una particolare classe *"Model"*, la quale gestisce, attraverso i suoi metodi interni, la creazione degli oggetti, le loro azioni e le regole di iterazione.

La classe in questione non appartiene in verità al mondo dei "*Bug*"; essa è piuttosto una classe che incapsula il programma di simulazione. Così facendo, è possibile trasformare il modello teorico di simulazione in un oggetto, con cui è possibile interagire. Ciò avviene tramite messaggi di *input*, che definiscono variabili fondamentali su cui la simulazione opera, i quali saranno inviati al programma da parte di un utente esterno. Prendendo in esempio il tutorial che si sta approfondendo, i messaggi di *input* definiscono le dimensioni dello spazio in cui i "*Bug*" si muovono, il numero stesso dei "*Bug*" e il numero di cicli.

Il programma 5 del *tutorial* è composto da cinque *files*:

- ActionGroup.py
- Bug.py
- ModelSwarm.py
- Start5objectSwarmModelbugs.py
- Tools.py

Per comprendere lo scopo di questo tutorial e la strutturazione dei file sopra elencati, farò di seguito un approfondimento per ciascun di esso.

4.5.1 ActionGroup.py

"ActionGroup" è una classe che invia strutture di messaggi agli oggetti della simulazione.

Questi messaggi, diversi per ogni classe e azione che si andrà a compiere, saranno letti da metodi interni alla classe "ModelSwarm"; per ogni dicitura il modello farà operare in maniera diversa gli agenti.

Per meglio comprendere quanto detto, proviamo ad immaginare che la classe *"ActionGroup"* assegni delle etichette ai *Bug* relative ad azioni per cui saranno chiamati ad operare. Queste etichette possono essere pensate come dei comandi. Una volta che tutte le etichette sono state assegnate, sarà programmata nella classe di gestione "*ModelSwarm*" un piano di esecuzione del programma definito *"schedule"* utilizzando i comandi inviati dalla classe "*ActionGroup*".

La funzione della "*schedule*" è quella di conferire rigorosità ed ordine alle azioni svolte in simulazione.

Le etichette sopra citate sono assegnate ad oggetti, a cui sono collegati dei metodi.

```
#ActionGroup.py
from Tools import *

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

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

    # place into the instances of this class parts of code generated
    # by the agents, in the form

    # exec("instruction 1; instruction 2; ...")
```

Figura 6 ActionGroup tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

4.5.2 Bug.py

Questa parte del *tutorial* non contiene sostanziali modifiche rispetto alle versioni precedenti approfondite in questo capitolo. L'unica differenza è quella relativa ai

metodi collegati ad applicazioni grafiche, che rappresentano le posizioni raggiunte dai "*Bug*". Questi metodi sono utilizzati nelle versioni più avanzate del tutorial di Slapp.

```
# methods for Tk graphic applications
def getXPos(self):
    return self.xPos

def getYPos(self):
    return self.yPos

def setGraphicItem(self, grI):
    self.graphicItem=grI

def getGraphicItem(self):
    return self.graphicItem
```

Figura 7 Bug tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

4.5.3 ModelSwarm.py

Questo file rappresenta il cuore della gestione del *tutorial* 5 ed è uno dei fondamenti della programmazione ad oggetti.

In una tipica simulazione Swarm, è necessario creare un file per la creazione degli oggetti, per la gestione delle loro azioni e per la gestione delle iterazioni. Per la programmazione di questo file, esiste una procedura pressoché identica, indipendente dal modello di simulazione specifico.

Tutti i dettagli di creazione e gestione, che nel *tutorial* precedente erano stati inseriti nel file *"start"*, ora sono incapsulati in un oggetto chiamato "m*odelSwarm*".

Per avere una visione schematizzata dei compiti eseguiti da "*modelSwarm*" vedere Figura 8.

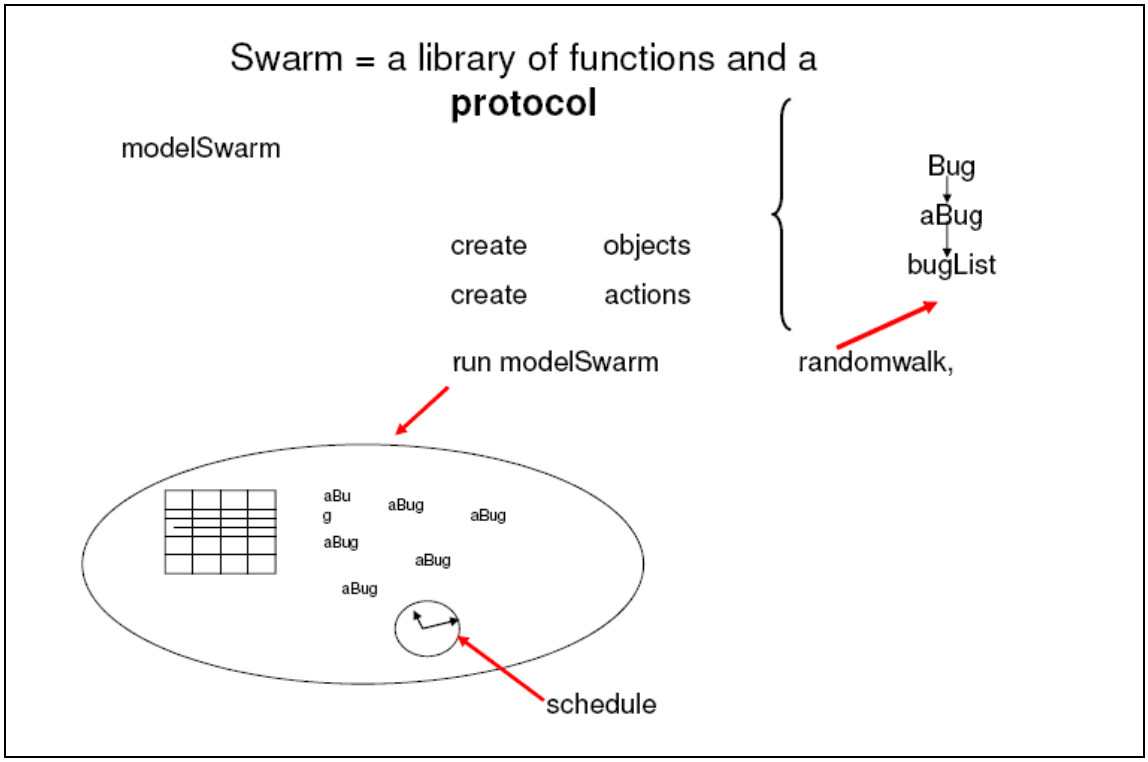


Figura 8 Schema di organizzazione ModelSwarm tratto da *http://eco83.econ.unito.it/terna/slapp/*

Come è possibile immaginare, osservando l'immagine, la programmazione è divisa in tre grandi gruppi:

1) creazione dell'oggetto "*aBug*", che incapsula la classe "*Bug*" (gli oggetti saranno elencati all'interno della lista "*bugList*")
2) creazione delle azioni
3) *"run modelSwarm*", che farà agire i diversi agenti seguendo specifiche condizioni di azione,

elencati nella schedule. Inoltre, è presente la variabile tempo, che sarà incrementata di un'unità ogni qualvolta i Bug compiranno il movimento a loro comandato.

Il codice di programmazione presente in Figura 9 visualizza nella prima parte le istruzioni che consentono di importare i metodi necessari al corretto funzionamento della simulazione programmati nei file "Tools", "Bug", "ActionGroup".

Nell' "__init__" della classe figurano attributi assegnati in input dal file "start".

Le variabili "self.conclude" e "self.t" si riferiscono, rispettivamente, al settaggio a tempo zero della variabile che controllerà la conclusione del ciclo e al settaggio della variabile che controllerà il tempo.

La prima è una variabile *booleana*[20] settata sul falso. Solo nel momento in cui il ciclo sarà compiuto, il metodo definito "*do3*" cambierà il valore della variabile, settandola sul vero.

La seconda variabile riferita al tempo, viene programmata inizialmente pari ad un valore negativo di un'unità per avere il primo ciclo identificato con tempo uguale a 0.

Con il metodo "*buildObjects*" nulla di diverso viene fatto per la costruzione degli oggetti rispetto a quanto già spiegato nella sezione numero 4 del *tutorial* riguardante il file "*start*".

```
#ModelSwarm.py
from Tools import *
from Bug import *
from ActionGroup import *


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
        self.conclude=False
        self.t=-1 # time will start with a 0 value in the first step

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

Figura 9 Parte prima file ModelSwarm tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

Il metodo "*buildActions*" visibile in Figura 10 gestisce sia le azioni compiute dagli insetti, sia la "*schedule*", cioè la procedura di esecuzione dei comandi.

Le azioni sono inserite in un gruppo "actionGroup" al fine di farle eseguire in un ordine specifico nella medesima unità di tempo.

[20] Una variabile booleana, nell'ambito dei linguaggi di programmazione, è un tipo di variabile che può assumere rispettivamente due soli valori o Vero/Falso, True/False, 0/1.

Il metodo definito "*do1*" programma il movimento, il metodo "*do2a*" "*do2b*" rappresentano funzioni che identificano la posizione e il metodo "*do3*" conclude il ciclo.

Il modello non prevede l'utilizzo di cicli "*for*" e fa ricorso a strutture dati passive, contenenti messaggi da inviare. Sono presenti quattro tipi di azioni: "move", "talk all", "talk one" e "end" collegate ai metodi sopra enunciati.

Ogni metodo e ogni azione è richiamata attraverso l'uso di una variabile, ad esempio, "self.actionGroup1.do = do1".

La stringa "*move*", collegata all'azione di movimento, è inviata a tutte le istanze presenti nella lista "*bugList*". Agli insetti è comunicato il messaggio di eseguire il metodo "*randomWalk*" della classe "*Bug*", che prevede la possibilità di muoversi secondo una variabile casuale uniforme compresa tra 0 e 5.

Ogni volta che è applicata la funzione "move" l'orologio virtuale sarà incrementato di un'unità.

Nel caso in cui il valore della variabile "t+1" sia pari a "nCycles", numero di cicli inserito in input dall'utente, la simulazione termina. Questo comando è visualizzato nell' "*if*", il quale, a condizione verificata, inserisce come primo elemento della lista "actionList" l'elemento "end".

Non è possibile osservare questo procedimento, in quanto esso è contenuto nel file "Tools.py", che sarà approfondito più avanti.

Le azioni "*talk all*" e "*talk one*", come già anticipato, sono funzioni che richiedono la stampa della posizione raggiunta dagli insetti. La prima azione "*talk all*" richiede a tutte le istanze di stampare le proprie coordinate e di indicare il numero del ciclo, attraverso il metodo "*report position*" della classe Bug.

La seconda azione, "*talk one*", invia l'ordine di comunicare la propria posizione solamente alla prima istanza della lista "*bugList*".

L'azione "*end*" termina la simulazione assegnando alla variabile "*conclude*", che appartiene all'oggetto "*modelSwarm*", il valore "*True*".

La "*schedule*" ha il compito di dare una strutturazione ed un ordine alle azioni. In questo caso, il programma alterna fasi di movimento a fasi di *report* delle posizioni degli insetti.

```
# actions
def buildActions(self):

    self.actionGroup1 = ActionGroup ("move")
    def do1(address, nCycles, actionList):

        # keep safe the original list
        address.bugListCopy=address.bugList[:]
        # never in the same order (please comment if you want to keep
        # always the same sequence
        random.shuffle(address.bugListCopy)
        # move with a jump, to have to transfer a parameter
        # the format is: collection, method, parameters by name
        # ask each agent, without parameters
        askEachAgentIn(address.bugListCopy,Bug.randomWalk,
                                    jump=random.uniform(0,5))

        self.t+=1 #the clock running
        if self.t+1==nCycles:
             insertASubStepElementInNextStep_firstPosition(actionList,"end")

    self.actionGroup1.do = do1 # do is a variable linking a method

    self.actionGroup2a = ActionGroup ("talk all")
    def do2a(address):

        # ask each agent, without parameters

        print "Time = ", self.t, "ask all agents to report position"
        askEachAgentIn(address.bugList,Bug.reportPosition)

    self.actionGroup2a.do = do2a # do is a variable linking a method

    self.actionGroup2b = ActionGroup ("talk one")
    def do2b(address):
        # ask a single agent, without parameters
        print "Time = ",self.t,"ask first agent to report position"
        askAgent(address.bugList[0],Bug.reportPosition)

    self.actionGroup2b.do = do2b # do is a variable linking a method

    self.actionGroup3 = ActionGroup ("end")
    def do3(address):
        self.conclude=True
    self.actionGroup3.do = do3 # do is a variable linking a method

    # schedule
    self.actionList = [["move"], ["talk all"], \
                       ["move"], ["talk one"], \
                       ["move"], ["talk one"]]
```

Figura 10 Parte seconda file ModelSwarm tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

Il metodo "*run*" conferisce azione alla simulazione. La prima condizione verifica che, fino a quando la variabile *"conclude"* è diversa da *"True"*, il programma estragga il primo elemento dalla lista "actionList" riposizionandolo alla fine di essa ed inserendolo, allo stesso tempo, nella variabile "*step*". L'estrazione ed il riposizionamento appena descritti non possono essere osservarti da Figura 11, poichè "*extractAStepAndRotate*" è un metodo contenuto nel file "*Tools.py*".

In base all'azione estratta, il metodo "*run*" definisce quale istruzione eseguire, richiamando le funzioni attuate in "*buildActions*".

```
# run
def run(self):

    while not self.conclude:
        step=extractAStepAndRotate(self.actionList)

        while len(step)>0:
            subStep=extractASubStep(step)


            if subStep == "move":
                self.actionGroup1.do(self, self.nCycles, self.actionList)
                # self here is the model env.
                # not added automatically
                # being do a variable

            if subStep == "talk all":
                self.actionGroup2a.do(self)

            if subStep == "talk one":
                self.actionGroup2b.do(self)

            if subStep == "end":
                self.actionGroup3.do(self)
```

Figura 11 Parte terza file ModelSwarm tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

4.5.4 Start5objectSwarmModelbugs.py

E' il file principale da cui è possibile attivare la simulazione. Rispetto alla versione precedente alcuni comandi sono stati eliminati e spostati all'interno del file "*ModelSwarm.py*".

In Figura 12 sono presenti alcune istruzioni che consentono all'utente di inserire i valori desiderati relativi al numero di insetti, alle dimensioni del mondo e al numero di cicli di simulazione.

Inoltre, è stata creato l'oggetto modelSwarm, istanziando la classe omonima.

I comandi che seguono richiamano i metodi programmati in "*ModelSwarm*", i quali a loro volta richiamano la costruzione degli oggetti, delle azioni e il conseguente comando "*run*".

```
#start 5 objectSwarmModelBugs.py
from Tools import *
from ModelSwarm import *

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(nBugs, nCycles, worldXSize, worldYSize)

# create objects
modelSwarm.buildObjects()

# create actions
modelSwarm.buildActions()

# run
modelSwarm.run()

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

Figura 12 Start5objectSwarmModelbugs tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

4.5.5 Tools.py

All'interno di questo file sono gestiti gli strumenti utilizzati da classi interne al programma di simulazione. I cicli "*for*" non presenti nel file "*ModelSwarm.py*" sono gestiti nel file visibile in Figura 13 e richiamati a necessità all'interno del programma.

I moduli "*askEachAgentIn*" e "*askAgent*" permettono di individuare le istanze destinatarie di messaggi.

I restanti moduli programmati consentono di gestire la lista delle azioni.

```
import random


# applying a method to a collection of instances
def askEachAgentIn(collection,method,**k):
    """ collection, method, dict. of the parameters (may be empty)"""

    for a in collection:
            method(a,**k)


# applying a method to an instance of a class
def askAgent(agent,method,**k):
    """ agent, method, dict. of the parameters (may be empty)"""
    method(agent,**k)


# extracting a step and rotating a list
def extractAStepAndRotate(aList):
        if len(aList)==0:
            print "Error: action list is empty"
            exit(0)

        aSubList=aList.pop(0)
        if type(aSubList)!=list:
            print "Error: the elements of the action list need to be a list"
            exit(0)
        aList.append(aSubList)

        return aSubList[:] # with [:] we return the elements
                           # of aSubList, not the address

# extracting a subStep
def extractASubStep(aSubList):
        if len(aSubList)>0: return aSubList.pop(0)
        else: return []

# insert an element in next sub-step (first position)
def insertASubStepElementInNextStep_firstPosition(aList,what):
    aList[0].insert(0,what)

# insert an element in nest sub-step (last position)
def insertASubStepElementInNextStep_lastPosition(aList,what):
    aList[0].append(what)
```

Figura 13 Tools tratto da objectSwarmModelBugs, *tutorial* 5 Slapp

4.6 objectSwarmObserverAgents_AESOP_turtleLib

In questa sezione sarà approfondita solamente una parte della sezione 6 del tutorial di Slapp. La parte in questione è rappresentata dalla classe degli "*ObserverSwarm*". Attraverso la creazione di quest'ultima, al programma sarà data un'ulteriore stratificazione dal punto di vista della gestione. L'evoluzione percorsa nel tutorial ci ha permesso di comprendere che, in primo luogo, la gestione era inserita direttamente nello stesso file in cui si programmava la classe dei bug. L'evoluzione a questa

strutturazione è poi stata la programmazione di un file "*Start*" che gestiva la creazione degli oggetti e delle azioni, il quale è poi stato parzialmente incorporato dal successivo sviluppo del file "*Model*" approfondito nel *tutorial 5*.
In questa sezione vedremo che la gestione del programma appartiene alla classe "*ObserverSwarm*", la quale comprenderà alcuni moduli che prima appartenevano al file *Model* e gestirà le operazioni per la rappresentazione grafica dei dati generati dalla simulazione. Inoltre, sarà proprio la classe che descriverò che sarà richiamata dal file "*Start*" per la creazione degli oggetti e gestione delle azioni.
Obiettivo di questa introduzione era di riassumere, in breve, i passi per che hanno portato alla creazione della classe "*ObserverSwarm*".

4.6.1 ObserverSwarm.py

Non potendo scendere nei particolari della programmazione visibile in Figura 14, in quanto necessiterebbe di una spiegazione approfondita delle classi e dei metodi richiamati, porrò maggiore attenzione al programma in generale per così comprendere la funzione complessiva di questa classe. Nella prima parte è visibile la creazione degli "step". Per "step" si intende una frazione di ciclo. All'interno di uno step, tutti i "*Bug*" costruiti ed inseriti nella "*BugList*" vengono richiamati, estratti uno ad uno dalla lista e fatti agire. Nella costruzione degli oggetti è possibile notare i messaggi di "*input"* (solitamente programmati nel file "*Start.py*"), che saranno inseriti da un utente esterno. Concettualmente, questo ci può portare ad immaginare le modalità di programmazione del file, da cui si darà avvio alla simulazione. Esso sarà semplicemente una sequenza di "*import*" con la successiva creazione di un oggetto, che incapsulerà la classe "ObserverSwarm" e i relativi metodi ad essa collegati. Il file "*Start*" sarà, dunque, un semplice interruttore che darà avvio alla programma.

```
class ObserverSwarm:

    # creation step
    def __init__(self, project0):
        global project
        project=project0
        self.v=[]
        if project=='school':
            self.p=Pen(0,0)
            self.p.white(-200,150)
        else: self.p=0

    # create objects
    def buildObjects(self):
        global cycle
        cycle=1
        mySeed = input("random number seed (1 to get it from the clock) ")
        if mySeed == 1:
           random.seed()
        else:
           random.seed(mySeed)
        self.nAgents = input("How many 'bland' agents? ")
        self.worldXSize= input("X Size of the world? ")
        self.worldYSize= input("Y Size of the world? ")
        self.nCycles = input("How many cycles? (0 = exit) ")
        self.conclude=False

        self.modelSwarm = ModelSwarm(self.nAgents,
                             self.worldXSize, self.worldYSize, project)
        self.modelSwarm.buildObjects()
```

Figura 14 Parte prima ObserverAgent tratto da objectSwarmObserverAgents, *tutorial* 6 Slapp

In Figura 15 è possibile osservare il linguaggio di programmazione per la costruzione delle azioni. Questo metodo, così come quello descritto in Figura 14, è omonimo di due metodi presenti nella classe "ModelSwarm". Essi, infatti, sono richiamati rispettivamente all'interno della classe "ObserverAgent". Per quanto riguarda il metodo "*do1*", esso incrementerà di un'unità la variabile global "*cycle*". Una volta che la condizione verificata con "*if*" sarà accertata, all'interno dell'actionList sarà posta la scritta "*end*". Questo appena descritto è un modo elegante per porre fine alla simulazione.

```
# actions
def buildActions(self):

    global project
    print "#### Project", project,"starting."
    print

    observerActions=open("./"+project+"/observerActions.txt")
    self.actionList=observerActions.read().split()
    observerActions.close()

    self.modelSwarm.buildActions()

    # clock, do not remove or move from here; the position in
    # list can be also different
    self.actionGroup1 = ActionGroup (self.actionList[2])
    def do1(address, nCycles, actionList):
        global cycle
        cycle+=1 #the clock running
        print "Time =%2d" % cycle
        if cycle>nCycles:
            insertElementNextPosition(actionList,"end")
    self.actionGroup1.do = do1 # do is a variable linking a method

    self.actionGroup2a = ActionGroup (self.actionList[1])
    self.actionGroup2a.do = do2a # do is a variable linking a method

    self.actionGroup2b = ActionGroup (self.actionList[4])
    self.actionGroup2b.do = do2b # do is a variable linking a method

    # previous steps, if empty, are defined in the specific
    # oAction.py file
    # repeat for new or different steps

    # do not remove or move from here and dot not rename
    self.actionGroup3 = ActionGroup ("end")
    def do3(address):
        self.conclude=True
    self.actionGroup3.do = do3 # do is a variable linking a method
```

Figura 15 Parte seconda ObserverAgent tratto da objectSwarmObserverAgents, *tutorial* 6 Slapp

La programmazione seguente a questo file riguarda il "*run*". La struttura è molto simile a quella visualizzabile in Figura 11, per questa ragione evito di riportare la programmazione e di ripetere concetti già approfonditi.

Capitolo 5

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

La programmazione orientata agli oggetti è una filosofia di programmazione estremamente efficace e vicina al programmatore, il quale è posto davanti ad un elevato numero di scelte sulle tecniche di sviluppo più versatili e vicine alla vita reale. Lo stile di programmazione che si vuole presentare, come si leggerà in seguito, permette una rappresentazione della realtà ricreata da semplici unità informatiche interagenti. La programmazione di funzioni operanti su flussi di dati, sia inseriti dall'utente sia generati dal programma stesso, permettono di riprodurre virtualmente un fenomeno reale. In generale, la metodologia di sviluppo di un programma ad oggetti è di tipo "*top-down",* cioè la risoluzione dei problemi inizia da livelli di astrazione maggiori, per giungere ai problemi più concreti legati all'implementazione. Gli elementi fondamentali di tale paradigma[21] sono gli oggetti, che in un primo momento sono definiti, descrivendone le caratteristiche, poi creati e allocati in

[21] Per paradigma si intende uno stile di programmazione, ossia un insieme di strumenti forniti da un linguaggio di programmazione per la stesura di programmi. Esistono diversi approcci alla programmazione, chiamati appunto paradigmi. Alcuni esempi riguardano il paradigma imperativo, il paradiga funzionale e il paradigma logico.

memoria e, infine, usati in relazione l'uno con l'altro. La base di questa tecnica di programmazione è la struttura, che ci permette di modellare entità astratte.
Nel corso di questo capitolo si illustreranno le caratteristiche principali della programmazione ad oggetti e, brevemente, i diversi ambienti di simulazione su cui è stato programmato il modello. Inoltre, sarà introdotto il significato dei vari strumenti che incontreremo nella programmazione del simulatore, osservabili nel codice (visibile al fondo di questa tesi).

5.1 DEFINIZIONE

La programmazione orientata agli oggetti (OOP, *Object Oriented Programming*) è uno stile di programmazione, che prevede la creazione di oggetti software che interagiscono gli uni con gli altri attraverso lo scambio di informazioni.
Per programma informatico si intende uno strumento che, unito alla potenza dei calcolatori, è capace di ricreare una porzione della realtà. Secondo il paradigma che si sta definendo, la porzione di mondo reale è ricreata programmando una serie di oggetti software, quindi, una serie di operazioni prodotte su dati. Un oggetto software non è, quindi, altro che un'adeguata rappresentazione virtuale, simulata dal calcolatore, di un oggetto nella realtà, sia esso concreto o astratto. Il fenomeno reale osservato, che si vuole studiare, è quindi ricreato grazie alla cooperazione e all'interazione di oggetti.
La programmazione ad oggetti si fonda sull'astrazione di un problema reale, unendo concetti informatici a teorie scientifiche.
Come già anticipato, un oggetto è una singola entità, che combina sia strutture dati sia comportamenti: i dati sono visti come variabili e le procedure per accedere ai dati sono viste come metodi.
Questo tipo di programmazione mette a disposizione del programmatore una serie di strumenti, che consentono di esprimere i problemi in maniera relativamente semplice.
Un programma si configura, quindi, come un insieme di oggetti che, interagendo

tramite messaggi, danno vita a strutture anche molto complesse. Il vantaggio di questo tipo di impostazione è che non vi sono limiti alle tipologie di casi rappresentabili.

5.2 DALLA PROGRAMMAZIONE PROCEDURALE ALLA PROGRAMMAZIONE AD OGGETTI

Python è un linguaggio orientato agli oggetti che permette, però, sia la programmazione tradizionale (procedurale) sia il nuovo paradigma ad oggetti.
Esso è, quindi, collocabile in quella categoria di linguaggi, detti ibridi (altro esempio è il programma C++[22]).
La programmazione tradizionale si è sempre basata sull'utilizzo di strutture dati (come le liste), funzioni e procedure. Questo metodo di sviluppo del software è detto funzionale (o procedurale) perché con esso si organizza un intero programma in moduli, che raccolgono gruppi di funzioni. Ogni funzione accede ad uno o più gruppi di dati.
I metodi di sviluppo funzionale conservano notevoli punti di debolezza:

a) la stretta connessione tra funzioni e dati causa, nel caso in cui si debbano apportare modifiche al software, effetti collaterali su altri moduli con enormi difficoltà di *debug*[23] della applicazione
b) difficoltà di riutilizzo del software. Ogni volta in cui si vuole riciclare una funzione occorre apportare delle modifiche strutturali affinché si adegui alla nuova applicazione.

La programmazione orientata agli oggetti è un modo alternativo di scomposizione di un progetto software: in essa l'unità elementare di scomposizione non è più

[22] C++ è un linguaggio di programmazione orientato agli oggetti, con tipizzazione statica, ovvero di assegnazione di tipi alle variabili. Rappresenta un'estensione del linguaggio C, di cui conserva i punti di forza quali la potenza, flessibilità di gestione dell'interfaccia hardware e software, la possibilità di programmare a basso livello e l'efficienza di C. Punto di forza del linguaggio C++ è appunto la dinamicità data dalla programmazione ad oggetti, che rende tale linguaggio, una piattaforma ideale per l'astrazione dei problemi di alto livello.

[23] Debugging (o semplicemente debug) è una procedura che consiste nell'individuazione della porzione di software affetta da errori (bug).

l'operazione (la procedura) bensì l'oggetto, inteso come modello di un'entità reale (un oggetto nel mondo reale).

Questo approccio porta ad un nuovo modo di concepire il programma: il software è costituito da una serie di entità (gli oggetti) interagenti, ciascuno provvisto di una struttura dati e dell'insieme di operazioni, che l'oggetto è in grado di effettuare su quella struttura. Poiché ciascun oggetto incapsula i propri dati e ne difende l'accesso diretto da parte del mondo esterno, si è certi che cambiamenti del mondo esterno non influenzeranno l'oggetto nè il suo comportamento.

D'altra parte, per utilizzare un oggetto basta conoscere quali dati esso immagazzina e che operazioni esso fornisce per operare su questi dati, senza curarsi dell'effettiva realizzazione interna dell'oggetto stesso.

Questo nuovo concetto di programmazione è, inoltre, molto adatto a contesti in cui si possono definire relazioni di interdipendenza (contenimento, uso, specializzazione) tra i concetti da modellare.

Un ambito che più di altri riesce a sfruttare i vantaggi della programmazione ad oggetti è quello delle interfacce grafiche.

5.3 L'ASTRAZIONE NELLA PROGRAMMAZIONE AD OGGETTI

La programmazione ad oggetti, come già sottolineato, è un modo alternativo di affrontare e scomporre il progetto software. Il suo paradigma è che l'unità elementare di scomposizione non è più la procedura bensì l'oggetto, inteso come modello di entità reale.

Quest'ultima operazione, chiamata *abstraction (*o *data abstraction*), ha un'importanza fondamentale nei linguaggi orientati agli oggetti ed è uno dei metodi fondamentali con il quale gli esseri umani affrontano i sistemi complessi. Esistono varie definizioni del concetto di astrazione. Booch definisce l'astrazione come segue: <<con l'attività di astrazione si individuano le caratteristiche di un oggetto che lo distinguono da tutti gli altri tipi di oggetti fornendo, quindi, per l'oggetto in questione, un ben preciso ambito concettuale, relativo al punto di vista dell'osservatore>>.

L'attività di astrazione implica la facilità di definire oggetti che rappresentino "attori" che possono svolgere lavori, riportare e cambiare il loro stato, e comunicare con gli altri oggetti del sistema.

Un sistema complesso è, così, visto come un insieme di oggetti interagenti (Figura 1), ciascuno dei quali provvisto di una struttura dati e dell'insieme delle operazioni, che l'oggetto è capace di compiere su tale struttura. Un oggetto, inclusa la propria struttura dati, non viene influenzato dai cambiamenti che avvengono al di fuori di esso.

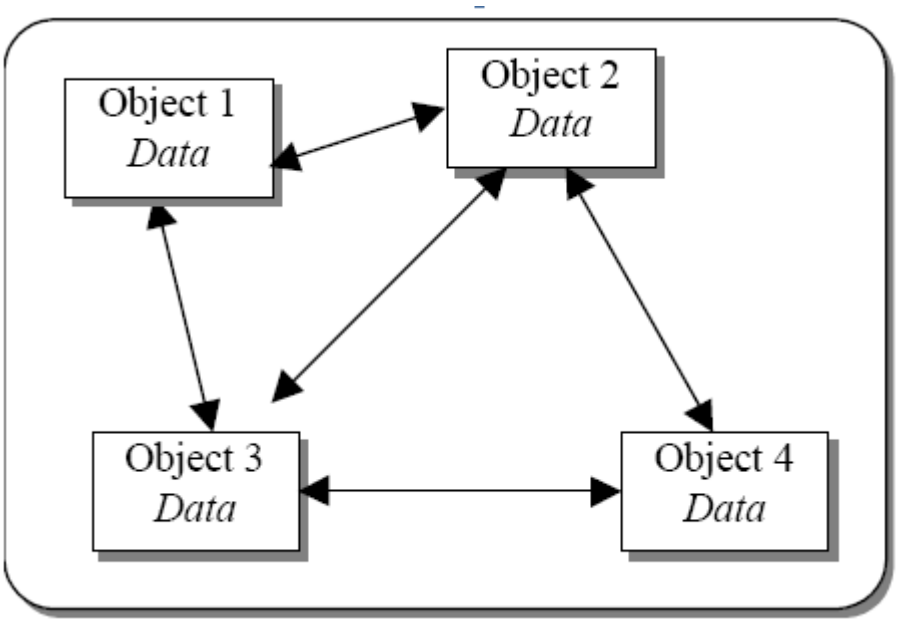


Figura 1 Sistema complesso descritto da un insieme di oggetti interagenti.

5.4 PROPRIETA'

In genere, un linguaggio di programmazione è definito ad oggetti quando permette di implementare i tre meccanismi seguenti:

- incapsulamento
- ereditarietà
- polimorfismo

L'incapsulamento è la proprietà per cui un oggetto contiene (incapsula) al suo interno degli attributi (dati). Lo scopo principale dell'incapsulamento è, appunto, dare accesso ai dati incapsulati solo attraverso i metodi accessibili all'esterno.

Se gestito opportunamente, l'incapsulamento permette di vedere l'oggetto come un *black-box* (scatola nera) di cui, attraverso l'interfaccia, è noto come si comporta e come interagisce con l'esterno, ma non come esegue le funzioni.

L'incapsulamento prevede di raggruppare in un'unica entità (la classe) la definizione delle strutture dati e delle procedure, che operano su di esse.

Le classi definiscono, appunto, gli "oggetti" software dotati di *attributi* (dati) e *metodi* (procedure), che operano sui dati dell'oggetto stesso.

L'ereditarietà permette essenzialmente di definire le classi, a partire da altre già definite. Una classe o una sottoclasse, derivata attraverso l'ereditarietà, mantiene i metodi e gli attributi della classe da cui deriva (classe base o superclasse); inoltre, possono essere aggiunti nuovi attributi e metodi e può essere modificato il comportamento dei metodi ereditati (*overriding*).

Si noti in Figura 2 come, grazie alla proprietà di ereditarietà, le informazioni comuni delle classi "Studente" e "Professore" sono fattorizzate nella classe "Persona", da cui "Studente" e "Professore" le ereditano.

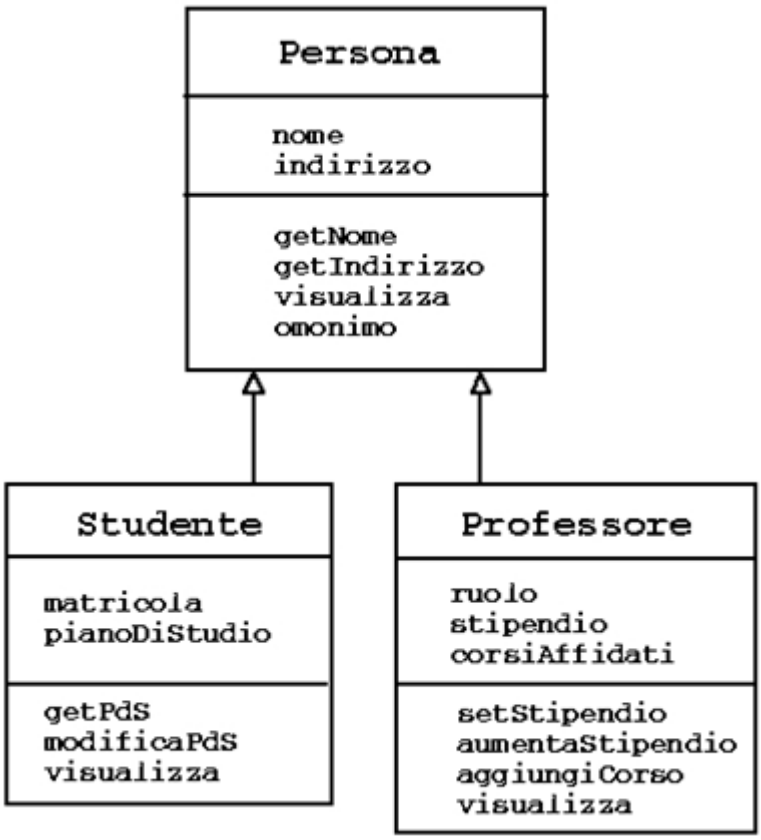


Figura 2 Rappresentazione della proprietà di ereditarietà nella programmazione ad oggetti.

Quando una classe eredita da una superclasse si parla di eredità singola, viceversa di ereditarietà multipla. Alcuni linguaggi (tra gli altri Java[24]) forniscono il supporto esclusivo all'eredità singola. Altri (Phyton, C++) prevedono l'eredità multipla.

L'ereditarietà può essere usata come meccanismo per gestire l'estensione e il riutilizzo del codice e risulta particolarmente vantaggiosa sfruttando le relazioni di eredità, esistenti nei concetti modellati, attraverso il costrutto di classe. Oltre all'evidente riutilizzo del codice della superclasse, l'ereditarietà permette la definizione di codice generico, attraverso il meccanismo del polimorfismo.

Il polimorfismo è una proprietà del codice che permette diversi comportamenti a seconda del contesto di esecuzione. In pratica, ciò avviene quando lo stesso codice può essere utilizzato con istanze di tipi diversi. Nella programmazione ad oggetti, il polimorfismo è legato alle relazioni di eredità tra classi, garantendo delle proprietà in

[24] Java è un linguaggio di programmazione orientato agli oggetti, creato da James Gosling e altri ingegneri di Sun Microsystems.

comune tra gli oggetti. La possibilità che le classi derivate implementino in modo differente i metodi e le proprietà dei propri antenati rende possibile che gli oggetti, appartenenti a delle sottoclassi di una stessa classe, rispondano diversamente alle stesse istruzioni.

5.5 CLASSI

Nella programmazione orientata ad oggetti, una classe è definibile come un unione di metodi per la gestione di variabili, usata come modello per creare oggetti. In informatica, le classi sono utilizzate per l'effettiva programmazione di un fenomeno osservato che si vuole riprodurre virtualmente. Ad esempio, all'interno del simulatore, è stata creata una classe che riportasse in simulazione le caratteristiche di comportamento di differenti agenti studiati nella realtà. Ogni oggetto software, inoltre, è creato richiamando una classe. Il modello "classe" comprende attributi e metodi, che saranno condivisi da tutti gli oggetti (istanze) creati (ovvero *istanziati*).
Una classe può rappresentare una persona, un luogo o una cosa ed è, quindi, l'astrazione di un concetto implementata in un programma informatico. Essa definisce, al proprio interno, lo stato ed il comportamento dell'entità che rappresenta.
Grazie alle relazioni di ereditarietà, nuove classi di oggetti sono derivate da classi esistenti, ereditando le loro caratteristiche ed accorpandole con le proprie[25].
Definire una classe coincide essenzialmente con la definizione dei propri membri:

- attributi: rappresentano quelle proprietà che descrivono le caratteristiche peculiari di un oggetto (ad esempio, riferendoci ad una persona: altezza, peso)
- metodi: identificabili come proprietà, atte a svolgere delle azioni

Dal punto di vista matematico il concetto di classe è definibile come un insieme in modo intensivo, ovvero, sono definite le caratteristiche e non sono elencati gli elementi. In altre parole, sono definiti i metodi ma non sono forniti i dati su cui i metodi operano.

[25] Per comprendere meglio questo concetto, fare riferimento al manuale utente in cui sarà spiegata una particolare classe programmata a partire da un'altra classe.

5.6 OGGETTI

Un oggetto è un'istanza di una classe, ossia, la realizzazione di un dato secondo le specifiche funzioni definite nella classe stessa. Un oggetto è la rappresentazione di una classe all'interno del programma, in cui, oltre al nome della classe, vengono definite le sorgenti dati a cui le variabili, richiamate nei metodi, devono fare riferimento.

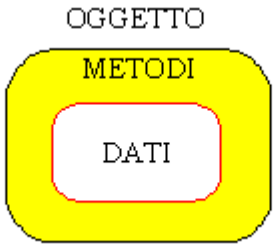


Figura 3 Struttura di un oggetto in Python

Esso contiene gli attributi, definiti nella classe, e il valore degli attributi determina il suo valore (o stato). Secondo il principio, noto come *information hinding*[26], l'accesso agli attributi di un'istanza è permesso solo tramite metodi richiamati su quello stesso oggetto. Facendo riferimento al simulatore, un oggetto creato sulla classe rappresentante gli agenti "*random*" può solamente richiamare metodi interni a questa specifica classe e non metodi appartenenti a classi diverse.

5.7 ELEMENTI DISTINTIVI

La programmazione ad oggetti si riassume nella creazione di variabili, nella loro gestione attraverso particolari metodi e infine nell'iterazione con l'ambiente esterno attraverso la gestione di messaggi. La strutturazione di ogni programma, dal più complesso al più semplice, può essere pensata come una serie di metodi, attributi,

[26] I termini incapsulamento e information hinding (letteralmente "occultamento delle informazioni") vengono spesso utilizzati come sinonimi, anche se, a rigore, esiste una differenza concettuale: l'information hinding è il principio teorico su cui si basa la tecnica di incapsulamento. Una parte del programma può nascondere informazioni incapsulate in un modulo, o in un costrutto di altro tipo, dotato di interfaccia.

classi che circondano un corpo centrale che rappresenta il motore del programma. Qualsiasi oggetto, come già spiegato in precedenza, può inglobare una classe. A questo oggetto può essere applicato qualsiasi metodo, purchè interno alla classe a cui fa riferimento. Quanto appena enunciato rappresenta una delle maggiori peculiarità di questo tipo di programmazione: una volta creato il corpo centrale del programma, esso può essere riutilizzato un numero indefinito di volte per correggere e risolvere i fenomeni osservati in natura che si vogliono simulare.

I linguaggi procedurali implicano l'utilizzo di tipi di dati concepiti per rappresentare unità di memorizzazione della macchina; i linguaggi ad oggetti consentono invece, l'espressione di idee organizzate all'interno del computer in base a entità, gli oggetti appunto, concepiti per rappresentare unità concettuali.

Ogni volta che si crea un nuovo elemento (istanza) da una classe, si crea una unità in grado di compiere operazioni su variabili tramite delle funzioni (metodi). Queste, se definite all'interno della classe originaria, possono essere interpretate correttamente da tutti gli oggetti che da questa derivano. Si può operare sulle istanze inviando messaggi con cui si richiede l'attivazione di un determinato metodo che queste, quindi, riconoscono.

I messaggi interpretabili da ciascuna classe sono associati a funzioni; quando si "comunica" con un oggetto viene chiamata la funzione corrispondente all'esigenza da assolvere.

5.8 UN ESEMPIO DI LINGUAGGIO DI PROGRAMMAZIONE ORIENTATO AGLI OGGETTI: PYTHON

Python è un linguaggio di programmazione dinamico orientato agli oggetti, rilasciato pubblicamente per la prima volta nel 1991 dal suo creatore Guido van Rossum, programmatore olandese attualmente operativo in Google. Per curiosità, Python deve il suo nome alla commedia *Monty Python's Flying Circus,* in onda sulla BBC nel corso degli anni 70.

Python supporta diversi tipi di paradigmi di programmazione, tra i quali, il già citato object-oriented, il paradigma imperativo e il paradigma funzionale. Inoltre, è provvisto di diverse librerie standard che lo rendono adatto a molti impieghi e offre la possibilità di aggiungere librerie scritte sia con Python, per soddisfare particolari esigenze, sia con altri linguaggi di programmazione (esempio C). Inoltre è dotato di un elevato numero di interfacce che permettono di sfruttare, attraverso Python, le potenzialità di altri linguaggi di programmazione. Un esempio è l'interfaccia *rpy*, che consente di gestire e utilizzare gli strumenti di R con Python. Nella simulazione, R[27] è stato utilizzato, attraverso l'interfaccia rpy2, per la rappresentazione grafica delle distribuzioni dei prezzi.

Attualmente, lo sviluppo di Python viene gestito da una organizzazione no-profit e deve gran parte delle sue evoluzioni all'enorme e dinamica comunità internazionale di sviluppatori. L'aspetto che riguarda l'*open source* è una delle caratteristiche di questo linguaggio, perché si è riusciti a creare un linguaggio avanzato, semplice, dinamico e professionale partendo da un principio di libertà di utilizzo, consulto e sviluppo.

5.9 DEFINIZIONE DI CLASSI E OGGETTI IN PYTHON

In Python le classi sono dichiarate con la direttiva "class" con una struttura del tipo visualizzabile in Figura 4.

```
class <nome classe> [(<classe madre>,...)]:
    <elenco dati membro da inizializzare>
    <elenco metodi>
```

Figura 4 Dicitura di una "classe" in Python

I dati membro si esprimono come le normali variabili di Python. Se si devono inizializzare dei dati membro, si procede attraverso una semplice assegnazione, oppure attraverso la definizione al momento dell'utilizzo, esattamente come avviene per le variabili normali.

[27] R è un linguaggio di programmazione *open source* adatto per elaborazioni statistiche e grafiche

Per i metodi è necessario utilizzare la medesima sintassi delle funzioni con alcuni accorgimenti.

Ogni metodo deve avere come primo parametro l'oggetto stesso, infatti ogni volta che viene richiamato il metodo, Python sistema nel primo parametro il riferimento all'oggetto stesso.

Questo permette di accedere ai dati membri appartenenti allo stesso oggetto. Normalmente si usa chiamare questo primo parametro "self "[28].

In Figura 5 è possibile visualizzare la codificazione di una classe in Python.

```
class persona:
    nome = 'francesco'
    cognome = 'lovera'
    indirizzo = 'via_langhe'
    telefono = '313'
    stato_civile = 'celibe'
    print nome,cognome,indirizzo,telefono

    def cambia_indirizzo(self,s):
        self.indirizzo = s
    def cambia_telefono(self,s):
        self.telefono = s
    def cambia_stato_civile(self,s):
        self.stato_civile = s
    def stampa(self):
        print self.nome,self.cognome,self.indirizzo,self.stato_civile
```

Figura 5 Codice di programmazione di una "classe" in Python

Se ora si volesse istanziare un oggetto, è sufficiente richiamare il nome della classe come se fosse una funzione.

Se volessi accedere ai dati membri e ai metodi è necessario utilizzare il "punto".

```
p1 = persona()  # le parentesi sono obbligatorie
p1.nome = 'stefano' # assegnazione diretta ai dati membri
p1.cognome = 'riccio'
p1.cambia_indirizzo('via html n. 10')  # richiamo un metodo
p1.stampa()    # risultato : stefano riccio via html n. 10
```

Figura 6 Codice per creare un "oggetto" e richiamare i relativi metodi in Python

[28] Per convenzione, il primo argomento di ogni metodo di una classe viene chiamato "self". Questo argomento ricopre il ruolo della parola riservata "this" in C++ o Java, ma self non è una parola riservata in Python è semplicemente una convenzione sui nomi.

p1 è un oggetto della classe persona che contiene i valori visibili in Figura 6. Si può notare come è stato possibile assegnare ai dati membri i valori direttamente oppure attraverso gli appositi metodi.

Si è utilizzato il metodo “cambia_indirizzo” scrivendo un solo parametro (anche se il metodo ne richiederebbe due). Questo è stato possibile perché Python associa al parametro “self” l’oggetto p1. Utilizzando self posso accedere direttamente ai dati membro all’interno della classe.

Per ulteriori dettagli relativi alla programmazione Python, fare riferimento al “Manuale utente” di questa tesi, dove sarà possibile comprendere molte delle strutture presentate in questo capitolo, direttamente applicate al codice di programmazione del simulatore.

Capitolo 6

MANUALE UTENTE

La motivazione che mi ha spinto a scrivere il seguente capitolo è la volontà di voler avvicinare, sopratutto coloro i quali non conoscono il linguaggio di programmazione Python, alla comprensione delle modalità operative del mio modello. Il mio intento è, quindi, non solo quello di poter chiarire situazioni riguardanti il simulatore, bensì qualora necessario, di fornire una guida di concetti informatici spiegati in modo semplificato, come introduzione alla programmazione. La necessità di un percorso non tecnico, che guida verso la creazione di un simulatore, è propria di tutti coloro i quali, desiderino realizzare un progetto preciso ma con una conoscenza informatica non adeguata. Il "Manuale utente" nasce, quindi, dal reale bisogno che ho riscontrato all'inizio della creazione del simulatore, poichè i numerosi manuali, libri di testo e codici informatici a disposizione, in realtà, potrebbero rivelarsi di difficile comprensione per una persona poco esperta che voglia intraprendere un progetto di simulazione di dati legato ai mercati finanziari. Per comprensione non intendo capire, in maniera superficiale, come opera un programma informatico, il significato delle varie funzioni e del linguaggio in generale, bensì la capacità di immaginare un

problema e convertire i risultati della propria immaginazione su una piattaforma informatica, attraverso un codice di programmazione.

Uno degli ostacoli maggiori, da me riscontrati, è riassunto molto bene nel titolo di un manuale, consigliato durante il corso universitario di "Informatica e simulazioni per l'economia": pensare da informatico.

Questo capitolo sarà suddiviso in più parti che rappresentano i passi da me compiuti nella creazione e nella strutturazione del programma. Leggendo il capitolo ci si accorgerà che le evoluzioni presentate sono quelle proposte dallo schema Slapp. Inoltre, sottolineo che il capitolo è stato scritto durante il processo reale di programmazione, pertanto alcune variabili e funzioni hanno subito, quasi certamente, delle modifiche rispetto alla prima programmazione. Esorto, dunque, il lettore a fare affidamento, per il codice ufficiale, a quello che sarà riportato al fondo di questa tesi.

Manuale Utente Parte 1

FILES: Agent.py , Book.py, Model.py, Start.py

1.1 File: Agent.py

In questo file è stato programmato il comportamento di un agente *zero intelligence*.
L'agente esprime la propria volontà di acquisto o di vendita in base all'esito del valore generato dalla funzione "*random.random()"*, la quale restituisce un valore casuale compreso tra 0 e 1. Se la variabile in questione è un valore superiore a 0.5, l'agente casuale vende; nel caso in cui il valore della variabile casuale sia inferiore a 0.5, l'agente casuale acquista.
Il limite, di cui si è parlato, è stato scelto volontariamente pari al valore medio dell'intervallo di numeri casuali che possono essere generati, al fine di avere, almeno da un punto di vista teorico, un equilibrio tra compratori e venditori.
Il prezzo presentato dagli agenti è pari all'ultimo prezzo di contrattazione del titolo, registrato nella "*priceList*", moltiplicato per un coefficiente. Tale coefficiente è una variabile casuale uniformemente distribuita in un intervallo compreso tra 0,5 e 1,5.

I prezzi e le identità degli agenti protagonisti delle offerte sono inviati al *Book*, il quale ha il compito di gestire le transazioni.

I moduli "*setExecutedBuyOrder"* e "*setExecutedSellOrder"* ricevono informazioni dal *Book* nel caso di esito positivo della transazione. Per ogni transazione conclusa, il *Book* invia ai rispettivi agenti la comunicazione del risultato raggiunto. Dal momento che si sta trattando il caso di agenti *zero intelligence,* l'informazione trasmessa è semplicemente l'identità degli agenti che hanno acquistato e venduto, a cui è legata inoltre la quantità unitaria commerciata.

In caso di acquisto, la quantità sarà positiva pari a uno; in caso di vendita invece, la quantità sarà negativa pari a uno. Così facendo, aumentando il numero di cicli, si saprà in ogni istante il bilancio di operazioni fatto da ciascun agente.

Nel mio modello, intenzionalmente, non ho ancora approfondito la questione delle quantità offerte in acquisto e offerte in vendita, al fine di evitare problemi legati agli ordini parzialmente eseguiti.

```
import random

class Agent:
    def __init__(self,number,priceList,aBook):
        self.number=number
        self.assets = 0
        #self.assetsS = 0
        self.aBook=aBook
        self.priceList=priceList
        #print "Agent number", self.number,

    def randomTransaction(self):
        self.randomValue=random.random()
        self.price=self.priceList[-1]
        self.SellPrice=0
        self.BuyPrice=0

        if self.randomValue > 0.5:
            self.SellPrice=self.price*random.uniform(0.5,1.5)
            print "agent=",self.number,"SellPrice","%.2f" % self.SellPrice
            self.aBook.setSellDecision([self.SellPrice,self])
        if self.randomValue < 0.5:
            self.BuyPrice=self.price*random.uniform(0.5,1.5)
            print "agent=",self.number, "Buy price","%.2f" % self.BuyPrice
            self.aBook.setBuyDecision([self.BuyPrice,self])


    def setExecutedBuyOrder(self):
        self.assets+=1
        print "Agent",self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
        print "Agent",self.number,"Assets",self.assets
```

Figura 1 Linguaggio di programmazione del file agent

1.2 File: Book.py

In Figura 2 è presentato il codice di programmazione del *Book* di transazione.

Come si può notare, la prima istruzione importa il file *"agent"*salvato nella medesima cartella di lavoro.

Nell' *"__init__"* sono presenti due voci che saranno esclusivamente utilizzate per la stampa dei prezzi, così da poter visualizzare meglio il corretto funzionamento del programma.

Il metodo "*SetBuyDecision*" riceve le offerte di acquisto (Bid) espresse dagli agenti e, nel caso in cui ci sia immediata controparte, conclude la transazione nello stesso momento in cui riceve l'offerta. In caso contrario, ovvero qualora non ci siano proposte

di vendita o le proposte di vendita presenti in lista non soddisfino l'offerta di acquisto corrente, quest'ultima sarà inserita nella lista delle offerte di acquisto, che a loro volta saranno ordinate in modo decrescente attraverso il comando "*.reverse*".

L'ordinamento degli ordini è una parte fondamentale del *Book* perché, seguendo il regolamento del *Book* reale di transazione, l'offerta di acquisto che ha priorità sulle altre offerte presenti in lista è quella con valore maggiore.

La variabile "*firstInformationInList*" rappresenta il valore inferiore tra le proposte di vendita registrate nella "*sellList*". Tale valore è il prezzo con cui si conclude la transazione nel caso in cui l'offerta di acquisto corrente fosse di pari misura o superiore. Questo principio segue il regolamento di transazione, per cui se un'offerta di vendita è già presente sul mercato e non è ancora stata eseguita, l'offerta di acquisto che sarà presentata in futuro, secondo una logica di mercato, sarà al massimo pari e non superiore. Nel nostro caso, le offerte di acquisto che giungeranno al Book potranno essere anche superiori al minor prezzo di vendita presente in "sellList" perché gli agenti *zero intelligence* presentano offerte casuali. Tramite questa programmazione, cioè che "*firstInformationInList*" sarà il prezzo di transazione ufficiale del titolo, il principio di logica di mercato non sarà violato.

All'interno della condizione "*else*" sono elencati i comandi che permettono la conclusione della transazione spiegati precedentemente; inoltre, è presente il codice che invia le informazioni al file "*agent.py*" necessarie per visualizzare identità e bilancio titoli per ciascun agente.

Nello stampare la "*buyList*", ho dovuto inserire un comando che mi permettesse di visualizzare solamente due cifre decimali dopo la virgola. Poichè i valori *float* si trovano all'interno di una lista, la procedura risulta differente da quella vista in precedenza per la stampa delle offerte pervenute dai singoli agenti e dal "*Trading price*".

```
import agent

class Book:
    def __init__(self,priceList):
        self.priceList=priceList
        self.buyList=[]
        self.sellList=[]
        self.buyPrintList=[]
        self.sellPrintList=[]

    def setBuyDecision(self,information):

        price = information[0]
        identity = information[1].number

        if len(self.sellList)==0:
            self.buyList.append(information)
            self.buyList.reverse()
            self.buyPrintList.append(information[0])
            self.buyPrintList.reverse()
          #  print "informazione Buy",information[0]
        elif price < self.sellList[0][0]:
            self.buyList.append(information)
            self.buyList.reverse()
            self.buyPrintList.append(information[0])
            self.buyPrintList.reverse()
          #  print "prezzo<selList"
        else:
            firstInformationInList=self.sellList.pop(0)
            firstInformationInListPrint=self.sellPrintList.pop(0)
            self.priceList.append(firstInformationInList[0])
            print "TRADING PRICE","%.2f" % firstInformationInListPrint
            information[1].setExecutedBuyOrder()
            firstInformationInList[1].setExecutedSellOrder()

        print "Buy List",",".join(("%.2f" % price for price in self.buyPrintList))
        print "Sell List",",".join(("%.2f" % price for price in self.sellPrintList))
```

Figura 2 Parte prima del codice di programmazione Book di negoziazione

In Figura 3 è possibile osservare la seconda parte del codice di programmazione del Book.

Il modulo *"setSellDecision"* è la versione speculare,per offerte di vendita, del modulo "*setBuyDecision*".

Le novità di questa seconda parte sono:

- il modulo che cancella le liste, una volta ultimata la giornata di negoziazione
- il modulo che si occupa della stampa dei prezzi di conclusione delle transazioni, che troviamo nella "*priceList*".

```
def setSellDecision(self,information):

    price = information[0]
    identity = information[1].number

    if len(self.buyList)==0:
        self.sellList.append(information)
        self.sellList.sort()
        self.sellPrintList.append(information[0])
        self.sellPrintList.sort()
       # print "informazione Sell",information[0]
    elif price > self.buyList[0][0]:
        self.sellList.append(information)
        self.sellList.sort()
        self.sellPrintList.append(information[0])
        self.sellPrintList.sort()
      #  print "prezzo>sellList"
    else:
        firstInformationInList=self.buyList.pop(0)
        self.priceList.append(firstInformationInList[0])
        firstInformationInListPrint=self.buyPrintList.pop(0)
        #firstInformationInListPrint=firstInformationInList[0]
        print "TRADING PRICE","%.2f" % firstInformationInListPrint
        firstInformationInList[1].setExecutedBuyOrder()
        information[1].setExecutedSellOrder()

    print "Sell List",",".join(("%.2f" % price for price in self.sellPrintList))
    print "Buy List",",".join(("%.2f" % price for price in self.buyPrintList))

def cleanLists(self):
    del self.buyList[:]
    del self.sellList[:]
    del self.priceList[1:]
    del self.buyPrintList[:]
    del self.sellPrintList[:]

def getStartingPrice(self):
    print "price List", ','.join(("%.2f" % price for price in self.priceList))
    #return self.priceList
```

Figura 3 Parte seconda del codice di programmazione Book di negoziazione

1.3 File: Model.py

Il codice di programmazione del file *model*, visibile in Figura 4, è stato studiato per la gestione dei file necessari per la simulazione, richiamati attraverso "*import*".

Seguendo il protocollo di programmazione Slapp, la classe ModelSwarm contiene i moduli per la creazione degli oggetti e per la gestione delle azioni.

Il modulo *"builObjects"* crea "*self.aBook*" e "*anAgent*". La serie di oggetti *"anAgent"* sono inseriti all'interno di una lista *"AgentList".* Così facendo, si segue uno dei fondamenti della programmazione ad oggetti quale la creazione degli agenti separata dal codice di azione di questi ultimi. Il numero di agenti creati ed inseriti in lista sarà scelto dall'utente tramite l'inserimento di un *input* presente nel codice di programmazione del file *"start.py"*.

Nel modulo *"buildRandomActions"* sono gestite le azioni e i tempi.

I tempi sono rappresentati dal numero di cicli i quali saranno decisi dall'utente tramite inserimento di *input* dal file *"start.py".*

```
import agent
import book
import random

class ModelSwarm:
    def __init__(self, nAgents,numCicle):
        self.numCicle=numCicle
        self.nAgents = nAgents
        self.AgentList = []
        self.price= 1
        self.priceList=[self.price]

    def buildObjects(self):
        self.aBook=book.Book(self.priceList)


        for i in range(self.nAgents):
            anAgent = agent.Agent(i,self.priceList,self.aBook)

            self.AgentList.append(anAgent)


    def buildActions(self):             #non viene richiamato da start!
        #self.aBook.cleanLists()
        for anAgent in self.AgentList:
            anAgent.randomTransaction()
        self.aBook.getStartingPrice()

    def buildRandomActions(self):
        random.shuffle(self.AgentList)
        m=0
        while m!=self.numCicle:
            for anAgent in self.AgentList:
                anAgent.randomTransaction()
            self.aBook.getStartingPrice()
            self.aBook.cleanLists()
```

Figura 4 Codice di programmazione file model

1.4 File: Start.py

File necessario per l'avvio della simulazione. Una volta avviata la simulazione, all'utente sarà richiesto l'inserimento di due input che rispettivamente rappresentano:

- il tempo: attraverso il numero di cicli
- grandezza del mercato per numero di partecipanti: attraverso il numero di agenti

```
import model
import book
import agent

numCicle=input("How many cicles?")
nAgents=input("How many agents?")

ModelS= model.ModelSwarm(nAgents,numCicle)
ModelS.buildObjects()
#ModelS.buildActions()
ModelS.buildRandomActions()
```

Figura 5 Codice di programmazione file start

Manuale Utente Parte 2

FILES: ActionGroup.py , Model.py, Tools.py

2.1 File: ActionGroup.py

Nel file ActionGroup.py è stata programmata la classe omonima con l'obiettivo di inviare strutture di messaggi agli agenti interni alla classe "*Model*".

Le stringhe inviate da "*ActionGroup",* che richiamano le differenti azioni compiute in simulazione, verranno inserite all'interno della lista "*self.actionGroupList*" che rappresenta il piano di esecuzione delle azioni del mio programma.

```
#ActionGroup.py

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

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

Figura 1 Codice di programmazione riferito a ActionGroup.py

2.2 Model.py

Il questa versione aggiornata del "Model" si è cercato di seguire la procedura di programmazione spiegata nel *tutorial* 5 di Slapp. Come è possibile osservare da Figura 3 nella creazione delle azioni non sono presenti cicli "for", programmati nel file "Tools.py".

L'*"__init__"* e il metodo *"buidobject*" sono rimasti invariati dalla prima versione di programmazione.

Le novità le troviamo all'interno del metodo "*buildActions*".

La classe *"ActionGroup",* richiamata in questo metodo, invia il messaggio "bid_ask" a tutti gli agenti creati ed inseriti all'interno della lista *"AgentList".* Con questo messaggio si sta ordinando ai protagonisti della simulazione di formulare un'offerta di acquisto o di vendita.

Rispetto alla versione precedente del modello, ora il prezzo offerto al tempo "*t*" è frutto della funzione "random.gauss (u,s)". Il valore che indica la media è pari all'ultimo prezzo di negoziazione elencato in "priceList". Per quanto riguarda il valore che indica la deviazione standard, ho scelto volutamente 0.03 per avere maggiore variabilità nei prezzi offerti.

```
if self.randomValue > 0.5:
    self.SellPrice= random.gauss(self.price,0.03)
    print "agent=",self.number,"SellPrice","%.2f" % self.SellPrice
    self.aBook.setSellDecision([self.SellPrice,self])
if self.randomValue < 0.5:
    self.BuyPrice= random.gauss(self.price,0.03)
    print "agent=",self.number, "Buy price","%.2f" % self.BuyPrice
    self.aBook.setBuyDecision([self.BuyPrice,self])
```

Figura 2 Modalità di formazione dei prezzi di offerta, tratto da Agent.py

Tornando alla programmazione del "Model", la lista contenente gli agenti, sarà copiata in *"AgentListCopy";* questa procedura viene fatta per disporre in ordine casuale, attraverso il comando *".shuffle*", le informazioni contenute nella lista.

L'aumento della casualità del programma è il risultato dell'operazione sopra descritta.

Con l'istruzione "*Tools.askEachAgentIn*" è richiamato un metodo interno a "*Tools*" che con l'ausilio di un ciclo "*for*" eseguirà le offerte di ciascun agente elencandole nel "*Book"*. Per maggiori dettagli a riguardo di questa funzione vedere il Capitolo 1 del "Manuale Utente" nella sezione "File: Agent.py" e " Book.py".

Con il modulo "do1" è richiamato il "*Tools*"per la stampa della lista dei prezzi di transazione.

Con il modulo "do2" è richiamato il "Tools" per la pulizia delle liste a conclusione della giornata di transazioni.2

Nella "*actionGroupList*" sono elencate le azioni eseguite dal "*run*" assegnate dalla classe "*ActionGroup*".

```
import Tools
import Agents
import Book
import random
import ActionGroup

class ModelSwarm:
    def __init__(self, nAgents,numCycles):
        self.numCycles=numCycles
        self.nAgents = nAgents
        self.AgentList = []
        self.price= 1
        self.priceList=[self.price]

    def buildObjects(self):
        self.aBook=Book.Book(self.priceList)


        for i in range(self.nAgents):
            anAgent = Agents.Agent(i,self.priceList,self.aBook)

            self.AgentList.append(anAgent)

    def buildActions(self):

        self.actionGroup1 = ActionGroup.ActionGroup ("bid_ask")
        def do1(address, numCycles):

            address.AgentListCopy = address.AgentList[:]
            random.shuffle(address.AgentListCopy)

            Tools.askEachAgentIn(address.AgentListCopy,Agents.Agent.randomTransaction)

        self.actionGroup1.do = do1

        self.actionGroup2 = ActionGroup.ActionGroup ("check")
        def do2(address):
            Tools.askBookCheckIn(address.aBook.getStartingPrice())
        self.actionGroup2.do = do2

        self.actionGroup3 = ActionGroup.ActionGroup ("clean")
        def do3(address):
            Tools.askListsCleaned(address.aBook.cleanLists())
        self.actionGroup3.do = do3

        #schedule
        self.actionGroupList = ["bid_ask","check","clean"]
```

Figura 3 Parte prima file Model.py

Programmando il modulo “buildActions” secondo il protocollo di programmazione del *tutorial* 5 risulta semplice la gestione del “*run”*. Esso si basa su una serie di condizioni che devono verificarsi all’interno dell’*”actionGroupList”*. Ad ogni condizione è legata l’esecuzione di un metodo.

```
def run(self):
    for n in range(self.numCycles):
        for s in self.actionGroupList:
            if s== "bid_ask":
                self.actionGroup1.do(self,n)
            if s== "check":
                self.actionGroup2.do(self)
            if s== "clean":
                self.actionGroup3.do(self)

        print "\n"
```

Figura 4 Parte seconda file Model.py

2.3 Tools.py

In Figura 5 è visualizzabile la programmazione dei moduli interni al file “Tools.py”. In questa prima versione la programmazione risulta essere molto semplice. Questi moduli sono richiamati dal file “Model.py”.

```
import random


def askEachAgentIn(collection,method):

    for a in collection:
            method(a)

def askBookCheckIn(method):
    return method

def askListsCleaned(method):
    return method
```

Figura 5 Tools.py

Manuale Utente Parte 3

FILES: Book.py, Model.py, Observer.py, Start.py

Le novità a cui si porrà maggiore attenzione in questo capitolo riguarderanno:

- definizione di "*step*". Ad ogni "*step*", tutti gli agenti saranno interrogati sulla propria volontà di partecipare al mercato. Coloro i quali saranno considerati partecipanti, quindi agenti attivi, sono elencati all'interno di una lista definita *"self.agentActionList"*. I membri di questa lista faranno offerte di acquisto o di vendita, come già presentato nei capitoli precedenti del manuale utente. Il problema, che si è presentato durante la scrittura del programma, è la possibilità che lo stesso agente concluda una transazione con se stesso. Questo si verifica nel caso in cui, in due "*step"* diversi, due offerte di acquisto e di vendita siano presentate dal medesimo agente ed esse risultino le migliori sul mercato. Tale problema è stato risolto programmando il "*Book*" in modo tale da poter scorrere e riconoscere le identità degli agenti a cui appartiene ciascuna offerta inserita in lista: ciò impedirà che la transazione avvenga tra medesimi agenti. Essa avverrà, invece, con la prima miglior offerta di vendita,

elencata nella lista, presentata da un agente diverso dal proponente offerta di acquisto.

- creazione della classe "*ObserverAgent,*" la quale gestirà la simulazione come studiato nel *tutorial* 6 di Slapp
- creazione del modulo che utilizza la funzione "*.write*", che copia i prezzi di transazione all'interno di un file *".txt"*

Le classi e i file che non hanno subito cambiamenti nel programma, rispetto alla versione precedente, non saranno presentati. Per visionarli, si dovrà fare riferimento ai capitoli precedenti di questo manuale utente.

3.1 File: Book.py

In Figura 1 è possibile osservare il programma di una parte del Book.

Il problema, enunciato nell'introduzione di questo capitolo, è stato risolto all'interno di questa classe. Ricordo che con "*information*" stiamo facendo riferimento ad un vettore di due colonne: nella prima troviamo il prezzo di offerta e nella seconda l'identità dell'agente offerente.

Nel secondo "*elif*", programmato in "*setBuyDecision*", si verifica la condizione in cui l'offerta di acquisto pervenuta da un agente è idonea alla transazione, ma l'identità è coincidente con la migliore offerta di vendita elencata in "*self.sellList*". In questo caso, viene fatto uno *screening* delle offerte di vendita elencate in lista. Quando saranno trovati i dati che soddisfano la verifica, quindi identità diverse compatibilmente con il valore delle offerte, essi saranno estratti dalla lista e la transazione sarà conclusa.

Nella possibilità in cui, tutte le offerte di vendita presenti in lista appartengano allo stesso agente o non esista compatibilità di prezzo con le offerte di vendita successive alla prima, l'offerta di acquisto in questione è elencata in "*buyList*".

Le stringhe stampate, visibili in programmazione, sono state inserite esclusivamente per una verifica di funzionamento, ma nel programma ufficiale saranno eliminate.

Obiettivo futuro sarà quello di snellire il programma, creando una funzione che raccoglierà le righe di programmazione ridondanti, presenti in Figura 1.

Il modulo "*setSellDecision*" è programmato in maniera speculare al "*setBuyDecision*"; la differenza sostanziale consiste nelle offerte di vendita che giungeranno al *Book* e non nelle offerte di acquisto, a cui sopra si è fatto riferimento.

```
class Book:
    def __init__(self,priceList):
        self.priceList=priceList
        self.buyList=[]
        self.sellList=[]
        self.finalPriceForStep=[]

    def setBuyDecision(self,information):
        price = information[0]
        identity = information[1]

        if len(self.sellList)==0:
            self.buyList.append(information)
            self.buyList.sort(reverse = True)
        elif price < self.sellList[0][0]:
            self.buyList.append(information)
            self.buyList.sort(reverse=True)
        elif identity == self.sellList[0][1]:
            for i in range(0,len(self.sellList)):
                if identity != self.sellList[i][1]:
                    if price >= self.sellList[i][0]:
                        firstInformationInList=self.sellList.pop(i)
                        self.priceList.append(firstInformationInList[0])
                        print "TRADING PRICE","%.2f" % firstInformationInList[0]
                        information[1].setExecutedBuyOrder()
                        firstInformationInList[1].setExecutedSellOrder()
                        print "transazione avvenuta con il successivo",i
                        break
                else:
                    print "Sn sempre io",i
                    if i==(len(self.sellList)-1):
                        self.buyList.append(information)
                        self.buyList.sort(reverse=True)
                        print "solo io faccio offerte"
                        break
        else:
            firstInformationInList=self.sellList.pop(0)
            self.priceList.append(firstInformationInList[0])
            print "TRADING PRICE","%.2f" % firstInformationInList[0]
            information[1].setExecutedBuyOrder()
            firstInformationInList[1].setExecutedSellOrder()
```

Figura 1 Programma del modulo "setBuyDecision" tratto da Book di borsa

In Figura 2 sono programmati i moduli per la creazione della lista, che inserirà un prezzo di transazione per *"step"*. Questo prezzo coincide con l'ultimo prezzo di transazione avvenuto all'interno di ciascun "*step*" oppure con il prezzo di transazione dello *"step"* precedente.

Il modulo "*getFinalPriceForStep*" è utilizzato per copiare i "*finalPriceForStep*" in un file esterno txt.

```
def setFinalPriceForStep(self):
    self.finalPriceForStep.append(self.priceList[-1])
    #print "end_Of_StepPrice", ','.join("%.2f" % price for price in self.finalPriceForStep)

def getFinalPriceForStep(self):
    return self.finalPriceForStep
```

Figura 2 Programma del modulo "setFinalPriceForStep" tratto da Book di borsa

3.2 File: Model.py

Le novità presenti nel file "*Model*" sono rappresentate dalla creazione del modulo *"step"* e dalla condizione inserita all'interno del modulo "do1" che permette di scindere gli agenti tra attivi ed inattivi, in base ad una funzione di probabilità, ad ogni step.

Ogni agente è estratto dalla lista e interrogato sulla propria volontà. Nel caso in cui la variabile casuale "*action*" presenti valore superiore a 0.5, l'agente sarà considerato attivo e sarà elencato in "agentActionList". In caso contrario, l'agente sarà considerato inattivo e non parteciperà al mercato per il tempo di uno step. Coloro i quali saranno elencati in "*agentActionList*" parteciperanno al mercato.

Altra novità, visibile in Figura 3, riguarda la cancellazione delle liste degli agenti considerati attivi. La sequenza delle operazioni è elencata nella "*schedule"*.

```
def buildActions(self):

    self.actionGroup1 = ActionGroup.ActionGroup ("active_inactive")
    def do1(adress):
        adress.agentListCopy = adress.agentList[:]
        random.shuffle(adress.agentListCopy)

        while len(adress.agentListCopy)!=0:
            for i in adress.agentListCopy:
                action = random.random()
                ask_will_single_agent = adress.agentListCopy.pop(0)
                if action >0.5:
                    self.agentActionList.append(ask_will_single_agent)
                    #print "ask_will_single_agent",ask_will_single_agent
                else:
                    inactive = ask_will_single_agent
                    #print "inactive",ask_will_single_agent

    self.actionGroup1.do=do1

    self.actionGroup2 = ActionGroup.ActionGroup ("bid_ask")
    def do2(adress):
        adress.agentActionListCopy = self.agentActionList[:]
        Tools.askEachAgentIn(adress.agentActionListCopy,Agents.Agent.randomTransaction)
    self.actionGroup2.do = do2

    self.actionGroup3 = ActionGroup.ActionGroup ("cleanActionList")
    def do3(adress):
        del self.agentActionListCopy[:]
        del self.agentActionList[:]
    self.actionGroup3.do = do3

    #schedule
    self.actionGroupList = ["active_inactive","bid_ask","cleanActionList"]
```

Figura 3 Programma del modulo "buildActions" tratto dal Model

In Figura 4 è possibile osservare la programmazione del modulo "*step*", il quale sarà richiamato dal file "Observer.py". La programmazione di questa funzione è molto simile alla programmazione del modulo "*run*". Ad ogni *"step",* gli agenti saranno interrogati per stabilire la loro attività/inattività; quindi, a tutti coloro i quali sono attivi sarà chiesto di fare un'offerta. Per ultimo, le liste che includono gli agenti attivi, saranno cancellate per poter ricominciare un nuovo "*step*".

```
    def step(self):
        for s in self.actionGroupList:
            if s=="active_inactive":
                self.actionGroup1.do(self)
            if s=="bid_ask":
                self.actionGroup2.do(self)
            if s=="cleanActionList":
                self.actionGroup3.do(self)
```

Figura 4 Programma del modulo "step" tratto dal Model

3.3 Observer.py

Il file “Observer.py” è stato programmato studiando il *tutorial* 6. Come già anticipato nella teoria del capitolo di Slapp, questo file rappresenta un’ulteriore stratificazione nella gestione del programma.

Nel modulo “*buildObjects*”, oltre alla riga di programmazione che richiama il modulo omonimo in “*Model.py*”, è possibile osservare la serie di “*input*” che saranno inseriti da un utente esterno e che definiscono le dimensioni della nostra simulazione. Essi sono, appunto, legati al numero di cicli, al numero di agenti e al numero di step per ciclo.

```
class ObserverAgent:
    def __init__(self):
        self.AgentList = []
        self.price= 1
        self.priceList=[self.price]

    #create objects
    def buildObjects(self):

        self.numCycles=input("How many cycles?")
        self.nAgents=input("How many agents?")
        self.steps = input("How many step?")

        self.modelSwarm = Model.ModelSwarm(self.nAgents)

        self.modelSwarm.buildObjects()
```

Figura 5 Programma del modulo “buildObjects” tratto dal ObserverAgent

Il modulo programmato in Figura 6 permette di comprendere la gestione e la costruzione delle azioni. Il modulo “*do4*” non solo richiama il modulo “*step*” programmato all’interno della classe “*ModelSwarm*”, bensì anche il modulo programmato all’interno della classe “*Book*”, che ha il compito di scrivere all’interno di una lista quelli che sono i prezzi finali di ogni “*step*”.

Il modulo “do6” copia, all’interno di un file esterno *“.txt”*, i prezzi che saranno oggetto di successivi studi, risultanti dalla simulazione. Non tutti i prezzi, infatti, saranno considerati.

Nella “*schedule”* è possibile visionare l’organizzazione delle azioni.

```
#actions
def buildActions(self):

    self.modelSwarm.buildActions()

    self.actionGroup4=ActionGroup.ActionGroup("step")
    def do4(adress):
        for i in range(self.steps):
            adress.modelSwarm.step()
            adress.modelSwarm.aBook.setFinalPriceForStep()
    self.actionGroup4.do = do4

    self.actionGroup5 = ActionGroup.ActionGroup ("get_prices")
    def do5(adress):
        adress.modelSwarm.aBook.getStartingPrice()
    self.actionGroup5.do = do5

    self.actionGroup6 = ActionGroup.ActionGroup ("print_txt")
    def do6(adress):
        with open("prices.txt","w") as f:
            f.write(str(adress.modelSwarm.aBook.getFinalPriceForStep()))
            f.close()
            print "print_file_txt"
    self.actionGroup6.do = do6

    self.actionGroup7 = ActionGroup.ActionGroup ("clean")
    def do7(adress):
        adress.modelSwarm.aBook.cleanLists()
    self.actionGroup7.do = do7

#schedule
    self.actionGroupList = ["step","get_prices","print_txt","clean"]
```

Figura 6 Programma del modulo "buildActions" tratto dal ObserverAgent

Ultima parte di programmazione del file "ObserverSwarm" è rappresentata dal programma visibile in Figura 7. Il "run" stabilisce la sequenza delle operazioni, che devono essere fatte in sequenza e in accordo con i valori stabiliti in *"input".*

```
#run
def run(self):
    for n in range(self.numCycles):
        for s in self.actionGroupList:
            if s== "step":
                self.actionGroup4.do(self)
            if s== "get_prices":
                self.actionGroup5.do(self)
            if s== "print_txt":
                self.actionGroup6.do(self)
            if s== "clean":
                self.actionGroup7.do(self)
        print
```

Figura 7 Programma del modulo "run" tratto dal ObserverAgent

Manuale Utente Parte 4

FILES: Book.py, TrendAgents.py, VolumeAgents.py, BestOffertAgents.py, Model.py

Le evoluzioni del programma presentate in questo capitolo interessano, rispetto alla versione precedente, i seguenti punti:

- modifica di una delle funzioni interne al Book. In questa versione, ogni qual volta un'offerta di acquisto o di vendita perverrà al Book, esso controllerà che lo stesso agente non abbia fatto offerte di "segno[29]" opposto. Nel caso in cui, l'agente avesse fatto offerte di "segno" opposto in "*step*" precedenti, il Book le eliminerà.
- creazione di diverse classi, ciascuna delle quali rappresenta un agente diverso. Gli agenti saranno dotati di intelligenza e presenteranno le loro offerte di mercato seguendo una strategia. Alcuni, ad esempio, potranno esprimere la propria offerta confrontando l'ultimo prezzo negoziato con la media mobile di un numero variabile di prezzi negoziati, partendo dal tempo (t-2); altri

[29] Con questa espressione mi riferisco alle offerte di acquisto, nel caso in cui al Book pervenga un'offerta di vendita. Viceversa, nel caso in cui al book pervenga un'offerta di acquisto.

esprimeranno la propria decisione in base ai volumi di acquisto e di vendita. Altri ancora avanzeranno offerte “al meglio”, le quali permetteranno loro di concludere una transazione con certezza.

4.1 File: Book.py

In Figura 1 è possibile osservare le modifiche al programma del Book, presentate nell’introduzione del capitolo. La prima condizione “*if*” verifica che nella lista opposta non ci siano offerte avanzate dallo stesso soggetto proponente l’offerta in questione. Nel caso in cui si trovino offerte con la medesima identità, esse non verranno copiate nella lista, detta “sellList”. Attraverso questa modifica, si eviteranno problematiche future legate al numero eccessivo di offerte elencate in lista. Un altro problema che sarà evitato è quello legato alla “non coerenza strategica”: se un soggetto esprime una volontà di acquisto, nello stesso istante egli non può essere incluso all’interno di una lista per proposte di vendita.

Altro comando visibile in Figura 1 è “*self.getVolBuy*”. Esso richiama un modulo programmato all’interno della classe “Book”, che tiene il conto del numero di offerte di acquisto pervenute. Specularmente, è presente lo stesso comando per le offerte di vendita, le quali saranno utilizzate dall’agente, che baserà la propria strategia sui volumi di offerte di acquisto e di vendita.

```
def setBuyDecision(self,information):
    price = information[0]
    identity = information[1]

    self.getVolBuy()


    if len(self.sellList)>0:
        self.sellList2 = []
        for i in self.sellList:
            if identity != i[1]:
                self.sellList2.append(i)
        self.sellList = self.sellList2[:]

    if len(self.sellList)==0:
        self.buyList.append(information)
        self.buyList.sort(reverse = True)
    elif len(self.sellList)>0:
            if price < self.sellList[0][0]:
                self.buyList.append(information)
                self.buyList.sort(reverse=True)
            else:
                firstInformationInList=self.sellList.pop(0)
                self.priceList.append(firstInformationInList[0])
                print "TRADING PRICE","%.2f" % firstInformationInList[0]
                information[1].setExecutedBuyOrder()
                firstInformationInList[1].setExecutedSellOrder()
```

Figura 1 Modulo "setBuyDecision" tratto dal file Book.py

In Figura 2 è possibile visualizzare i moduli di programmazione, che forniranno valori utili alle nuove classi di agenti. "*getMaxPriceSellList*" fornirà alla classe "*UniformVolAgent*" il miglior prezzo di vendita presente in lista. Questo permetterà all'agente, facente capo a questa classe, di esprimere un'offerta di acquisto vincente, ogni qual volta fosse chiamato ad operare. Stesso principio ma di segno opposto è quello espresso da "*getMaxPriceBuyList*".

"*getVolBuy*" è il modulo che aggiorna la variabile programmata per il conteggio delle offerte di acquisto. Esso, unito a "*getVolBuy*", sarà usato dalla classe che attuerà la propria strategia sulla base dei volumi di acquisto e di vendita.

```
    def getMaxPriceSellList(self):
        self.maxSellPrice = self.priceList[-1]
        if len(self.sellList)>0:
            self.maxPriceSellList = self.sellList[0][0]
            return self.maxPriceSellList
        else:
            return self.maxSellPrice

    def getMaxPriceBuyList(self):
        self.maxBuyprice = self.priceList[-1]
        if len(self.buyList)>0:
            self.maxPriceBuyList = self.buyList[0][0]
            return self.maxPriceBuyList
        else:
            return self.maxBuyprice

    def setFinalPriceForStep(self):
        self.finalPriceForStep.append(self.priceList[-1])
        #print "end_Of_StepPrice", ','.join("%.2f" % price for price in self.fin

    def getFinalPriceForStep(self):
        return self.finalPriceForStep

    def getVolBuy(self):
        self.volBuy+=1
        #print "BUY Volums",self.volBuy

    def getVolSell(self):
        self.volSell-=1
        #print "SELL Volums",self.volSell
```

Figura 2 Moduli interni al file Book.py

4.2 File: TrendAgents.py

In Figura 3 è possibile osservare la programmazione della classe “TrendAgent”.

Ogni classe di agenti ha internamente una variabile definita da un’apposita dicitura, che riprende il nome e la natura della classe di appartenenza dell’agente. Questa dicitura è fondamentale per la programmazione, poiché consente di chiamare gli agenti, raccolti all’interno della lista “agentActionList”, ad operare in base alla loro classe di appartenenza. Questa condizione è osservabile all’interno del file “*Model.py*”.

Essi non sono attivi all’apertura dei mercati, ma entrano in gioco qualora un sufficiente numero di contratti sia stato concluso.

Gli agenti che fanno parte di questa classe decidono di acquistare o di vendere in base ad un confronto tra l’ultimo prezzo negoziato e una media mobile di prezzi, variabile

da agente ad agente. Nel caso in cui il risultato della media mobile sia inferiore all'ultimo prezzo negoziato, l'agente farà un'offerta di acquisto, poiché la fase di mercato in cui egli opera è in rialzo. Nel caso in cui, invece, il risultato della media mobile sia superiore all'ultimo prezzo negoziato, l'agente proporrà un'offerta di vendita.

Le offerte di vendita e di acquisto proposte hanno come base di partenza l'ultimo prezzo negoziato moltiplicato o diviso per un coefficiente, che è una variabile uniformemente distribuita tra 1 e 1,2. In caso di acquisto, si moltiplica l'ultimo prezzo negoziato per il coefficiente "self.coefficient"; viceversa, in caso di vendita, l'ultimo prezzo negoziato si divide per il coefficiente "self.coefficient".

La decisione di offerta è, comunque, legata ad una variabile "*random*" al fine di aumentare la variabilità del programma.

Altri moduli programmati, ma non visibili in Figura 3, riguardano le istruzioni inviate dal Book alla classe "TrendAgent" per avvisare ogni agente dell'avvenuta transazione.

```
def trendTransaction(self):
    self.randomProbability = random.random()
    self.price = self.priceList[-1]
    self.coefficient = random.uniform(1,1.2)
    self.sellPrice = 0
    self.buyPrice = 0
    self.averagePrice = 0
    self.sumPrice = 0
    entrance = 10

    if len(self.priceList) > entrance:
        param = random.randint(4,10)
        i=2
        while i!= param+2:
            self.sumPrice = self.priceList[-i] + self.sumPrice
            i=i+1
        self.averagePrice = self.sumPrice / param

        print "Sum prices",self.sumPrice, "Everage prices", self.averagePrice

        if self.averagePrice < self.price:
            if self.randomProbability > 0.2:
                self.buyPrice = self.price * self.coefficient
                print "agent.Trend=",self.number,"Buy price","%.2f" % self.buyPrice
                self.aBook.setBuyDecision([self.buyPrice,self])
        if self.averagePrice > self.price:
            if self.randomProbability > 0.2:
                self.sellPrice = self.price / self.coefficient
                print "agent.Trend=",self.number,"SellPrice","%.2f" % self.sellPrice
                self.aBook.setSellDecision([self.sellPrice,self])
```

Figura 3 Modulo "trendTransaction" tratto dal file TrendAgents.py

4.3 File: VolumeAgents.py

"VolumeAgent" è una particolare categoria di agenti, che non considerano l'andamento dei prezzi di mercato, bensì i volumi di acquisto o di vendita ad ogni determinato istante. Essi agiscono spinti dall'andamento dei mercati verso correnti rialziste o ribassiste.

La classe "*VolumeAgent*" attraverso la variabile "volSell" e "volBuy", richiamate all'interno del Book, è a conoscenza in ogni istante della differenza tra i volumi di acquisto e quelli di vendita.

Nel caso in cui la variabile che determina il numero delle offerte di acquisto sia superiore alla variabile che determina il numero delle offerte di vendita, l'agente in questione è maggiormente propenso a seguire il mercato e a proporre un'offerta di acquisto. Viceversa, nel caso in cui il numero delle offerte di vendita sia superiore al numero di offerte di acquisto, l'agente è maggiormente propenso ad avanzare un'offerta di acquisto.

```
class VolumeAgent:
    def __init__(self,number,priceList,aBook):
        self.number = number
        self.assets = 0
        self.type = "VolumeAgent"
        self.aBook = aBook
        self.priceList = priceList

    def volTransaction(self):
        self.randomProbability = random.random()
        self.price = self.priceList[-1]
        self.sellPrice = 0
        self.coefficient = random.uniform(1.0,1.2)
        self.buyPrice = 0
        gapVol = random.randint(2,6)

        if (self.aBook.volSell + self.aBook.volBuy) > gapVol:
            if self.randomProbability > 0.4:
                self.buyPrice = self.price * self.coefficient
                print "agent.Vol=",self.number,"Buy price","%.2f" % self.buyPrice
                self.aBook.setBuyDecision([self.buyPrice,self])
        else:
            if self.randomProbability > 0.4:
                self.sellPrice = self.price / self.coefficient
                print "agent.Vol=",self.number,"SellPrice","%.2f" % self.sellPrice
                self.aBook.setSellDecision([self.sellPrice,self])
```

Figura 4 Classe "VolumeAgent" tratta dall'omonimo file

4.4 BestOfferAgents.py

Il programma visibile in Figura 5 determina il comportamento dell'agente, che propone offerte vincenti ad ogni sua azione. Il differenziale tra i volumi di vendita e i volumi di acquisto fornisce all'agente l'input per operare. Avendo inserito una classe che non valuta i prezzi di mercato, bensì i volumi, è molto probabile che si creino correnti di agenti che propongono solamente offerte di acquisto e correnti di agenti che propongono solamente offerte di vendita. La categoria degli agenti *"BestOfferAgents"* smorza questa possibilità.

La classe *"BestOfferAgent"* ingloba la classe *"VolumeAgent"*, acquisendo, così, i metodi propri di quest'ultima, nonchè i nuovi moduli programmati.

```
class BestOffertAgent(VolumeAgents.VolumeAgent):
    def __init__(self,number,priceList,aBook):

        VolumeAgents.VolumeAgent.__init__(self,number,priceList,aBook)

        self.number = number
        self.assets = 0
        self.type = "BestOffertAgent"
        self.aBook = aBook
        self.priceList = priceList

    def uniformTransaction(self):
        self.sellPrice = 0
        self.buyPrice = 0
        gapVol = random.randint(4,8)

        if (self.aBook.volSell + self.aBook.volBuy) > gapVol:
            if self.aBook.buyList > 0:
                self.sellPrice = self.aBook.getMaxPriceBuyList() + 0.1
                print "agent.BestOffert=",self.number,"SellPrice","%.2f" % self.sellPrice
                self.aBook.setSellDecision([self.sellPrice,self])

        if (self.aBook.volSell + self.aBook.volBuy) < -gapVol:
            if self.aBook.sellList > 0:
                self.buyPrice = self.aBook.getMaxPriceSellList() + 0.1
                print "agent.BestOffert=",self.number,"SellPrice","%.2f" % self.buyPrice
                self.aBook.setBuyDecision([self.buyPrice,self])
```

Figura 5 Classe "BestOffertAgent" tratta dall'omonimo file

4.5 Model.py

In Figura 6 è possibile notare il particolare della classe *"Model"* che permette di individuare, all'interno della lista *"agentActionList"*, le diverse tipologie di agenti ed i rispettivi comandi richiamati.

```
self.actionGroup2 = ActionGroup.ActionGroup ("bid_ask")
def do2(adress):
    adress.agentActionListCopy = self.agentActionList[:]
    for i in adress.agentActionListCopy:
        if i.type == "RandomAgent":
            i.randomTransaction()
        elif i.type == "VolumeAgent":
            i.volTransaction()
        elif i.type == "BestOffertAgent":
            i.bestTransaction()
        else:
            i.trendTransaction()
    #Tools.askEachAgentIn(adress.agentActionListCopy,Agents.Agent.random
self.actionGroup2.do = do2
```

Figura 6 Modulo per il riconoscimento dei diversi agenti e per il richiamo dei rispettivi comandi, tratto da "Model.py"

Manuale Utente Parte 5

FILES: LevelPriceRealDataAgent.py , VariationPriceRealDataAgent.py, BSCalculation.py, CoveredAgents.py

Le evoluzioni del programma presentate in questo capitolo interessano, rispetto alla versione precedente, i seguenti punti:

- creazione di una nuova classe di agenti, che baserà la propria strategia di acquisto o di vendita, sulle informazioni ricavate dai dati forniti da Borsa Italiana sull'indice Fste Italia All Share[30]. In questo caso, l'agente agirà in base ai livelli di prezzo raggiunti dal prezzo simulato, confrontato con i livelli raggiunti dal Ftse
- creazione di una nuova classe di agenti, che baserà la propria strategia di negoziazione sui medesimi dati utilizzati dalla classe precedente, da cui, tuttavia, ricaverà le variazioni percentuali di valore e non i livelli di valore.

[30] Per informazioni riguardo ai dati, che compongono l'indice e la metodologia di calcolo dello stesso, fare riferimento alla sezione teorica di questa tesi.

- creazione di un file in cui saranno programmati i moduli, utilizzati dalla classe *Covered*, di cui parlerò più tardi, preposti al conteggio del prezzo delle opzioni *call* e *put,* secondo la formula di Black and Scholes
- creazione di una classe Covered, la quale rappresenta agenti che operano applicando la strategia di copertura della posizione attraverso la vendita di opzioni *call* e *put.* Caratteristica di questa classe è la capacità, non solo di ricavare dati da funzioni esterne riguardanti i prezzi delle opzioni, ma anche di calcolare i rendimenti delle posizioni che ogni agente ha assunto sul mercato.

5.1 File: LevelPriceRealDataAgents.py

Le azioni, compiute da questa particolare categoria di agenti, si basano sulla capacità di presentare un'offerta di acquisto o di vendita, confrontando il prezzo dell'ultima transazione avvenuta, interna allo step, e il prezzo registrato dall'indice Ftse all Share al minuto successivo (step successivo). Questo agente è programmato come se conoscesse, con alcuni secondi di anticipo, il livello che il prezzo avrà, o che dovrebbe avere, secondo un corretto *pricing* nell'immediato futuro. Nel caso in cui il mercato sopravalutasse il titolo, l'agente presenterà un'offerta di vendita in linea con il livello registrato dal Ftse. In caso contrario, egli presenterà un'offerta di acquisto in linea con il livello espresso dal Ftse nell'immediato futuro. La caratteristica di questo agente è che egli presenterà offerte in linea con i livelli raggiunti dal Ftse all share. La motivazione per cui si è voluto programmare questa classe è il tentativo di far assumere alle serie di prezzi simulati delle distribuzioni, che si avvicinino a distribuzioni di prezzo reale, rilevate dall'indice Ftse nella settimana di riferimento dal 13/12/2011 al 17/12/2011.

Il codice in Figura 1 presenta la funzione che permette la lettura dei dati, in questo caso i valori del Ftse, memorizzati su file di testo esterno. Inoltre, il dato dopo essere stato letto, viene copiato all'interno della lista *self.values.*

```
self.values=[]
with open("FTSE_ITALIA_ALL_SHARE.txt") as f:
    for line in f:
        dato=(line.split(';')[3])
        #print dato
        self.values.append(float(dato))
```

Figura 1 Funzione che permette la lettura di dati da file esterno *txt*

Il codice in Figura 2 presenta il cuore della classe level. Questa funzione determina, in base alle condizioni di mercato, l'invio dell'offerta di acquisto o di vendita al Book. La variabile j, la quale funge da perfetto contatore, seleziona il valore reale *t,* corrispondente al valore in simulazione. In pratica, il valore j allinea, dal punto di vista temporale, la simulazione alla realtà. Il conteggio avviene in base alla variabile step, la quale è incrementata ad ogni step e moltiplicata per la variabile *self.passo*. Attraverso questo meccanismo, nel caso in cui un utente esterno inserisca un numero di step pari a 50, il valore reale avrà un passo di 10 cioè, un dato ogni 10 minuti, dal momento che la somma di dati per giornata è pari a 500 (un dato al minuto). Questo meccanismo permette un'elevata flessibilità di decisione di step all'utilizzatore esterno. Un problema da considerare riguarda il fatto che, nel caso in cui sia inserito un numero di step che rende decimale il valore della variabile *self.step*, il programma darà segnale di errore per incapacità di riconoscere quale valore individuare in corrispondenza di j.

```
def levelRealPriceTransaction(self):
    nstep = self.nSteps
    self.passo = 500 / nstep
#   print "self.passo", self.passo
    self.randomValue=random.random()
    self.price=self.priceList[-1]
    self.sellPrice=0
    self.buyPrice=0
   # self.finalPriceForStepList = self.aBook.getFinalPriceForStep()
    j = 0

    if len(self.priceList)>0:
        b =0
        j = self.step*self.passo
        #simulatorPriceLevel= self.finalPriceForStepList[-1]
        realDataLevel = self.values[j]
#       print "realDataLevel",realDataLevel,"J",j,"self.step",self.step

        if realDataLevel > self.price:
            self.buyPrice = realDataLevel
#           print "realDataLevel",realDataLevel
#           print "agent.LevelPriceRealData=",self.number, "Buy price","%.2f" % self.buyPrice
            self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataLevel < self.price:
            self.sellPrice = realDataLevel
#           print "realDataLevel",realDataLevel
#           print "agent.LevelPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
            self.aBook.setSellDecision([self.sellPrice,self,self.type])
```

Figura 2 Codice relativo alla funzione levelRealPriceTransaction

Altre funzioni programmate in questa classe riguardano il conteggio di azioni possedute per singolo agente. Tale codice non sarà riportato perché già spiegato in parti precedenti del Manuale utente.

5.2 File: VariationPriceRealDataAgents.py

Di questa particolare categoria di agenti ne sono state programmate tre versioni differenti. In questa sezione riprenderemo il codice della versione maggiormente utilizzata in simulazione, la quale sarà posta in competizione con altre classi per lo studio sui fenomeni di interazione. La strategia di mercato che un agente *variation* mette in atto si basa sulla sua conoscenza di quelle che saranno le future variazioni dell'indice, con alcuni secondi di anticipo rispetto agli altri agenti. La sua convinzione è che le variazioni dell'indice bene approssimano le variazioni del prezzo del titolo, oggetto di transazione. La strategia applicata da questa categoria si basa sul confronto, al tempo t, tra la variazione di valore dell'ultimo contratto concluso e il contratto concluso all'inizio dello step e sul calcolo della variazione percentuale. Per esempio, se il tempo t è pari a 9:30:25 sec, il valore del prezzo simulato all'inizio dello step è quello

in corrispondenza dell'istante 9:30:00 sec. Vengono, quindi, calcolate le variazioni percentuali tra i prezzi registrati in corrispondenza dei tempi appena enunciati.
Allo stesso modo vengono calcolate le variazioni percentuali registrate dal valore del Ftse in corrispondenza dei seguenti tempi: 9:30:00 e 9:31:00.
I prezzi offerti avranno come base di partenza il prezzo dell'ultima transazione e saranno moltiplicati per lo spread tra variazione di prezzo simulato e variazione di valore reale.
In base alle variazioni registrate vengono fatti tutti i possibili confronti che un investitore razionale farebbe prima di prendere una posizione sul mercato in vendita oppure in acquisto.
In Figura 3 è possibile visualizzare il codice di programmazione della classe *variation*. Di seguito, indicherò in generale le situazioni che saranno analizzate dall'agente e rappresentate nel codice dalle varie condizioni "*if*", le quali, con un semplice ragionamento economico, possono essere ricondotte a strategie di trading sul mercato.
L'agente *variation* proporrà un'offerta di vendita quando:

- variazione del valore Ftse < variazione prezzo simulato.

L'offerta proposta sarà pari al prezzo dell'ultima transazione, diminuito di un valore percentuale pari allo scostamento tra valore percentuale reale e valore percentuale simulato.
L'agente *variation* proporrà un'offerta di acquisto quando:

- variazione del valore Ftse > variazione prezzo simulato.

L'offerta proposta sarà pari al prezzo dell'ultima transazione aumentato di un valore percentuale pari allo scostamento tra valore percentuale reale e valore percentuale simulato.
Nel codice è possibile osservare un numero elevato di condizioni, riconducibili ai due principi appena citati, che risolvono tutte le possibilità che si possono presentare in simulazione.

```
def variationRealPriceTransaction(self):
    self.nstep = self.nSteps
    self.passo = 500 / self.nstep
    self.sellPrice=0
    self.buyPrice=0
    b=0
    j=0

    self.finalPriceForStepList = self.aBook.getFinalPriceForStep()

    if len(self.finalPriceForStepList) > 1:
        self.price=self.priceList[-1]
        j = self.step*self.passo
        b = (self.step - 1)*self.passo
        realDataVariation = ((self.values[j]-self.values[b])/ self.values[b])
        simulatorPriceVariation= ((self.price - self.finalPriceForStepList[-1])/self.finalPriceForStepList[-1])
      #  print "REAL DATA VARIATION ",realDataVariation,"SIMULATOR Price Variation",simulatorPriceVariation

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                if simulatorPriceVariation > 0:
                    self.sellPrice = self.priceList[-1]*(1 +(realDataVariation+ simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                if simulatorPriceVariation < 0:
                    self.sellPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])

        if realDataVariation > simulatorPriceVariation:
            if realDataVariation > 0:
                if simulatorPriceVariation < 0:
                    self.buyPrice = self.priceList[-1]*(1 + (realDataVariation+simulatorPriceVariation))
           #         print "agent.VariationPriceRealData=",self.number, "Buy price","%.2f" % self.buyPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])
```

```
        if realDataVariation > simulatorPriceVariation:
            if simulatorPriceVariation > 0:
                if realDataVariation > 0:
                    self.buyPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation > simulatorPriceVariation:
            if simulatorPriceVariation < 0:
                if realDataVariation < 0:
                    self.buyPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation > 0:
                if simulatorPriceVariation > 0:
                    self.sellPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])
```

Figura 3 Codice relativo alla funzione variationRealPriceTransaction

5.3 File: CoveredAgents.py

L'agente *covered,* facente capo alla classe che approfondirò in questo paragrafo, basa la propria strategia di operatività sull'acquisizione di una posizione sul mercato,

ricoperta immediatamente attraverso la vendita di un'opzione. Le caratteristiche computazionali, di cui questa classe è dotata, sono molto avanzate, perché essa oltre a possedere una strategia di decisione di acquisto o di vendita di un titolo, è programmata sia in modo da poter inviare e acquisire dati riguardanti il prezzo di opzioni, calcolate mediante Black and Scholes, sia per permettere il calcolo del rendimento della strategia, attuata dall'agente i-esimo.

In Figura 4 è rappresentato il meccanismo di decisione, che guida un agente *covered* nell'acquisto del titolo con la conseguente copertura della posizione attraverso la vendita di un'opzione *call* o *put*. Entrambe vengono decise dopo aver identificato la situazione di mercato al tempo *t.* In pratica, ogni agente, in base ad una funzione casuale, decide quanti prezzi di transazione assumere per calcolare una media su valori storici, la quale verrà poi confrontata con il prezzo dell'ultima transazione avvenuta. Nel caso in cui la media dei prezzi storici sia inferiore al prezzo dell'ultima transazione al tempo *t-1,* si delineata una situazione di mercato in crescita. Per questa ragione, la posizione viene coperta attraverso la vendita di un'opzione *call,* OTM (*out of the money*). Nel caso opposto, in cui si prevede un andamento di mercato in decrescita, l'agente vende un'opzione *put* OTM. La vendita delle opzioni viene fatta *out of the money* così da poter guadagnare, nel caso in cui l'acquirente eserciti l'opzione a scadenza, sia dalla vendita dell'opzione, sia dallo spread tra prezzo di acquisto del titolo e prezzo di vendita del titolo.

La variabile, che definisce l'opzione OTM, è identificabile nel codice con la dicitura *self.strike*. La variabile strike, unita alla variabile tempo, il prezzo spot, la deviazione standard e il rendimento *risk free* rappresentano i dati necessari da calcolare ogni qual volta un agente *covered* sia chiamato ad operare; i seguenti dati, quindi, sono inviati al file in cui sono raccolti i metodi necessari per il calcolo del prezzo delle opzioni, secondo la formula di Black and Scholes.

```
def coveredTransaction(self):
    self.randomValue=random.random()
    self.price=self.priceList[-1]
    #self.sellPrice=0
    self.buyPrice=0
    self.BsCall = 0
    self.BsPut = 0
    self.averagePrice = 0
    self.sumPrice = 0
    entrance = 10
    self.will = random.random()

    if len(self.priceList) > entrance:
        param = random.randint(4,len(self.priceList)-2)
        i=2
        while i!= param + 2:
            self.sumPrice = self.priceList[-i] + self.sumPrice
            i=i+1
        self.averagePrice = self.sumPrice / param

         # print "self.numberAgent",self.numberAgent
          if self.flag == 0:
              if self.will < 0.4:
                  if self.averagePrice < self.price:
                  #    print "EVERAGE-PRICE",self.averagePrice,"SELF.PRICE",self.price
                      self.sigma = numpy.std([self.priceList])
                      self.S0 = self.priceList[-1]
                      self.K = self.S0 + random.randint(0,+10)
                      self.r = 0.03
                      #self.T = 0.5
                      #self.sigma = 0.1
                      self.T = self.numCycles / 365
                      self.flag = "CALL"
                      self.chooseCall=1
                      self.BsCall = BsCalculation.BlackScholesEuropeanCallPrice(self.S0,self.K,self.r,self.sigma,self.T)
                   #   print "CALL","self.S0",self.S0,"self.K",self.K,"self.r",self.r,"self.sigma",self.sigma,"self.T",self.T
                      self.buyPrice= self.price + random.random()
                      self.buyPriceCall = self.buyPrice
                      self.KCall = self.K
                   #   print "agent.Covered=",self.number, "Buy price","%.2f" % self.buyPrice
                   #   print "self.BsCall", self.BsCall, "self.K", self.K
                      self.aBook.setBuyDecision([self.buyPrice,self,self.type])

                  if self.averagePrice > self.price:
                    #  print "EVERAGE-PRICE",self.averagePrice,"SELF.PRICE",self.price
                      self.sigma = numpy.std([self.priceList])
                      self.S0 = self.priceList[-1]
                      self.K = self.S0 + random.randint(0,+10)
                      self.r = 0.03
                      #self.T = 0.5
                      #self.sigma = 0.1
                      self.T = self.numCycles / 365
                      self.flag = "PUT"
                      self.BsPut = BsCalculation.BlackScholesEuropeanPutPrice(self.S0,self.K,self.r,self.sigma,self.T)
                   #   print "PUT","self.S0",self.S0,"self.K",self.K,"self.r",self.r,"self.sigma",self.sigma,"self.T",self.T
                      self.buyPrice= self.price + random.random()
                      self.buyPricePut = self.buyPrice
                      self.KPut = self.K
                   #   print "agent.Covered=",self.number, "Buy price","%.2f" % self.buyPrice
                   #   print "self.BsPut",self.BsPut,"self.K", self.K
                      self.aBook.setBuyDecision([self.buyPrice,self,self.type])
```

Figura 4 Codice relativo alla funzione coveredTransaction

In Figura 5 è visibile il codice del modulo, che permette il calcolo del rendimento della posizione assunta da un agente i-esimo *covered.*

Le prime linee di codice sono state scritte per consentire al programma di calcolare il rendimento della posizione assunta dall'agente, in corrispondenza dell'ultimo *step* dell'ultimo ciclo.

In corrispondenza dello *step* in cui è calcolato il rendimento, ho volutamente impostato il programma come se la posizione venisse chiusa, cioè al prezzo di mercato

l'agente vende il titolo che ha in portafoglio. Il calcolo del rendimento è completo e considera sia la possibilità che l'opzione sia esercitata sia la possibilità che non sia esercitata, condizione specificata all'utilizzatore grazie a messaggi di output.

```
def numberStep(self):
    if self.buyPriceCall > 0:
        self.buyOffertListCall.append(self.buyPriceCall)
    else:
        self.buyOffertListCall.append(self.priceList[0])
        #self.KCallList.append(self.KCall)
    if self.buyPricePut > 0:
        self.buyOffertListPut.append(self.buyPricePut)
    else:
        self.buyOffertListPut.append(self.priceList[0])
        #self.KPutList.append(self.KPut)
    self.returnStrategy=0
    self.active +=1
    valueStrategy = 0
    self.numberSteps+=1
    self.numberAgent +=1
    if self.numberSteps == self.nSteps:
        self.numberCycles +=1
        self.numberAgent = 0
        if self.numberCycles == self.numCycles:
            if self.numberSteps == self.nSteps:
                if self.flag == "CALL":
                    if self.priceList[-1] > self.KCall:
                        if self.buyPriceCall > self.KCall:
                            valueStrategy = (self.cash + self.BsCall - self.buyPriceCall + (self.buyPriceCall - self.KCall))
                            self.returnStrategy = (valueStrategy - self.cash)/self.cash
                         #   print "returnStrategy","agent.Covered=",self.number,"EsercitaCall", self.returnStrategy,"self.KCall"
                            self.flag = 0
                        else:
                            valueStrategy = (self.cash + self.BsCall - self.buyPriceCall - (self.KCall - self.buyPriceCall))
                            self.returnStrategy = (valueStrategy - self.cash)/self.cash
                          #  print "returnStrategy","agent.Covered=",self.number,"EsercitaCall", self.returnStrategy,"self.KCall"
                            self.flag = 0
                    if self.priceList[-1] < self.KCall:
                        valueStrategy = (self.cash + self.BsCall - self.buyPriceCall + self.priceList[-1])
                        self.returnStrategy = (valueStrategy - self.cash)/self.cash
                     #   print "returnStrategy","agent.Covered=",self.number,"NonEsercitaCall", self.returnStrategy,"self.KCall"
                        self.flag = 0
                if self.flag == "PUT":
                    if self.priceList[-1] >  self.KPut:
                        valueStrategy=(self.cash + self.BsPut - self.buyPricePut + self.priceList[-1])
                        self.returnStrategy=(valueStrategy - self.cash)/self.cash
                     #   print "returnStrategy","agent.Covered=",self.number,"NonEsercitaPut", self.returnStrategy,"self.KPut"
                        self.flag = 0
                    if self.priceList[-1] < self.KPut:
                        if self.buyPricePut > self.KPut:
                            valueStrategy= self.cash + self.BsPut - self.buyPricePut - (self.buyPricePut - self.KPut)
                            self.returnStrategy=((valueStrategy - self.cash)/self.cash)
                         #   print "returnStrategy","agent.Covered=",self.number,"EsercitaPut", self.returnStrategy,"self.KPut"
                            self.flag = 0
                        else:
                            valueStrategy= self.cash + self.BsPut - self.buyPricePut + (self.KPut - self.buyPricePut)
                            self.returnStrategy=((valueStrategy - self.cash)/self.cash)
                         #   print "returnStrategy","agent.Covered=",self.number,"EsercitaPut", self.returnStrategy,"self.KPut"
                            self.flag = 0
```

Figura 5 codice relativo al calcolo dei rendimenti delle posizioni assunte dall'agente i-esimo

5.4 File: BsCalculation.py

Il file presentato in questo paragrafo, a differenza dei precedenti, non rappresenta una classe. Vi sono una serie di moduli che, utilizzati dalla classe *covered,* permettono il calcolo del prezzo delle opzioni secondo la formula di Black and Scoles. Per formula di Black and Scholes si intende l'espressione per il prezzo di non arbitraggio di un'opzione call (put) di tipo europeo, ottenuta sulla base del modello Black-Merton-Scholes.

La formula per il calcolo del prezzo di un'opzione *call* valutata in *t* con scadenza in *T* è data da:

$$C(S,t) = S_t N(d_1) - Ke^{-r(T-t)} N(d_2)$$

Per l'opzione put la formula corrispondente è la seguente:

$$P(S,t) = Ke^{-r(T-t)} N(-d_2) - S_t N(-d_1)$$

Per completezza e comprensione dei calcoli che saranno visibili nel codice di programmazione, di seguito sono inserite le formule per i calcoli rispettivamente di *d1* e *d2*.

$$d_1 = \frac{\ln \frac{S_t}{K} + \left(r + \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}; \quad d_2 = d_1 - \sigma\sqrt{T-t}$$

Le variabili inserite in queste formule sono calcolate dalla classe *covered* e inviate ai moduli visibili in Figura 6 per il calcolo dei prezzi. Esse rispettivamente si riferiscono a:

- S_t è il prezzo del sottostante
- K è lo strike e rappresenta il prezzo di esercizio dell'opzione
- r è il tasso di interesse privo di rischio su base annua
- N(.) denota la funzione di ripartizione di una variabile casuale normale
- σ è la deviazione standard del prezzo del titolo

Un problema affrontato e risolto riguardava il calcolo dela funzione di ripartizione di una variabile casuale normale di cui Python non dispone. In verità, si potrebbe semplicemente installare l'interfaccia *Scipy* e attraverso Python richiamare la funzione matematica che consente il calcolo. Personalmente ho avuto alcuni problemi di compatibilità tra la versione Python di cui disponevo, R e Scipy. Per questa ragione ho trovato una funzione attraverso un blog che con l'inserimento di alcune costanti mi permette di calcolare la funzione di ripartizione di una variabile casuale normale con

una buona approssimazione come visibile in Figura 6. Per poter fare i calcoli sopra citati è necessario installare e richiamare attraverso Python, Numpy, pacchetto informatico che consente di programmare e calcolare funzioni matematiche.

```
import math
import numpy

def erfcc(x):
    z = abs(x)
    t = 1. / (1. + 0.5*z)
    r = t * math.exp(-z*z-1.26551223+t*(1.00002368+t*(.37409196+
        t*(.09678418+t*(-.18628806+t*(.27886807+
        t*(-1.13520398+t*(1.48851587+t*(-.82215223+
        t*.17087277)))))))))
    if (x >= 0.):
        return r
    else:
        return 2. - r

def ncdf(x):
    return 1. - 0.5*erfcc(x/(2**0.5))


def d1(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return (math.log(S01/K1) + (r + sigma**2 / 2) * T) / (sigma * math.sqrt(T))


def d2(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return (math.log(S01/K1) + (r - sigma**2 / 2) * T) / (sigma * math.sqrt(T))

def BlackScholesEuropeanCallPrice(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return S0 * ncdf(d1(S0, K, r, sigma, T)) - K * math.exp(-r * T) * ncdf(d2(S0, K, r, sigma, T))

def BlackScholesEuropeanPutPrice(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return K * math.exp(-r * T) * ncdf(-d2(S0, K, r, sigma, T)) - S0 * ncdf(-d1(S0, K, r, sigma, T))
```

Figura 6 Funzioni necessarie per il calcolo del prezzo di opzioni call e put secondo Black and Scholes

Capitolo 7
SIMULAZIONI

Dallo studio prodotto sulle distribuzioni dei prezzi nei mercati finanziari, dall'osservanza delle dinamiche di boom e crash di borsa e dalla produzione di modelli, che si fondano sulla metodologia ABM, è scaturito un approfondito lavoro di programmazione di un simulatore di borsa per l'analisi quantitativa, sia degli andamenti del prezzo di un particolare titolo, sia dei rendimenti di particolari strategie, mediante opzioni.

La grande domanda a cui si vuole dare risposta è: «come il mercato reagisce all'interazione di agenti programmati con diverse strategie di investimento?».

La conduzione di un attento procedimento di verifica dell'effettivo funzionamento del simulatore si fonda su una serie di esperimenti ben organizzati, che hanno l'obiettivo di osservare particolari eventi e correlazioni, che emergono dalle interazioni degli agenti.

Ciò che può essere anticipato, invitando il lettore ad approfondire la lettura delle sperimentazioni e del codice di programmazione, è il sorprendente risultato raggiunto mediante la programmazione del simulatore e relativo alla ricreazione della realtà

attraverso la simulazione. Con questa affermazione si vuole dare risalto al fatto che, le distribuzioni di valori osservate nel mercato di borsa reale, non sono state meramente inserite all'interno del programma, bensì sono state rigenerate dall'interazione degli agenti.

Nel corso del capitolo, saranno analizzati specifici test sul funzionamento del simulatore e saranno fatte congetture sui fenomeni osservati. Inoltre, si cercherà di motivare, attraverso l'analisi di particolari variabili, le cause che hanno determinato lo sviluppo di particolari eventi.

7.1 ORGANIZZAZIONE DEGLI ESPERIMENTI

Gli esperimenti saranno così organizzati e descritti:

1- breve riepilogo delle caratteristiche degli agenti che popolano il mercato
2- spiegazione degli effetti attesi dall'interazione tra agenti, prima della realizzazione di qualsiasi esperimento
3- presentazione di grafici e commenti generali sul risultato dell'esperimento
4- approfondimento dei risultati ottenuti, cercando di spiegare il fenomeno attraverso lo studio di particolari variabili
5- giudizio finale sui fenomeni osservati dai test

Trattando il discorso delle variabili, prese come riferimento per analizzare i fenomeni osservati, ci soffermeremo su:

A. prezzi di transazione del titolo, frutto dell'incontro tra domanda ed offerta. I prezzi visibili nei grafici non corrisponderanno a tutti i prezzi di transazione, avvenuti in simulazione, bensì solo all'ultimo prezzo di ciascuno *step*. Questa approssimazione è necessaria per avere un'idea sulle distribuzioni dei prezzi, evitando di avere grafici poco chiari, dato l'elevato numero di *step* inserito
B. lunghezza delle liste per *step*. Per lunghezza di lista si intende il numero di offerte, rilevate al termine di ogni *step*, non ancora passate ad eseguito ed inviate al *Book* di borsa. Si vuole sottolineare questo aspetto perché, in Terna[2002], si è osservato come gran parte delle bolle e dei crash di prezzo

fossero conseguenza di uno sbilanciamento nel numero di offerte, presenti in uno dei due lati del *Book*

C. identità e numero di agenti che, per classe, hanno determinato il prezzo di contrattazione del titolo in un determinato *step*. E' molto utile osservare questa variabile in associazione ad un altro dato rilevato, quello degli spread, che sarà presentato nel passaggio seguente. Inoltre, un'altro aspetto utile, evidenziato da questa variabile, è l'influenza che una certa classe ha sul trend di mercato
D. *spread* tra migliore e peggiore prezzo in ciascun lato del *Book.* Come già detto, è utile osservare tale variabile in associazione alla variabile spiegata al punto C, per comprendere, in caso di forti squilibri registrati dallo spread, l'identità dell'agente che ha causato il brusco cambiamento di prezzo.

7.2 CONSIDERARE GLI ERRORI IMPERCETTIBILI

Considerando, in base ai controlli e ai test di funzionamento eseguiti, il modello creato come corretto, è, tuttavia, doveroso presentare alcune ipotesi dubitative sui risultati ottenuti.

Come affermato da Axtell L. e Epstein.M [1994], per comprendere come il simulatore opera, è necessario fare alcune analisi critiche, cercando di rispondere a questa domanda: «Is my program doing what I want it to?». Le analisi critiche riguardano, in particolare, la possibilità che ci siano impercettibili errori di funzionamento nel programma o nel calcolatore che, durante le conclusioni, devono essere assunti come possibili. Nel *paper* a cui sto facendo riferimento, sono riportati alcuni casi particolari che potrebbero causare errori nelle simulazioni.

La principale causa che non permette ad un osservatore di individuare possibili errori nei risultati, è che negli agent-based model si valutano i fenomeni macro emergenti dalle interazioni. Possibili "bugs", quindi, possono nascondersi nella struttura degli agenti semplici e causare una strana combinazione di perfetto funzionamento della simulazione, pur considerando dati errati.

Alcuni agenti, ad esempio, per questioni legate all'indicizzazione delle liste, possono presentare degli errori nella trascrizione o nella sovrascrittura dei dati. Esempio più grave del precedente, è la presenza di errori nel sistema operativo, relativi al processo di gestione della memoria; questo potrebbe non apparire come distorsione dei risultati.
Altro esempio è la possibilità che i modelli agent-based mostrino relativa dipendenza verso particolari caratteristiche di alcuni agenti. Come dimostrato da test svolti dai due studiosi, nel loro modello "Sugarscape" alcuni risultati ottenuti sono fortemente condizionati da particolari agenti operanti, i quali, se rimossi nel corso della simulazione, creano risultati contrari alla logica di evoluzione della società, creata nel modello.

7.3 TEST NUMERO 1: MERCATO POPOLATO DA SOLI AGENTI "ZERO INTELLIGENCE"

Premessa

Questo primo test è stato fatto per accertare il corretto funzionamento del simulatore. Obiettivo della mia tesi non è quello di osservare il mercato popolato da soli agenti *random*, per questa ragione poco tempo sarà dedicato al commento di questo grafico.
In un mondo popolato di soli agenti *zero intelligence* ci aspetteremo di osservare una distribuzione dei prezzi casuale, senza alcun orientamento logico. Nel caso in cui non si osservasse questo effetto, dovremmo porci alcuni dubbi riguardo al funzionamento del sistema oppure delle funzioni di prezzo programmate per gli agenti.

Spiegazione dei risultati ottenuti

Osservando il grafico di distribuzione dei prezzi, vediamo un perfetto *random walk*, in cui è possibile notare una serie di boom e crash dell'ordine di alcune centinaia di punti. Considerando che l'offerta fatta da un agente è una variabile casuale che può

assumere un valore massimo, pari a 3 punti indice, l'andamento registrato è a tutti gli effetti caratterizzato da picchi di prezzo positivo e picchi di prezzo negativo.

Il test condotto è stato programmato in questo modo:

- Numero di agenti *random*: 100
- Numero di *step* per giornata: 500 (uno per ogni minuto, considerando le 8 ore di contrattazione)
- Numero di giornate (cicli) simulati: 5 (equivalente ad una settimana di contrattazione)

Durante tutti i test da ma presentati, il numero di cicli e il numero di step rimane costante. La prima motivazione nasce dal fatto che, per presentare delle congetture sul comportamento dei mercati, ci si deve basare su dati consistente. La seconda motivazione è che il numero di step e di cicli non può superare l'ammontare indicato, perché alcuni agenti sono programmati con un numero di dati fissi, su cui basano le proprie strategie di acquisto e vendita (classi Ftse). La terza motivazione è che, impostando test preliminari, si è registrato che i valori inseriti in input, appena citati, consentono un sufficiente numero di dati a disposizione.

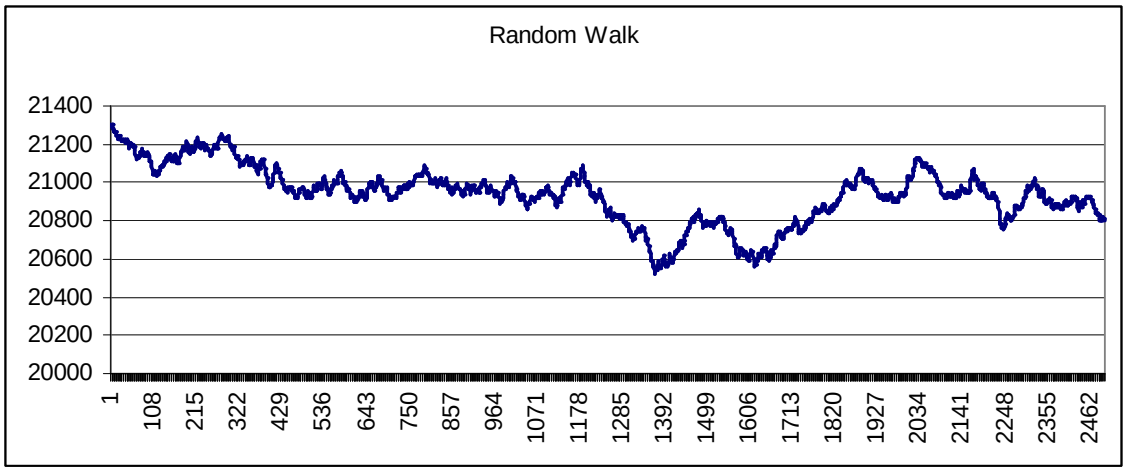


Figura 1 Serie di prezzi distribuiti come una "*random walk*" in un mercato popolato di soli agenti casuali

Giudizio conclusivo sperimentazioni

Come già anticipato, non mi soffermerò sull'analisi delle cause che hanno determinato le bolle e i crash di prezzo. Questa analisi sarà condotta per i futuri esperimenti.

Concludendo questo breve paragrafo, si può affermare che il programma funziona correttamente e gli agenti *random*, che saranno inseriti nella quasi totalità degli esperimenti, si comportano in maniera corretta.

7.4 TEST NUMERO 2: MERCATO POPOLATO DA AGENTI ZERO INTELLIGENCE E LEVEL PRICE REAL AGENT (LEGATI AL FTSE ALL SHARE)

Rappresentando il risultato generato dalle interazioni di queste due categorie di agenti si otterrà, come vedremo, uno dei grandi risultati ottenuti dalla programmazione del simulatore.

Procedendo per gradi, è opportuno riprendere le caratteristiche dei due agenti, che popolano il mercato virtuale:

- agente random: presenta offerte di acquisto e di vendita con casualità, offrendo un prezzo che avrà come base di partenza l'ultimo prezzo dell'ultima transazione virtuale avvenuta. Il prezzo presentato è programmato con una variabile casuale, che può aumentare la base di partenza, appena spiegata, al massimo di tre punti.
- agente level price real data: presenta un'offerta di acquisto o di vendita facendo un confronto tra il prezzo dell'ultima transazione avvenuta, interna allo step, e il prezzo registrato dall'indice Ftse all Share al minuto successivo (step successivo). Questo agente è programmato come se conoscesse, con alcuni secondi di anticipo, il livello che il prezzo avrà, o che dovrebbe avere, secondo un corretto *pricing* nell'immediato futuro. Nel caso in cui il mercato sopravalutasse il titolo, l'agente presenterà un'offerta di vendita in linea con il livello registrato dal Ftse. Viceversa, egli presenterà un'offerta di acquisto in linea con il livello espresso dal Ftse nell'immediato futuro.

Premessa

Programmando l'agente Ftse, ci si attendeva di poter dare alle distribuzioni del mercato virtuale alcune caratteristiche, o per lo meno, alcuni livelli di prezzo registrati dal Ftse All Share nella settimana tra il 13 Dicembre 2010 e il 17 Dicembre 2010. Questo avrebbe permesso di poter operare, attraverso strategie di mercato con opzioni, con un titolo virtuale inserito in un contesto di distribuzioni simile alla realtà. Anticipo già che il risultato raggiunto ha superato qualsiasi aspettativa.

Spiegazione dei risultati ottenuti

In Figura 2 è possibile osservare l'andamento del Ftse All Share, con frequenza "al minuto", nella settimana di contrattazione, citata in precedenza.

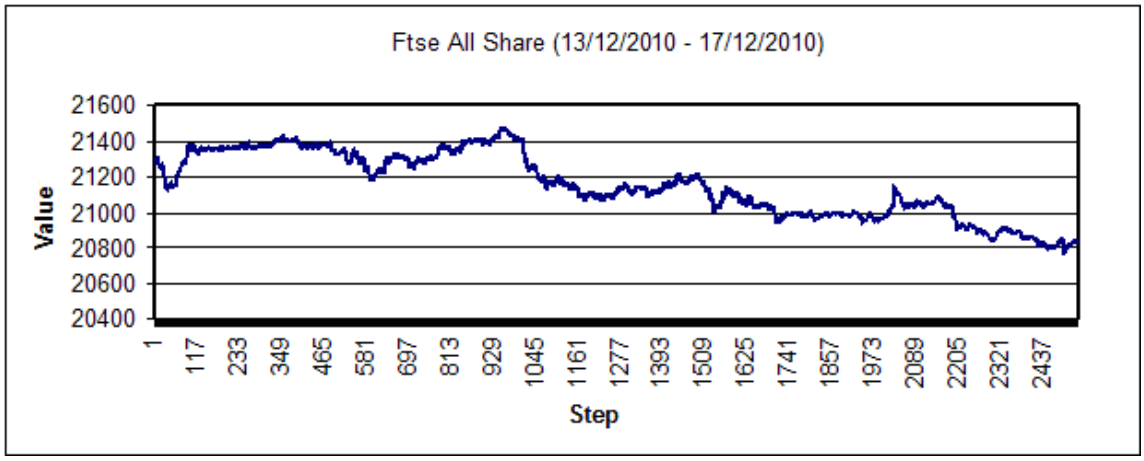


Figura 2 Andamento dell'indice Ftse All Share *(dati forniti da Borsa Italiana S.p.a)*

Come già accennato, il fine era di ricreare un andamento simile, anche solo di alcuni livelli, nel mercato virtuale.

In Figura 3 è possibile osservare l'andamento del prezzo, nel mercato simulato, in un mondo popolati da:

4. Numero agenti random: 100
5. Numero agenti level Ftse: 1
6. Numero di step per giornata : 500

7. Numero di cicli: 5

Figura 3 Serie di prezzi ottenuti da simulazione in un mondo popolato di 100 agenti random e 1 agente Level Ftse

Il primo risultato riscontrabile è che la distribuzione dei prezzi del titolo nel mercato simulato riprende l'andamento e i livelli di prezzo del mercato reale. L'aspetto che più colpisce è la composizione degli agenti del mercato simulato che ha riprodotto l'andamento del Ftse All Share. E' sufficiente un solo agente, che opera con una strategia di acquisto diversa dalla media della popolazione, per modificare tutto il mercato. Inoltre, dalle variabili registrate, che in seguito mostrerò, l'incidenza del solo agente Ftse, per transazioni a prezzi da lui precisamente indicati, è stata inferiore all'1%. Un'altro aspetto da sottolineare è che, ragionevolmente, il solo agente Fste non può concludere transazioni con se stesso, quindi, trova sempre nell'agente *random* una controparte che concluda transazioni, che permettono di ripetere i livelli del mercato reale.

Per evitare di osservare un numero troppo elevato di dati, porrò un frazionamento all'analisi, in modo tale da poter mostrare particolari aspetti legati alle cause dei cambiamenti di prezzo. L'esperimento che ho effettuato, programmando la classe, è mostrato in Figura 3; l'elevato numero di transazioni e di cicli ha reso possibile visualizzare l'effetto di interazione macro nella sua complessità e, quindi, la possibilità di avere un ottimo risultato di confronto con il Ftse reale, a cui la distribuzione faceva riferimento.

In Figura 4 è mostrato il particolare di 1000 dati Ftse, tratti dalla distribuzione originale.

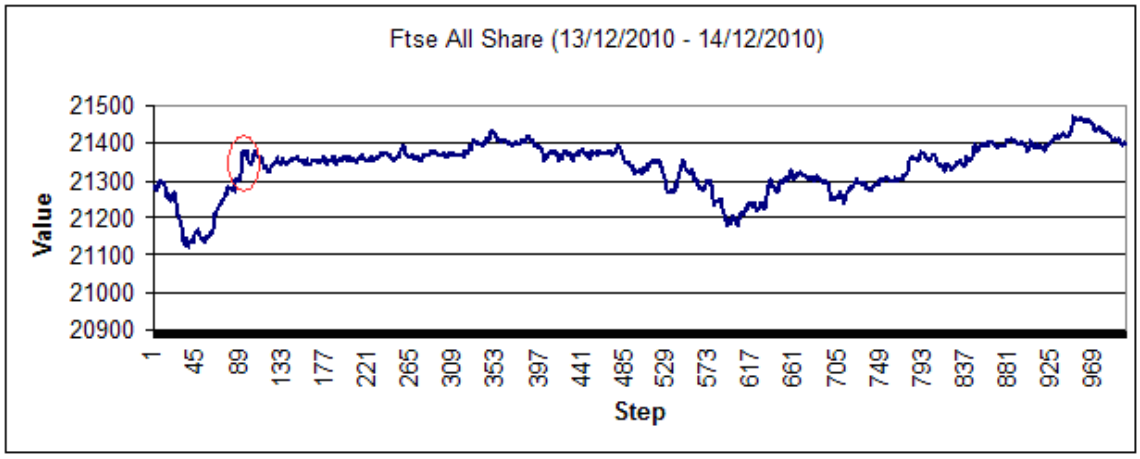


Figura 4 Particolare della distribuzione Ftse All Share relativa a 2 giorni di contrattazione

In Figura 5, invece, è presente il particolare dello stesso esperimento fatto in precedenza rappresentativo dei primi 1000 valori, come in Figura 4.

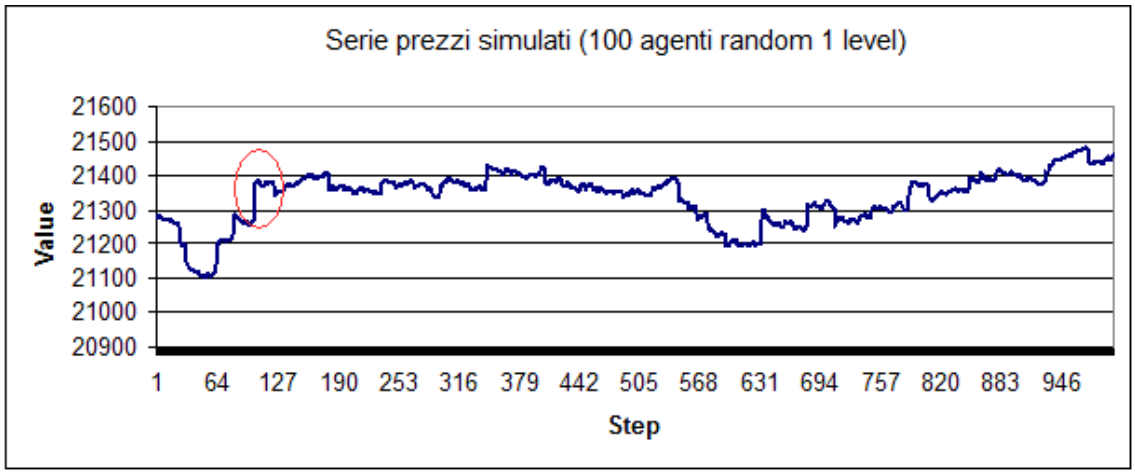


Figura 5 Particolare della serie di prezzi nel mercato simulato relativi a 2 giorni di contrattazione

Impostando con meno dati la creazioni dei grafici, la visualizzazione delle distribuzioni risulta più semplice.

Il primo test, a cui propongo di porgere attenzione, riguarda l'analisi delle variabili preposte per il conteggio degli agenti che, per ogni step, hanno determinato, attraverso la loro offerta, il prezzo di transazione. I grafici sono due e ciascuno rappresenta la situazione di un determinato agente. La prima osservazione riguarda il

limitato numero di prezzi, visibili nel grafico, proposti dai level agent; la seconda è relativa alle bolle e ai crash di prezzo, che si creano in corrispondenza degli step, in cui un agente level propone un'offerta. Questo sarà visibile meglio nelle prossime variabili che presenterò, le quali riportano gli spread in entrambe le liste tra migliore e peggiore prezzo; sarà possibile notare come, in corrispondenza di un'offerta dei level, gli spread aumentano notevolmente. Questo determina i bruschi cambiamenti di prezzo del titolo e, quindi, le corrispondenze dei livelli di prezzo con la distribuzione Ftse.

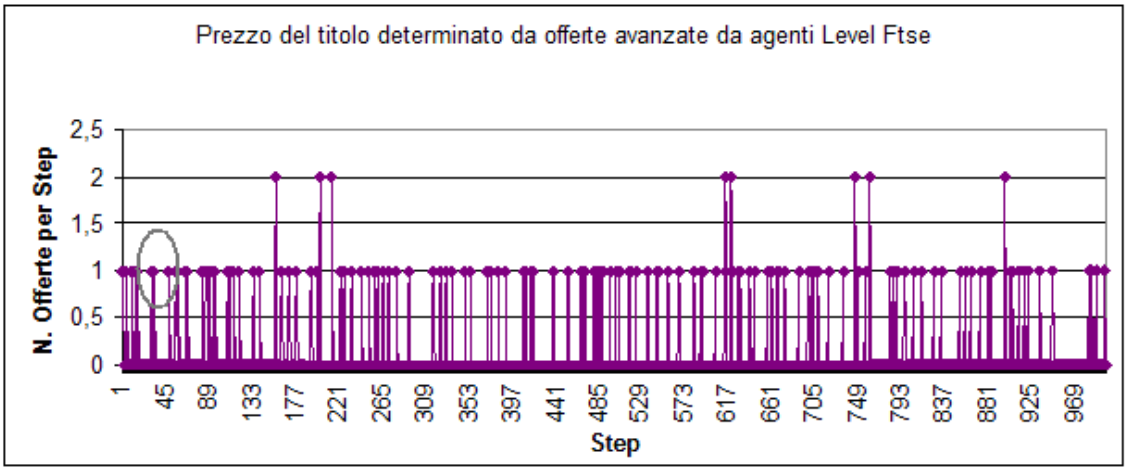


Figura 6 Numero di agenti level che, per step, hanno concluso una transazione ad un prezzo da loro stabilito

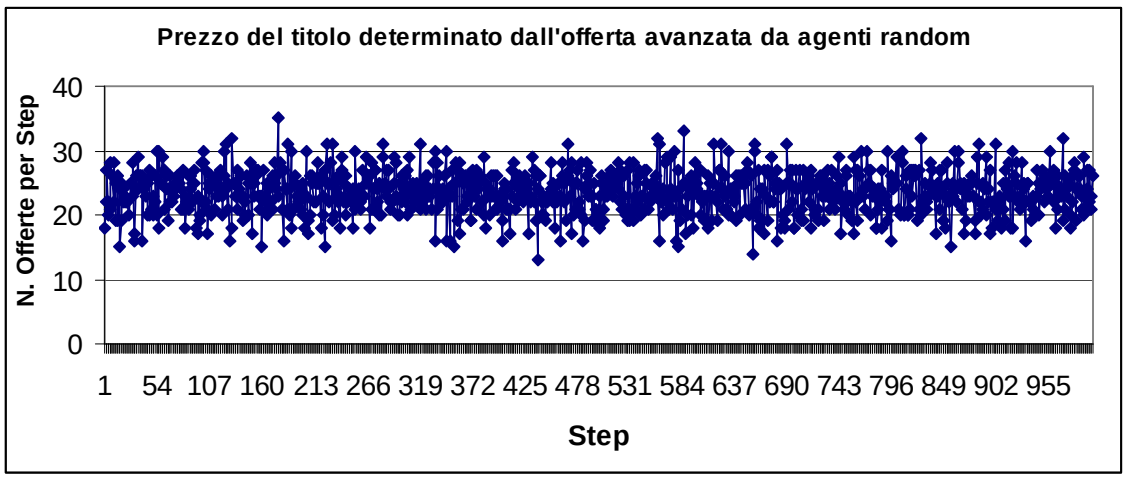


Figura 7 Numero di agenti random che, per step, hanno concluso una transazione ad un prezzo da loro stabilito

E' molto importante associare i due grafici, appena presentati, ai due che seguono; così facendo, emergerà un'elevata significatività dal loro confronto.

In particolare, ciò che emerge sono i bruschi cambiamenti di prezzo, in presenza degli spread. Inoltre, in corrispondenza degli spread più elevati vi sono le offerte proposte dagli agenti level passate ad eseguito (vedere Figura 6 e Figura 9 i riferimenti cerchiati). Chiaramente, non tutte le proposte offerte dai level causano elevati spread; in ogni caso, è dimostrato che l'identità dell'offerente appartiene al level agent in presenza dei più elevati spread.

L'effetto appena presentato è particolarmente significativo perché ben evidenzia la differenza tra il prezzo proposto dagli agenti Ftse, seguendo una strategia ben precisa, e i prezzi intorno ad un intervallo, proposto dai *random*. Inoltre, si può avanzare l'ipotesi che gli agenti "zero intelligence" concludano alcuni contratti fuori dalla media di prezzo con agenti level, per pura casualità. Questo crea, però, sui mercati un effetto a catena, che permette la formazione di trend simili a quelli reali, avvenuti nella settimana di riferimento. L'effetto a catena è smorzato da un'inversione di tendenza, causato, per la maggior parte dei casi, da offerte dei Ftse nuovamente fuori dalla media.

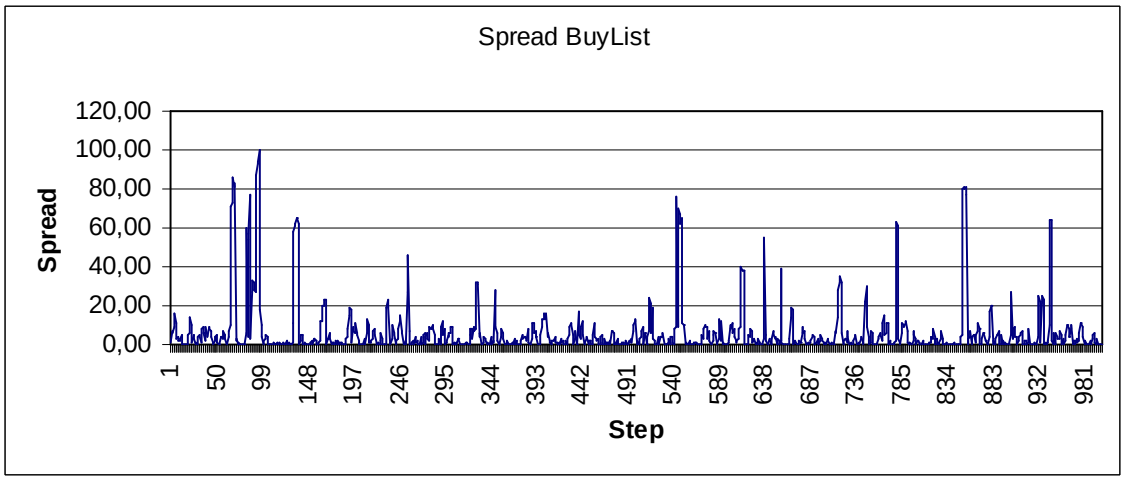


Figura 8 Spread, per step, calcolato tra il migliore e il peggiore prezzo di acquisto, presenti in lista

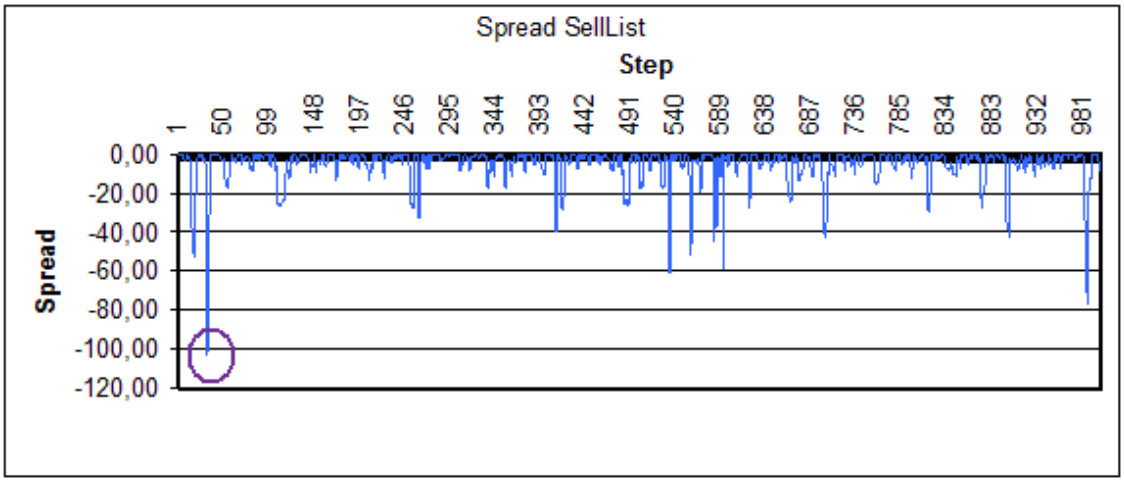


Figura 9 Spread, per step, calcolato tra il migliore e il peggiore prezzo di vendita, presenti in lista

Volendo cercare un collegamento teorico con il comportamento assunto nel mercato simulato dall'unico agente level, si può citare la teoria dei "lead steer" (proposta da Steer e già indicata nel Capitolo 1), in cui si considera che pochi agenti sul mercato sono capaci di fare il giusto *pricing* dei titoli e che i restanti partecipanti al mercato seguono le decisioni dei primi. Quindi per comprendere quali saranno i futuri movimenti di mercato basterà interrogare i "lead steer", evitando di considerare i restanti investitori. Questa spiegazione potrebbe essere fornita per individuare il comportamento di massa degli agenti "zero intelligence" e il comportamento, in controtendenza, dell'unico agente dotato di intelligenza. Per dovere di chiarezza, sono tenuto a sottolineare che la teoria appena citata, pur essendo affascinante, non trova un elevato numero di test computabili, che le permettano di essere accettata come uno dei fondamenti della teoria sul comportamento dei mercati finanziari.

Gli ultimi due grafici, visualizzabili in Figura 10 e 11, richiamano la teoria già osservata da esperimenti, fatti da Terna [2002], in cui si sostiene che le bolle e i crash di prezzo sono una conseguenza dello sbilanciamento, sotto il profilo della lunghezza, tra le liste di offerta. L'effetto sui prezzi è leggermente ritardato rispetto ai valori che si ottengono dai grafici.

Osservando i grafici, in realtà, gli effetti che si vorrebbero rappresentare non sono così nitidi. La motivazione è legata al fatto che la distribuzione di prezzi originali non è

caratterizzata da bolle e crash, anzi, risulta piuttosto “frastagliata”, a causa dell’effetto dei contratti conclusi a prezzi ravvicinati.

Un caso che può essere preso come esempio è rappresentato dallo step 94 (evidenziato dal cerchio rosso) sia in Figura 4 (nella realtà) sia in Figura 5 (in simulazione), dove si registra una bolla di prezzo.

Per la precisione di dati, il numero di offerte non eseguite nella BuyList risulta pari a 21, mentre nella SellList è pari a zero.

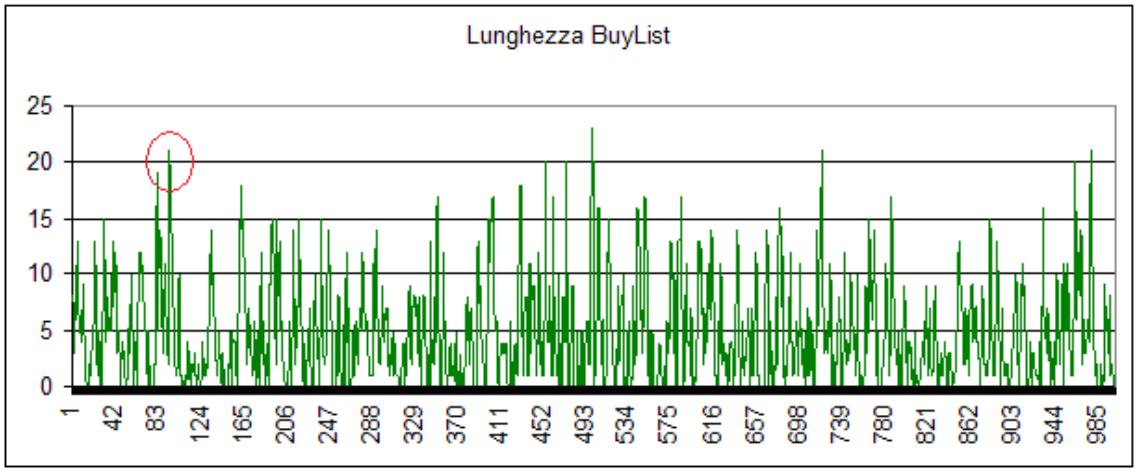


Figura 10 Lunghezza (numero di offerte non ancora eseguite) presenti nella Buy List

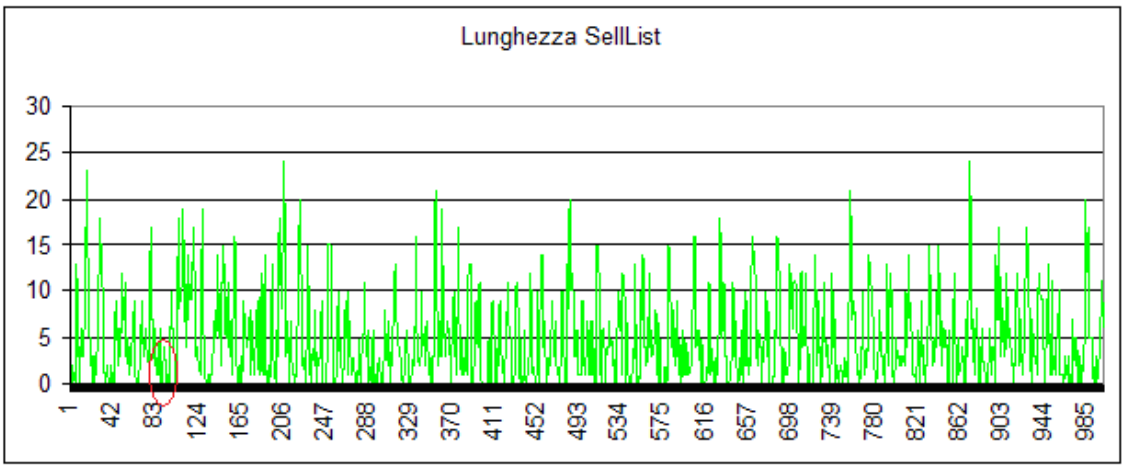


Figura 11 Lunghezza (numero di offerte non ancora eseguite) presenti nella Sell List

Giudizio conclusivo sperimentazioni

Concludendo, si è dimostrato, attraverso questo test e quello proposto nel paragrafo successivo, che è possibile riprodurre dati riscontrati nella realtà attraverso la simulazione.

Attraverso un'analisi critica, ho ottenuto una risposta convincente al problema sorto, relativo alla eccessiva precisione con cui il simulatore ha riprodotto i dati analizzati. Nel grafico in Figura 3, gli agenti *random* sono programmati con una funzione di prezzo, che riesce ad apportare solo piccole variazioni alla base di partenza, cioè all'ultima transazione avvenuta. In altre parole, gli agenti *random* hanno una bassa probabilità di incidere sulla distribuzione, poichè propongono offerte in un *range* di prezzo troppo limitato. Per contro, il comportamento aggregato di 100 agenti, per 500 offerte ad ogni *step,* dovrebbe produrre un'incidenza necessaria ad impedire la formazione di una distribuzione così simile alla realtà.

In Figura 12 è presente un grafico in cui ho simulato lo stesso esempio citato sino ad ora raddoppiando però il *range* dei possibili prezzi offerti dall'agente *random*. La distribuzione risulta ancora simile alla realtà ma è meno marcata.

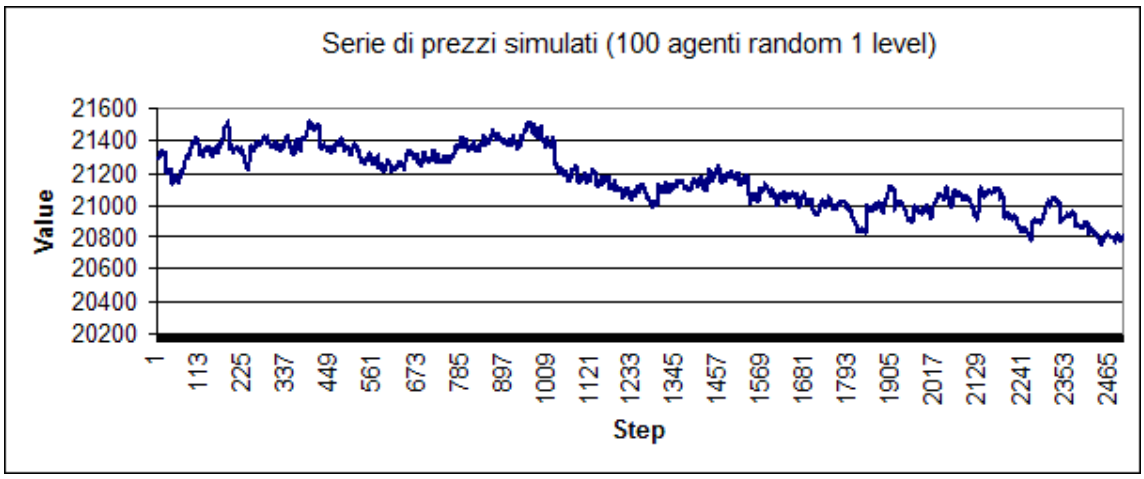


Figura 12 Serie di prezzi simulati con variazioni alla funzione di prezzo programmata nell'agente random

7.5 TEST NUMERO 3: MERCATO POPOLATO DA AGENTI ZERO INTELLIGENCE E VARIATION PRICE REAL AGENTS (LEGATI AL FTSE ALL SHARE)

In questo paragrafo saranno presentati i risultati ottenuti dall'interazione tra agenti *random* e agenti *variation*.

Gli agenti *random* conservano le medesime caratteristiche spiegate nel paragrafo precedente; le caratteristiche generiche del nuovo agente sono, invece, illustrate di seguito:

- agente *variation price real data*: la strategia di mercato che egli mette in atto si basa sulla sua conoscenza di quelle che saranno le future variazioni dell'indice, con alcuni secondi di anticipo rispetto agli altri agenti La sua convinzione è che le variazioni dell'indice bene approssimano le variazioni del prezzo del titolo oggetto di transazione.

Premessa

Il risultato, che si vuole ottenere, è quello di osservare come il comportamento di agenti, che basano la loro strategia sulle variazioni di valore dell'indice Ftse All Share, influisca sulla distribuzione dei prezzi nel mercato simulato. Nel caso in cui si riesca ad osservare una distribuzione di prezzi simulati vicina alla distribuzione di prezzi reali, si potrebbe concludere di aver raggiunto un risultato sorprendente, ancora più importante di quello descritto nel paragrafo precedente. La motivazione risiede nel fatto che, in questo caso, il nostro agente intelligente varia la strategia di azione ad ogni istante, confrontando mercato simulato e mercato reale ed inserendo un prezzo che non riprende livelli reali, bensì variazioni percentuali reali che moltiplicano prezzi simulati.

Anticipo che, il risultato desiderato è stato raggiunto. La calibrazione degli esperimenti ha presentato alcune difficoltà nella programmazione, ma anche nel trovare una giusta proporzione tra numero di agenti *random* e numero di agenti *variation,* che permettesse di visualizzare risultati evidenti, avendo un vincolo di agenti pari a cento.

Le sperimentazioni con la classe *variation* sono state fatte programmando la classe in tre differenti modalità; inoltre, per ciascuna modalità, è stato testato quale sistema raggiungesse maggiormente l'obiettivo prefissato. La motivazione per cui ho

programmato questa classe in tre modalità differenti è quella di voler dimostrare la capacità che il mio programma ha di ricreare un mercato e serie di prezzi reali. In questo modo, avrò la possibilità di capire quale modalità mi permette di ottenere il miglior risultato.
Di seguito anticipo brevemente come sono state programmate le classi e per ciascuna modalità di programmazione spiegherò i risultati ottenuti.

CASO 1. La strategia di azione dell'agente *variation* si basa sulle seguenti condizioni:

- Variazione positiva del valore dell'indice, nell'intervallo di tempo considerato, rispetto alla variazione di prezzo che ha registrato il titolo simulato. In questo caso, l'agente proporrà un'offerta di acquisto con base di partenza il prezzo simulato finale dello step precedente e con variazione percentuale quella registrata dall'indice Ftse.
- Variazione negativa del valore dell'indice, nell'intervallo considerato, inferiore rispetto alla variazione registrata dal titolo simulato. In questo caso, l'agente proporrà un'offerta di vendita con base di partenza il prezzo finale dello step precedente e con variazione percentuale quella registrata dall'indice Ftse.

In questa modalità, l'agente *variation*, non proporrà sempre un'offerta di acquisto o di vendita. Ad esempio, non farà un'offerta nel caso in cui la variazione del Ftse sia inferiore alla variazione del titolo simulato ma entrambe le grandezze siano positive. In questo caso, cercando di dare un'interpretazione al comportamento del mercato, il titolo simulato starebbe registrando un maggiore rendimento rispetto al Ftse. In questa prima modalità, dal momento che il mio obiettivo è quello di cercare di replicare il mercato reale (quindi di replicare i trend), senza prendere in considerazione le intensità, ho voluto che i miei agenti non operassero in tutte le condizioni di mercato. Nelle future modalità programmate, sarà catturato ogni minimo movimento e, in base a questo, sarà proposta un'offerta.
In Figura 13 è possibile notare il codice di programmazione. Per comprendere meglio i passaggi informatici, fare riferimento al manuale utente.

```
def variationRealPriceTransaction(self):
    self.nstep = self.nSteps
    self.passo = 500 / self.nstep
    self.sellPrice=0
    self.buyPrice=0
    b=0
    j=0

    self.finalPriceForStepList = self.aBook.getFinalPriceForStep()

    if len(self.finalPriceForStepList) > 1:
        self.price=self.priceList[-1]
        j = self.step*self.passo
        b = (self.step - 1)*self.passo
        realDataVariation = ((self.values[j]-self.values[b])/ self.values[b])
        simulatorPriceVariation= ((self.price - self.finalPriceForStepList[-1])/self.finalPriceForStepList[-1])
      #  print "REAL DATA VARIATION ",realDataVariation,"SIMULATOR Price Variation",simulatorPriceVariation


        if realDataVariation > simulatorPriceVariation:
            if realDataVariation > 0:
                self.buyPrice = self.finalPriceForStepList[-1]*(1 + realDataVariation)
         #       print "agent.VariationPriceRealData=",self.number, "Buy price","%.2f" % self.buyPrice
                self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                self.sellPrice = self.finalPriceForStepList[-1]*(1 + realDataVariation)
          #      print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                self.aBook.setSellDecision([self.sellPrice,self,self.type])
```

Figura 13 Codice di programmazione relativa classe *variation* (Caso 1)

Del seguente caso, saranno approfonditi anche gli aspetti relativi ad alcune variabili già osservate per i *level agent*, per cercare di comprendere meglio le cause che determinano i movimenti di prezzo. Per i successivi due casi, questo processo non sarà presentato; verranno posti sotto osservazione solamente i grafici di andamento del titolo simulato e del Ftse per evidenziare le analogie nelle due distribuzioni.

Spiegazione dei risultati ottenuti: CASO 1

In Figura 14 è possibile osservare il risultato generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 80
- numero agenti variation: 20
- numero step: 500
- numero cicli: 5

Il numero degli agenti è stato scelto in questo modo perché, dopo ripetuti tentativi di calibrazione, avendo come vincolo la sommatoria degli agenti pari a 100, esso è risultato il miglior compromesso finalizzato al raggiungimento dell'obbiettivo, da me prefissato. Per quanto riguarda il numero di cicli e il numero di step, la scelta, come nel primo caso, risulta forzata a causa del numero limitato di dati reali a mia disposizione.
In Figura 14 è possibile visualizzare il risultato della simulazione.

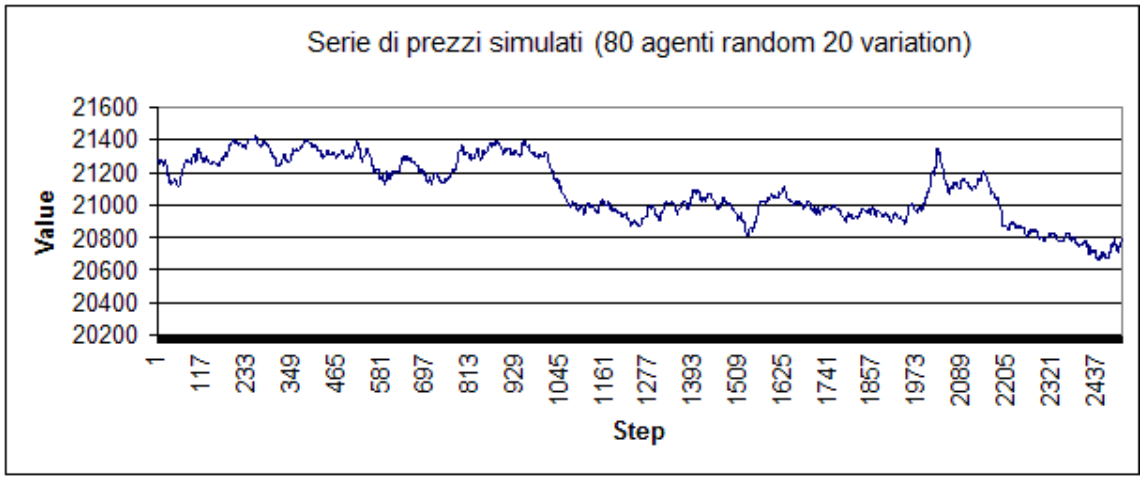


Figura 14 Serie di prezzi ottenuti da simulazione in un mondo popolato di 80 agenti random e 20 agenti variation

Chiaramente, le similitudini sono meno nitide rispetto all'esercizio proposto con i level. In ogni caso, il risultato ottenuto è molto soddisfacente, considerata la difficoltà di poter ripetere una distribuzione di prezzi simile alla realtà. Per apprezzare meglio il confronto, fare riferimento alla Figura 15 in cui sono rappresentati, sullo stesso grafico, gli andamenti del prezzo reale e del prezzo simulato.

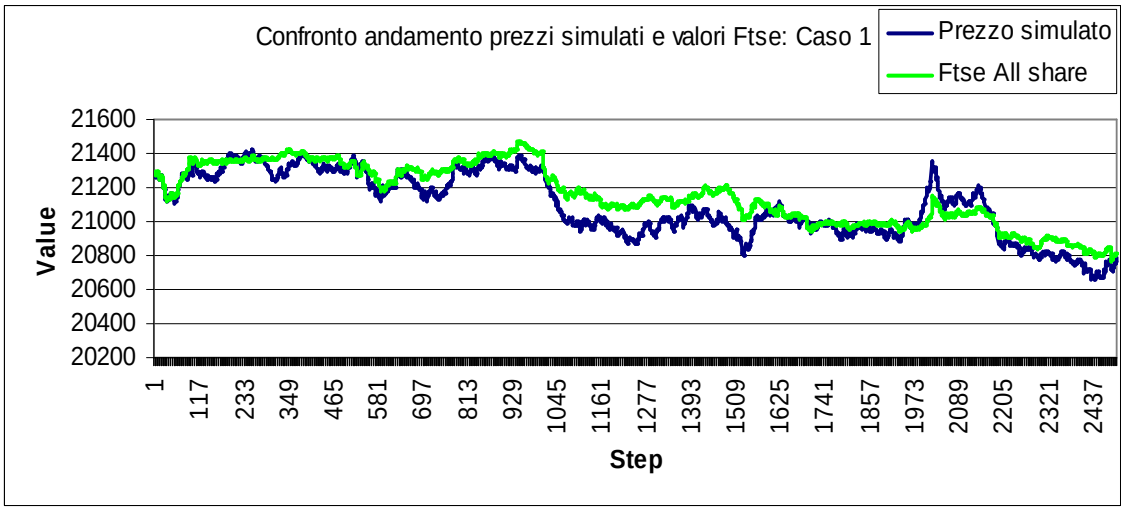


Figura 15 Confronto tra serie di prezzi simulati e andamento Ftse all Share (13-12-2011 / 17/12/2011)

Ciò che si può notare è che cambiamenti di andamento nella realtà hanno effetti molto più accentuati in simulazione. Questo effetto è particolarmente visibile se si prende in riferimento un intervallo più breve per numero di cicli. I numerosi cambiamenti di prezzo nel grafico del prezzo simulato, in Figura 16, saranno facilmente riconducibili a cambiamenti di prezzo nel grafico Ftse. Per aiutare il lettore nella comprensione di quanto indicato, evidenzio i punti di interesse.

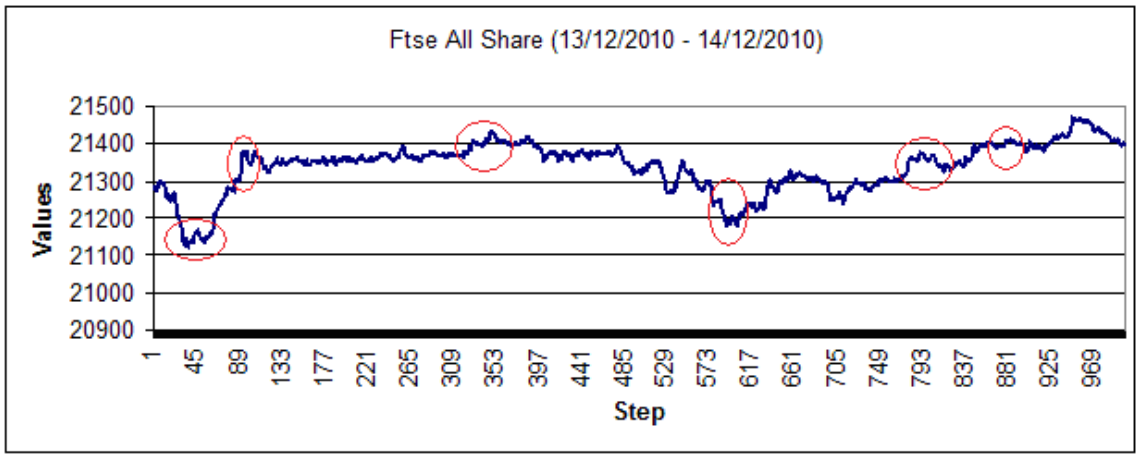


Figura 16 Particolare della distribuzione Ftse All Share relativa a 2 giorni di contrattazione

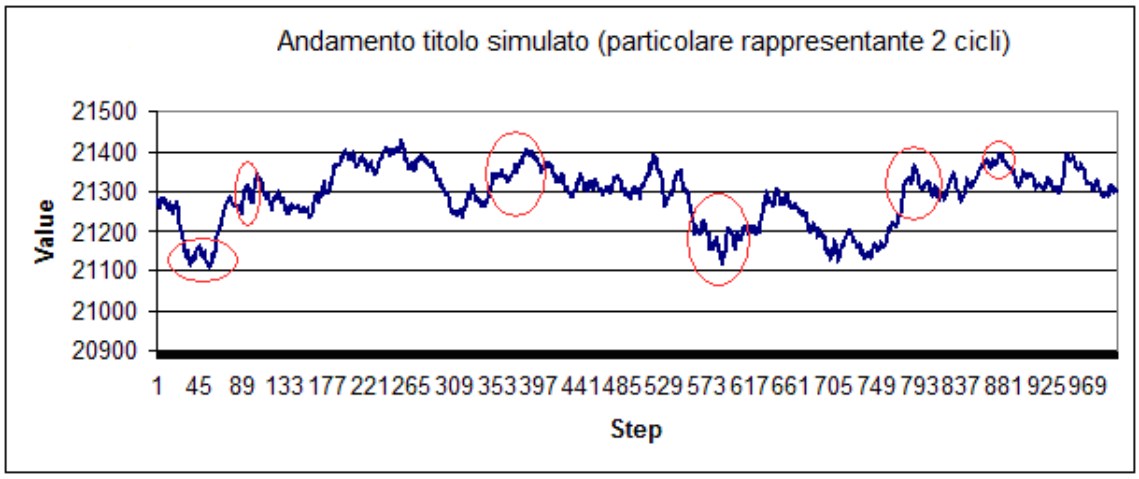


Figura 17 Particolare della serie di prezzi simulati (85 random, 15 variation) caso 1

Un commento generale può essere fatto in prima analisi, osservando le variabili che aiutano a comprendere le cause dei cambiamenti di prezzo. Il fenomeno che ho rilevato riguarda principalmente i seguenti aspetti:

1- l'andamento dei prezzi non è più condizionato fortemente dalle scelte dei *variation*

2- non ci sono chiare inversioni di trend

3- sono necessari più agenti *variation* per far sì che la distribuzione subisca dei condizionamenti in base alle decisioni di quest'ultimi

In pratica, seguendo la teoria già discussa per i *level*, anche in questo caso gli agenti intelligenti hanno più potere, attraverso le loro offerte, nel condizionare gli andamenti di prezzo. Infatti, la loro partecipazione ai mercati, essendo superiore rispetto al caso visualizzato in precedenza, crea un aumento degli spread nelle rispettive liste Buy e Sell.

Focalizziamo l'attenzione sui grafici che raffigurano gli spread. Il maggior numero di agenti e la maggior partecipazione ai mercati determina un aumento nella frequenza di spread elevati. La ragione risiede nella probabilità che questi agenti hanno di presentare offerte superiori al *range* medio.

Come è possibile osservare dal grafico, in Figura 18 e 19, facendo un confronto con gli spread nella simulazione dei *level*, si nota che la frequenza è molto aumentata. Questo è un indicatore che spiega l'aumento nella variabilità del prezzo del titolo. Inoltre, osservando i dati evidenziati sul grafico ci si accorge che, in prossimità dei più elevati

aumenti di spread nelle rispettive liste, si registra un boom di prezzo, se lo spread è aumentato nella Buy List, e un crash, se l'aumento dello spread è avvenuto nella Sell List.

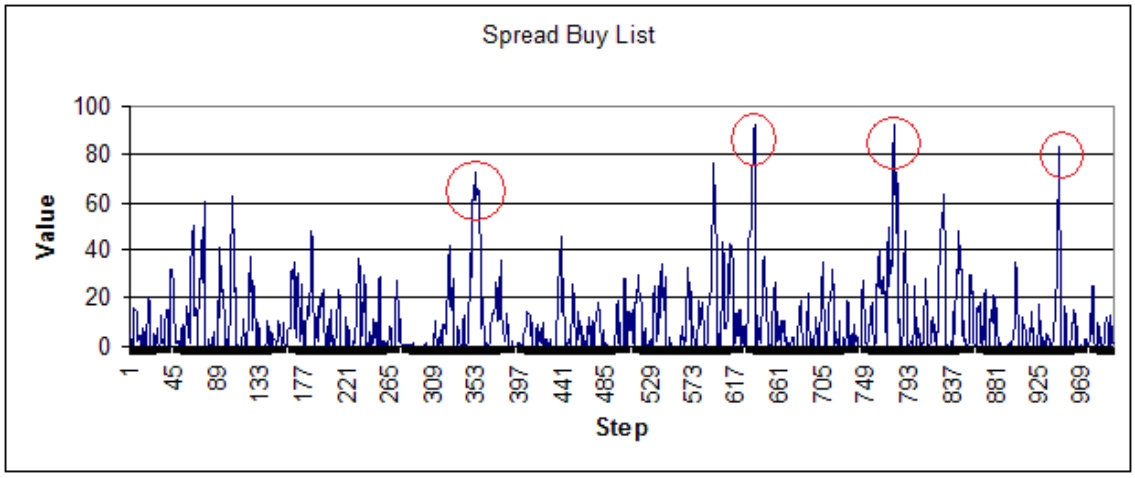


Figura 18 Spread, per step, calcolato tra il migliore e il peggiore prezzo di acquisto, presenti in lista

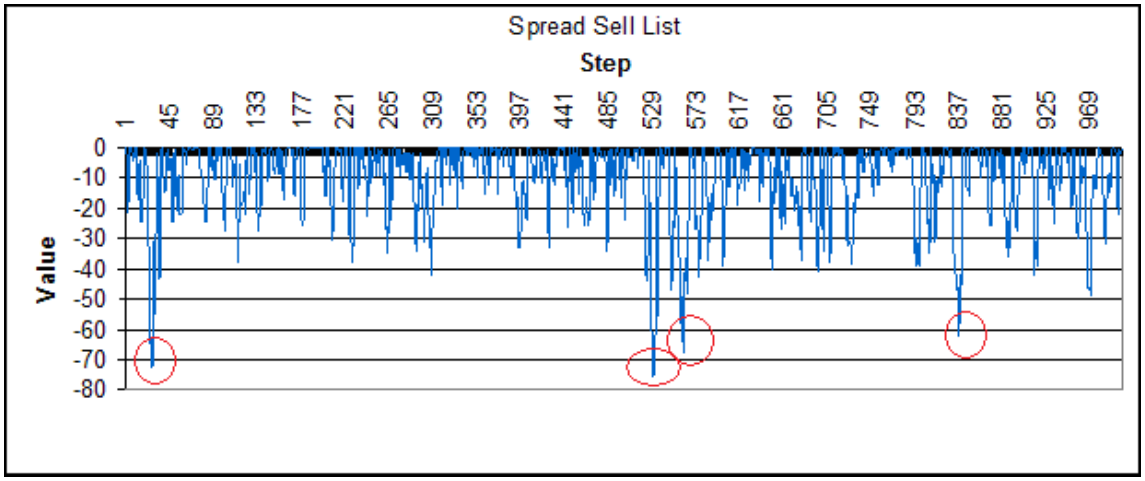


Figura 19 Spread, per step, calcolato tra il migliore e il peggiore prezzo di acquisto, presenti in lista

Giudizio conclusivo sperimentazioni: CASO 1

Concludendo, si può affermare che l'obiettivo prefissato al momento dell'ideazione della classe è stato raggiunto perché le serie di prezzi simulati si avvicina molto alla distribuzione del Ftse All Share preso come riferimento. Questo ci consente di utilizzare la classe *variation* come sistema per rappresentare distribuzioni di prezzi avvenute nella realtà in simulazione. Ciò che sarà fatto in futuro, infatti, sarà inserire

gli agenti programmati in questa classe in un mondo simulato, così da creare una sorta di competizione con altri agenti intelligenti. La domanda che ci porremo quando analizzeremo i risultati sarà quindi: ”Come il comportamento di una classe x avrebbe modificato l’andamento di un mercato reale?”.

CASO 2. La strategia di azione dell’agente *variation,* programmato in questa seconda modalità, prende in considerazione qualsiasi condizione di confronto tra le variazioni registrate dall’indice e quelle registrate dal titolo. A differenza del Caso 1, l’agente *variation,* in qualsiasi situazione di mercato sia chiamato ad operare, proporrà sempre un’offerta di acquisto o di vendita. Inoltre, il prezzo offerto sarà pari al valore registrato dall’ultima transazione nello step precedente moltiplicato per lo spread, nell’istante t, tra variazione del prezzo simulato e variazione dell’indice Ftse.
Il processo di decisione, in questa modalità, è più complesso rispetto al caso precedente, come è possibile notare dal codice programmato in Figura 20.

```
def variationRealPriceTransaction(self):
    self.nstep = self.nSteps
    self.passo = 500 / self.nstep
    self.sellPrice=0
    self.buyPrice=0
    b=0
    j=0

    self.finalPriceForStepList = self.aBook.getFinalPriceForStep()

    if len(self.finalPriceForStepList) > 1:
        self.price=self.priceList[-1]
        j = self.step*self.passo
        b = (self.step - 1)*self.passo
        realDataVariation = ((self.values[j]-self.values[b])/ self.values[b])
        simulatorPriceVariation= ((self.price - self.finalPriceForStepList[-1])/self.finalPriceForStepList[-1])
      #  print "REAL DATA VARIATION ",realDataVariation,"SIMULATOR Price Variation",simulatorPriceVariation

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                if simulatorPriceVariation > 0:
                    self.sellPrice = self.finalPriceForStepList[-1]*(1 +(realDataVariation+ simulatorPriceVariation))
          #         print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                if simulatorPriceVariation < 0:
                    self.sellPrice = self.finalPriceForStepList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #         print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])
```

```
        if realDataVariation > simulatorPriceVariation:
            if realDataVariation > 0:
                if simulatorPriceVariation < 0:
                    self.buyPrice = self.finalPriceForStepList[-1]*(1 + (realDataVariation+simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number, "Buy price","%.2f" % self.buyPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation > simulatorPriceVariation:
            if simulatorPriceVariation > 0:
                if realDataVariation > 0:
                    self.buyPrice = self.finalPriceForStepList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #         print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation > simulatorPriceVariation:
            if simulatorPriceVariation < 0:
                if realDataVariation < 0:
                    self.buyPrice = self.finalPriceForStepList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #         print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation > 0:
                if simulatorPriceVariation > 0:
                    self.sellPrice = self.finalPriceForStepList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #         print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])
```

Figura 20 Codice di programmazione relativa classe *variation* (Caso 2)

Visualizzazione dei risultati ottenuti: CASO 2

In Figura 21 è possibile osservare il risultato generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 85
- numero agenti variation: 15

- numero step: 500
- numero cicli: 5

In Figura 21 è possibile apprezzare il confronto tra le serie di prezzi simulati e l'andamento del Ftse All share. La simulazione in questo caso propone risultati simili a quelli già visualizzati nel caso 1.

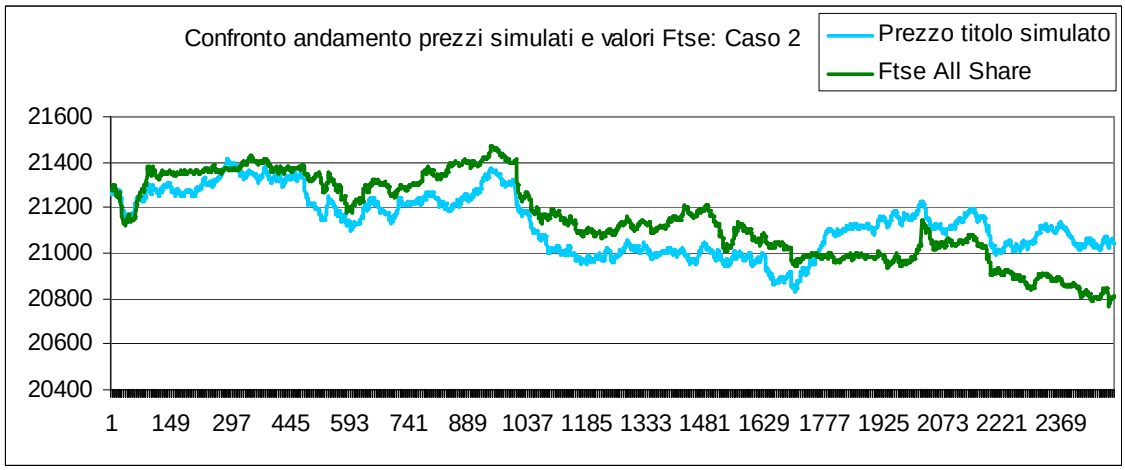


Figura 21 Confronto serie di prezzi simulati (agente *variation* programmato in modalità 2) e agente Ftse

Giudizio conclusivo sperimentazioni: CASO 2

Il risultato ottenuto dai test proposti in questa classe garantisce il reale funzionamento del programma. Il numero maggiore di condizioni inserite nel codice permette all'agente *variation* di proporre molti più prezzi rispetto al caso precedente. Questo ha permesso di ottenere risultati più che soddisfacenti inserendo un numero inferiore di agenti *variation* e un numero superiore di agenti *random.*

CASO 3. La strategia di azione dell'agente *variation,* programmato in questa modalità, è identica alla strategia presentata nel caso 2. Ciò che è stato modificato è la base dell'offerta proposta; in questo caso, infatti, il prezzo preso in considerazione sarà pari a quello dell'ultima transazione avvenuta nel mercato.

La differenza tra le sperimentazioni del caso 3 e del caso 2 è la maggiore volatilità delle offerte proposte dagli agenti *variation*. Nel caso 2, infatti, per ogni proposta offerta nel

medesimo step, gli agenti *variation* hanno la medesima base di partenza. Tale condizione non si verifica nel caso 3 come dimostrato dal codice di programmazione mostrato in Figura 23.

```
def variationRealPriceTransaction(self):
    self.nstep = self.nSteps
    self.passo = 500 / self.nstep
    self.sellPrice=0
    self.buyPrice=0
    b=0
    j=0

    self.finalPriceForStepList = self.aBook.getFinalPriceForStep()

    if len(self.finalPriceForStepList) > 1:
        self.price=self.priceList[-1]
        j = self.step*self.passo
        b = (self.step - 1)*self.passo
        realDataVariation = ((self.values[j]-self.values[b])/ self.values[b])
        simulatorPriceVariation= ((self.price - self.finalPriceForStepList[-1])/self.finalPriceForStepList[-1])
      #  print "REAL DATA VARIATION ",realDataVariation,"SIMULATOR Price Variation",simulatorPriceVariation

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                if simulatorPriceVariation > 0:
                    self.sellPrice = self.priceList[-1]*(1 +(realDataVariation+ simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation < 0:
                if simulatorPriceVariation < 0:
                    self.sellPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])

        if realDataVariation > simulatorPriceVariation:
            if realDataVariation > 0:
                if simulatorPriceVariation < 0:
                    self.buyPrice = self.priceList[-1]*(1 + (realDataVariation+simulatorPriceVariation))
           #         print "agent.VariationPriceRealData=",self.number, "Buy price","%.2f" % self.buyPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])
```

```
        if realDataVariation > simulatorPriceVariation:
            if simulatorPriceVariation > 0:
                if realDataVariation > 0:
                    self.buyPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation > simulatorPriceVariation:
            if simulatorPriceVariation < 0:
                if realDataVariation < 0:
                    self.buyPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setBuyDecision([self.buyPrice,self,self.type])

        if realDataVariation < simulatorPriceVariation:
            if realDataVariation > 0:
                if simulatorPriceVariation > 0:
                    self.sellPrice = self.priceList[-1]*(1 +(realDataVariation-simulatorPriceVariation))
          #          print "agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" % self.sellPrice
                    self.aBook.setSellDecision([self.sellPrice,self,self.type])
```

Figura 22 Figura 20 Codice di programmazione relativa classe variation (Caso 3)

Visualizzazione dei risultati ottenuti: CASO 3

In Figura 23 è possibile osservare il risultato generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 85
- numero agenti variation: 15
- numero step: 500
- numero cicli: 5

In Figura 23 è possibile osservare il confronto tra la serie di prezzi simulati e i livelli del titolo Ftse. In prima analisi, questa terza modalità mi garantisce di ottenere i risultati più soddisfacenti.

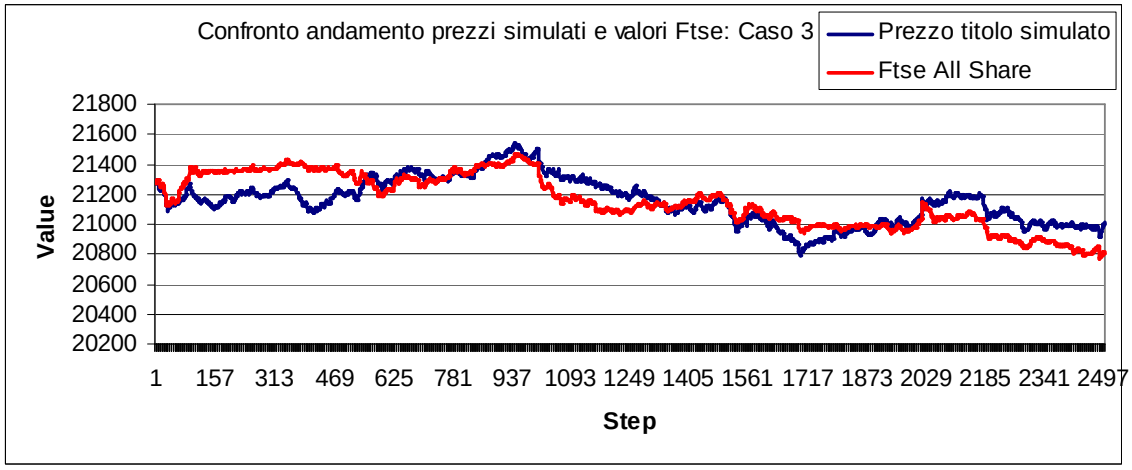


Figura 23 Confronto serie di prezzi simulati (agente *variation* programmato in modalità 3) e agente Ftse

Giudizio conclusivo sperimentazioni: CASO 3

Secondo gli esperimenti proposti nel Caso 3, il risultato ottenuto è il più soddisfacente rispetto a quelli ottenuti nei precedenti casi. Le due distribuzioni sono molto vicine e presentano i medesimi movimenti di prezzo.

Le funzioni decisionali programmate secondo questa terza metodologia sono le più vicine alla realtà, secondo il ragionamento economico che l'agente *variation* compie. Questo sta a significare che, maggiore è la precisione nella programmazione di funzioni che emulino comportamenti di agenti nella realtà, migliore è il risultato che si ottiene in termini di confronto tra simulazione e realtà. Per i futuri esperimenti, la classe

programmata secondo la modalità 3 sarà presa come riferimento per collegare la realtà alla simulazione.

7.6 TEST NUMERO 4: MERCATO POPOLATO DA AGENTI ZERO INTELLIGENCE,VOLUME AGENTS E VARIATION PRICE REAL AGENTS

In questo paragrafo saranno analizzati gli effetti creati dalle interazioni tra queste tre categorie di agenti. Lo studio dei comportamenti sarà fatto dividendo gli esperimenti in due grandi blocchi. Nella prima parte saranno osservati i comportamenti risultanti dall'interazione tra agenti *random* e agenti *volume*. Nella seconda parte, si introdurranno all'interno del mercato gli agenti volume. Questo sarà fatto per poter analizzare le distorsioni dal punto di vista della distribuzione dei prezzi che una categoria, dotata di una particolare strategia razionale (volume agents), crea rispetto alla distribuzione Ftse. Come visto nei paragrafi precedenti, l'agente variation garantisce una certa sicurezza nel condizionare la distribuzione, secondo gli andamenti del Ftse All Share.

Le caratteristiche degli agenti random e variation, programmati secondo la modalità 3, non variano. Le caratteristiche generali del nuovo agente sono, invece, indicate di seguito:

- agente *volume:* la strategia di azione che caratterizza questa particolare classe si basa sui volumi di transazione. Un agente volume agisce solamente nel caso in cui i volumi di transazione aumentino. Se si verifica la situazione di calo di transazioni, l'agente in questione rimarrà inattivo. Per quanto riguarda invece la decisione di acquisto o vendita, essa si baserà sui trend di mercato. Ad esempio, nel caso in cui il prezzo di mercato al tempo t risultasse inferiore ad una certa media di prezzi, considerando un numero variabile di titoli, l'agente proporrà un'offerta di vendita. Viceversa nel caso in cui il prezzo al tempo t risultasse superiore ad una certa media di prezzi, che considera un numero variabile di titoli. Le situazioni appena descritte rappresentano,

rispettivamente, trend in decrescita e trend in crescita; l'agente volume, attraverso le sue decisioni, rafforzerà questi trend.

TEST 1.

Premessa: agenti random e agenti volume

In questa primo esperimento, come anticipato in precedenza, saranno fatti agire agenti random e agenti volume insieme. L'ideazione della strategia, che guida le decisioni dell'agente volume, nasce da un concetto legato ai volumi di transazione. Nelle strategie di trading, nell'analisi tecnica e nello studio degli andamenti di prezzo viene data molta importanza ai volumi di scambio. Le motivazioni possono essere varie, tra le più importanti: liquidità del titolo e forza del trend. Un soggetto razionale operante sui mercati non solo valuta il prezzo del titolo, bensì anche la relazione che questa grandezza ha con i volumi di scambio. I volumi, inoltre, non devono essere considerati come grandezze assolute, bensì come tendenze.
L'effetto, quindi, atteso dall'operatività di questi agenti è di vedere trend di prezzo rafforzati, creando distribuzioni fortemente caratterizzate da andamenti al rialzo e al ribasso. Una distribuzione dei prezzi piatta sarebbe una caratteristica che mi farebbe dubitare sull'esattezza della funzione di decisione programmata.

Visualizzazione dei risultati ottenuti

In Figura 24 è possibile osservare il risultato generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 90
- numero agenti volume: 10
- numero step: 500
- numero cicli: 5

Come è possibile osservare dall'immagine, è stato raggiunto l'obiettivo desiderato: una serie di prezzi formata da una alternanza di picchi di prezzo.

Il risultato atteso dal lettore, probabilmente, è una sequenza di picchi molto più accentuata. A questo proposito è bene porre attenzione alla scala dei valori: essendo raffigurato un intervallo di 200 punti, qualsiasi movimento di prezzo può essere dell'ordine di decine di punti. Considerando, inoltre, che il prezzo proposto dagli agenti non è così elevato rispetto al prezzo base dell'ultima transazione, il fenomeno che si osserva può definitivamente essere considerato come una serie di picchi al rialzo e al ribasso di una certa consistenza.

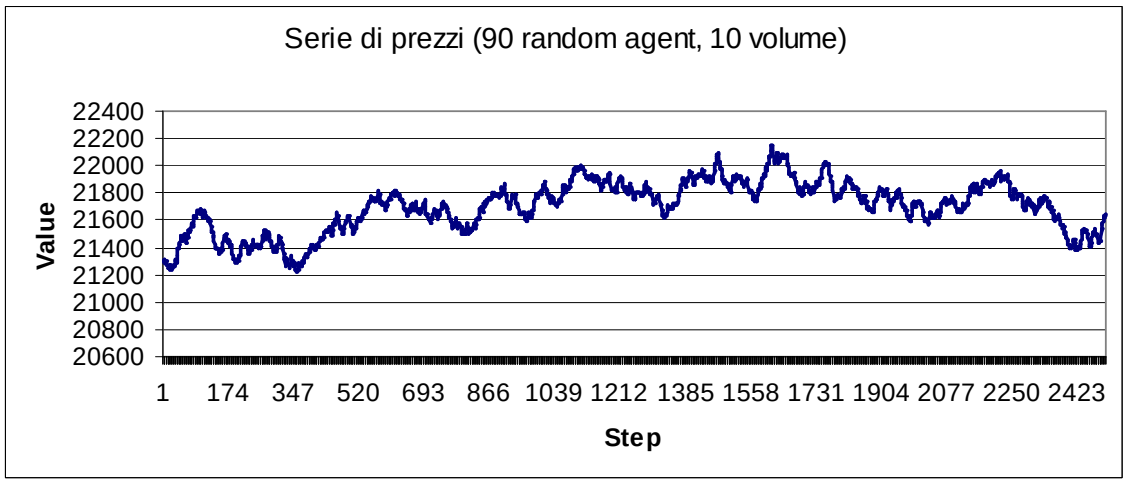


Figura 24 Serie di prezzi generati in un mercato popolato da 90 agenti random e 10 agenti volume

Per comprendere meglio i risultati ottenuti, credo sia necessario prestare attenzione alla variabile preposta per il conteggio dell'operatività degli agenti volume per step, la quale è strettamente collegata all'aumento degli spread. Per questa ragione mostrerò di seguito i due grafici, che rappresentano il particolare relativo ad un numero limitato di step, così da apprezzare sia l'operatività degli agenti volume, sia i cambiamenti di prezzo.

Osservando i due grafici, ho messo in evidenza alcuni punti in corrispondenza dei quali vorrei far notare l'operatività degli agenti volume. Quando essi entrano in azione, infatti, è molto probabile che lo facciano tutti nella medesima direzione in acquisto o in vendita causando l'aumento, in maniera spropositata, degli spread in una delle due liste. Un osservatore attento, potrebbe dubitare che tale fenomeno sia il medesimo, osservato prima. La risposta è che, nei precedenti casi, gli agenti erano capaci di

condizionare il mercato perché avanzavano offerte fuori dal *range* medio di offerte proposte. In questo caso, invece, le offerte proponibili dagli agenti volume sono in linea con quelle proposte dai *random. S*pread di prezzo, che raggiungono livelli pari ad 80, sono spiegabili solamente con serie di agenti che nel medesimo *step* offrono prezzi nella stessa direzione.

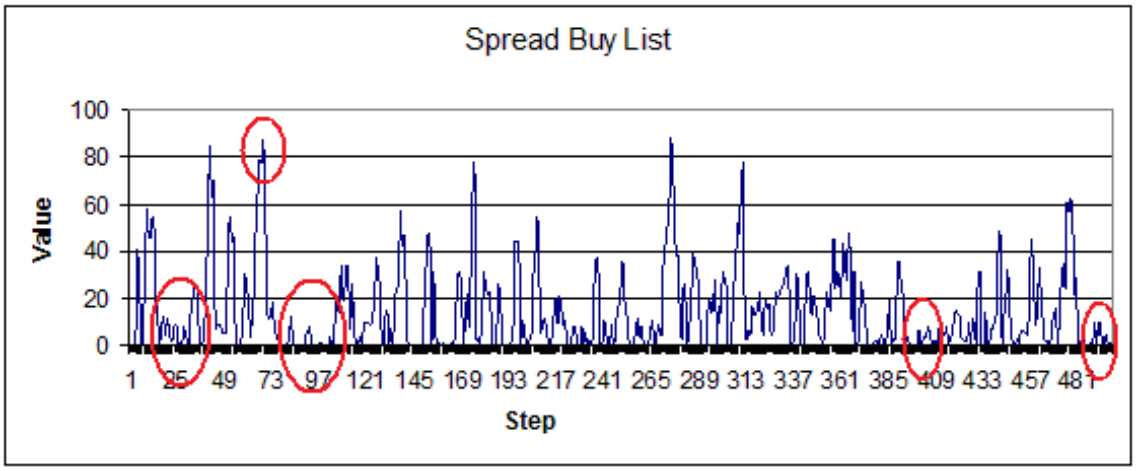


Figura 25 Spread, per step calcolato tra il migliore e il peggiore prezzo di acquisto, presenti in lista

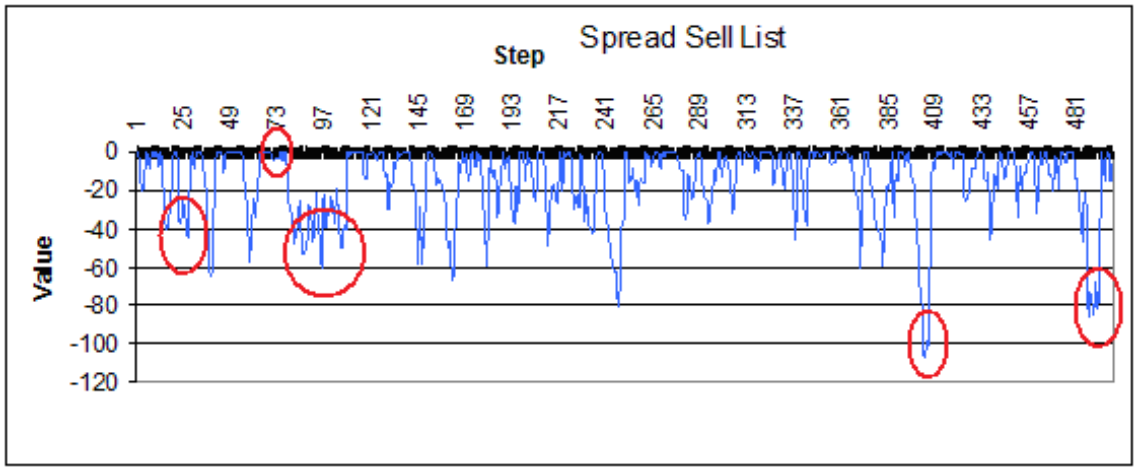


Figura 26 Spread, per step calcolato tra il migliore e il peggiore prezzo di vendita, presenti in lista

Giudizio conclusivo sperimentazioni

Considerando il grafico delle serie dei prezzi e la spiegazione relativa alle variabili preposte per il conteggio degli spread per step, si può concludere che l'agente volume influenza il mercato attraverso la sua operatività. L'obiettivo prefissato al momento

dell'ideazione del codice di programmazione della classe ha avuto un buon esito computazionale.

TEST 2

Premessa: agenti random, agenti volume e agenti variation

In questa seconda parte di test si vuole comprendere il comportamento aggregato di queste tre tipologie di agenti. Ciò a cui si pone maggiore attenzione è la distorsione che il comportamento dell'agente volume genera, nelle serie di prezzi, confrontando i valori simulati e la serie Ftse. Come è stato dimostrato in precedenza, l'agente *variation,* inserito in un mercato simulato, con una giusta calibrazione tra numero di agenti e numero di cicli, garantisce un certa affidabilità di distribuzione e di livello di prezzo, simili al Ftse su cui basa la propria strategia.

Visualizzazione dei risultati ottenuti

In Figura 27 è possibile osservare il risultato generato dal comportamento aggregato delle tre categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 80
- numero agenti volume: 10
- numero agenti variation: 10
- numero step: 500
- numero cicli: 5

Per quanto riguarda la calibrazione tra numero di agenti *volume* e agenti *variation*, si è deciso di inserire un numero pari per evitare di rendere squilibrato il mercato sul numero di agenti intelligenti.

Come è possibile osservare dal grafico, il risultato ottenuto è molto interessante. In prima analisi emerge che i due agenti intelligenti hanno modificato la distribuzione in questo senso:

a) agente *variation*: ha dato un'impronta chiara alla serie dei prezzi tant'è che le due serie, Ftse e variation, sono molto vicine e simili per movimenti di prezzo

b) agente volume: il suo contributo lo si può notare perché i movimenti di prezzo sono molto più accentuati e lo si nota dal confronto tra serie simulate e serie reali. Dal momento che la loro strategia è quella di rafforzare i trend, siano essi al rialzo o al ribasso, si può concludere che il loro operato è visibile.

Di seguito, sarà dato risalto a particolari variabili, per poter comprendere meglio il fenomeno oggetto di studio.

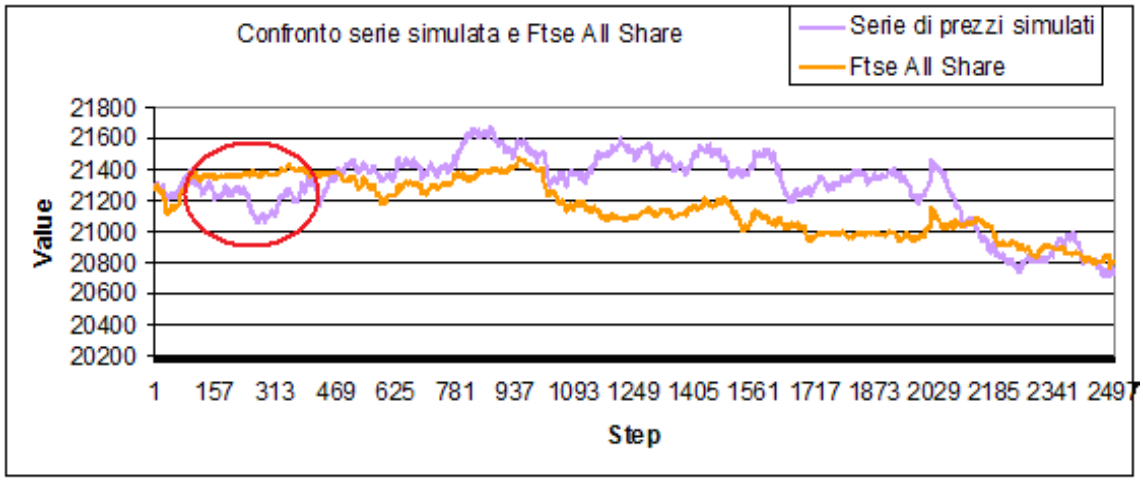


Figura 27 Confronto tra serie di prezzi simulati (agente *random,volume,variation*) e serie di valori Ftse

Gli effetti più interessanti, per poter comprendere il fenomeno, riguardano due casi specifici:

2 il primo aspetto è relativo ai cambiamenti di prezzo inversi rispetto al normale andamento dell'indice di riferimento

3 il secondo aspetto riguarda il presentarsi di trend rinforzati e accentuati rispetto alla distribuzione di riferimento

Entrambi gli effetti sono causati dall'operatività degli agenti *volume*.

Il primo fenomeno, cerchiato in Figura 27, è causato dall'operatività degli agenti che causano, attraverso le loro offerte di vendita, l'aumento degli spread nella Sell List e, contemporaneamente, l'annullamento di molti spread nella Buy List. Questo avviene perché gli agente volume, essendo in una fase di mercato di prezzi in calo, attivandosi aumentano la forza del trend e, quindi, il calo dei prezzi. In Figura 28, nel particolare di mercato dallo step 100 allo step 440 circa, molti spread sono pari a zero; questo spiega l'effetto di andamento inverso dei mercati, rispetto all'indice di riferimento.

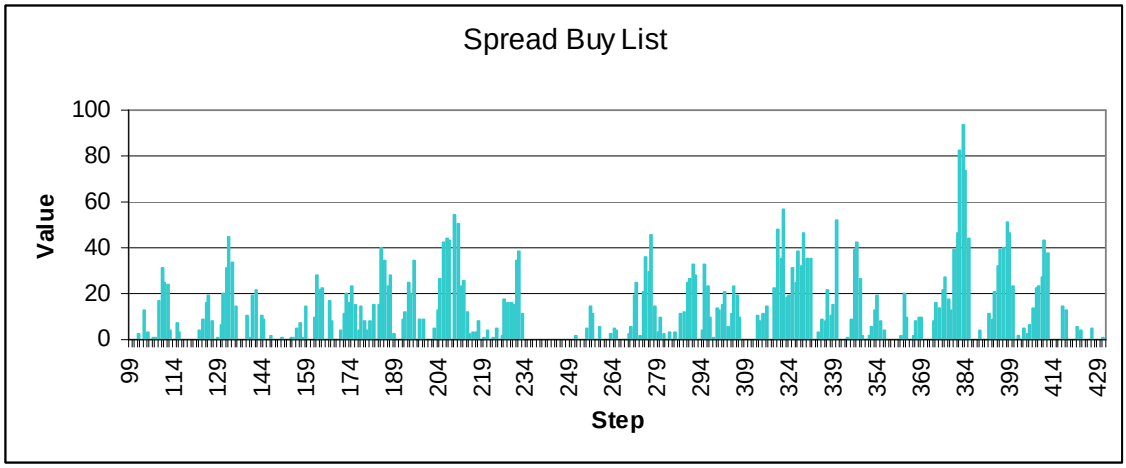


Figura 28 Spread, per step calcolato tra il migliore e il peggiore prezzo di acquisto, presenti in lista

Il secondo effetto è creato da una situazione analoga a quella appena presentata. La differenza riscontrabile è legata al fatto che i *variation,* precedentemente, operavano in una direzione di mercato opposta ai volume; in questo secondo caso, invece, la direzione delle offerte delle due classi si unisce e determina, con un effetto sommatorio, una maggiore forza di trend. Per evitare una ripetizione nei grafici (situazione già vista con la sola differenza nei tempi), non presento il grafico degli spread.

Giudizio conclusivo sperimentazioni

Il risultato ottenuto facendo interagire queste tre classi è molto interessante soprattutto se osservato in un punto specifico (dallo *step* 2000 in avanti) e messo a confronto con il test successivo (Figura 31) e il grafico presentato in Figura 23. Nei grafici a cui si sta facendo riferimento, dal punto 2000 in avanti, la distribuzione simulata ripete i medesimi movimenti della distribuzione reale ma ad un livello superiore rispetto alla distribuzione originale. Lo stesso comportamento non avviene nella simulazione in cui partecipano gli agenti *volume* perché, dal punto di crash che segue il boom in corrispondenza dello *step* 2000, gli agenti *volume* agendo rafforzando il trend riportano la distribuzione ad un livello inferiore rispetto al Ftse. Nel caso in cui i *volume* non agissero amplificando il trend in discesa, mai osserveremo il fenomeno

appena descritto perché i *variation*, non essendoci più così forti sbalzi di prezzo, presentano offerte nello stesso *range* dei *random*, quindi, non hanno il potere di condizionare la distribuzione così tanto come negli *step* precedenti. Questo fenomeno è dimostrabile con i *covered* nel test successivo, i quali, avendo caratteristiche diverse rispetto ai volume, non sono capaci di riportare la distribuzione simulata a livelli simili o inferiori ai valori Ftse dal punto 2000 in avanti.

7.7 TEST NUMERO 5: MERCATO POPOLATO DA AGENTI ZERO INTELLIGENCE,COVERED AGENTS E VARIATION PRICE REAL AGENTS

In questo paragrafo saranno osservati gli effetti, derivanti dall'interazione di queste tre classi. L'impostazione dei test sarà analoga alla precedente, quindi sarà divisa in due specifiche sezioni. Nella prima saranno fatti interagire solamente agenti random e agenti covered. Nella seconda, invece, saranno inseriti anche gli agenti variation, per confrontare con la serie di valori Ftse, il contributo degli agenti covered al mercato. In questa particolare serie di esperimenti, la composizione del mercato sarà leggermente diversa rispetto ai casi precedenti. La motivazione risiede nel fatto che gli agenti *covered* possono agire una sola volta, a causa della complicata funzione dedita al calcolo dei rendimenti. Per ottemperare a questo problema, inserirò un maggiore numero di agenti *covered*, che si distribuiranno nell'operare durante la settimana di contrattazione. In maniera semplificata, agendo in questo modo, ho risolto il problema di avere agenti *covered,* che agiscono in tutta la simulazione e non solo all'inizio. Se non avessi, infatti, inserito la funzione appena spiegata, i miei agenti avrebbero agito tutti nei primi 20 o 30 step, per poi non agire più, dati i vincoli del calcolo del rendimento.

Come in tutti i precedenti casi, gli agenti *random* e *variation* conservano sempre le medesime caratteristiche. Di seguito, spiegherò in generale, la strategia applicata dai *covered*:

- agente *covered*: la sua strategia di operatività si basa sull'acquisizione di una posizione sul mercato (mediante acquisto di un titolo), ricoperta

immediatamente attraverso la vendita di un'opzione. I tecnicismi collegati a questa classe sono molto avanzati, perché l'agente *covered* è dotato di sistemi che calcolano esattamente il prezzo di un'opzione call e put, attraverso la formula di Black and Scholes al tempo *t*. All'interno della formula sono inseriti i dati reali, calcolati da simulazione (mi sto riferendo a deviazione standard, strike, tempo a scadenza etc..). Inoltre, al termine della seduta di contrattazione, vi sono le formule che calcolano il rendimento ottenuto dall'operazione di mercato. Le potenzialità della classe *covered* potrebbero dare spazio alla stesura di un'intera altra tesi; mio malgrado, per mancanza di tempo, ho sfruttato solamente le sue potenzialità marginali.

TEST1

Premessa: agenti random, agenti covered

Come già spiegato, gli agenti *covered* assumono una posizione sul mercato per poi ricoprirla. A livello concettuale, un tipo di strategia simile viene fatta per cercare di ottenere un guadagno immediato dalla vendita delle opzioni e, nel caso di vendita dell'opzione OTM, per ottenere un extra rendimento, generato sia dalla vendita dei titoli ad un prezzo superiore a quello di acquisto (considerando l'andamento favorevole di mercato), sia dal prezzo di vendita delle opzioni. Nel caso di andamento contrario alle aspettative, invece, le perdite generate sulle azioni sono parzialmente coperte dai ricavi, generati dalla vendita delle opzioni, le quali non saranno esercitate. La strategia che si sta descrivendo può essere attuata da soggetti che, investendo nel comparto azionare, cercano di garantirsi limitati guadagni con una maggiore probabilità e parziali coperture sulle perdite.

Osserverò il fenomeno, che questa particolare categoria genera agendo nel mercato, da un punto di vista della simulazione.

L'effetto atteso riguarda improvvisi incrementi di prezzo, causati dall'intervento degli agenti *covered*, i quali, per come sono stati programmati, agiranno in *step* ravvicinati all'inizio di ogni giornata di contrattazione.

Visualizzazione dei risultati ottenuti

In Figura 29 è possibile osservare il risultato, generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 90
- numero agenti covered: 150
- numero step: 500
- numero cicli: 5

La scelta del numero di agenti *covered* è ricaduta forzatamente su un numero così elevato, poichè gli agenti di questa particolare categoria possono agire una sola volta per simulazione (per le cause spiegate in precedenza).

Come è possibile osservare dal grafico, le mie previsioni non erano errate. Nonostante l'effetto non sia così visibile e nonostante bisognerebbe osservare nei minimi dettagli tutte le variabili, a mia disposizione, preposte a questo fenomeno, l'entrata in azione degli agenti *covered*, che agiscono in step ravvicinati, crea movimenti di mercato. Inoltre, l'effetto simulato aiuta a capire come essi si esauriscano nei primi *step* dell'inizio di ogni giornata, dato l'elevato numero di step e la probabilità che gli agenti propongano un'offerta.

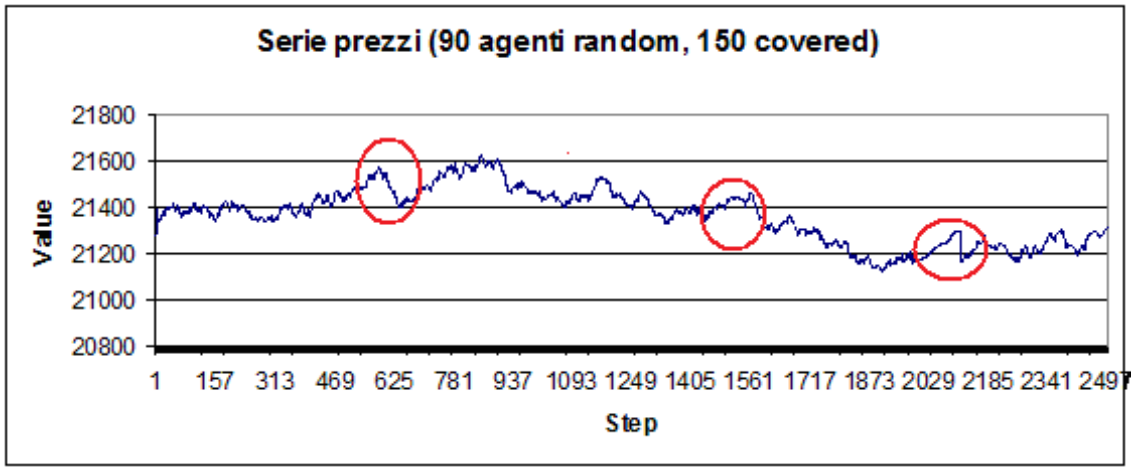


Figura 29 Serie di prezzi generati in un mercato popolato da 90 agenti *random* e 150 agenti *covered*

Una variabile che ritengo sia utile citare per osservare il fenomeno è quella preposta al conteggio di quanti agenti per classe, in ogni step, propongono e concludono transazioni ai prezzi da loro offerti. In questo modo, sarà così accertato che l'aumento dei movimenti di prezzo, in corrispondenza dell'inizio di ogni giornata di contrattazione (step 500-1000-1500-2000 ecc..), è provocato dagli agenti *covered*.

In Figura 30 è evidenziato il particolare riferito all'aumento dei prezzi, verificatosi all'inizio del quinto giorno di contrattazione.

Figura 30 Numero di agenti *covered* che, per step, fissano il prezzo di transazione attraverso le loro offerte

Giudizio conclusivo sperimentazioni

Nell'interazione tra agenti *covered* e agenti *variation* abbiamo ottenuto quanto atteso al momento della programmazione; quando essi si attivano, infatti, dal momento che propongono solamente offerte di acquisto e solitamente nei primi *step* di ogni giornata, conferiscono una spinta al rialzo del prezzo. In verità, l'effetto che si sarebbe potuto ottenere, nel caso in cui avessi programmato delle funzioni, che permettessero il conteggio del rendimento di più strategie per ogni singolo agente (conferendo maggior potere strategico agli agenti *covered*), sarebbe stato sicuramente più interessante e di maggior valenza dal punto di vista dello studio. Questo aspetto, ripreso anche nel prossimo test, sarà riportato nelle conclusioni, in relazione ai possibili futuri sviluppi della tesi.

TEST2

Premessa: agenti random, agenti covered, agenti variation

Nel seguente test saranno posti a confronto le tre categorie di agenti. Questo test rappresenta, dal mio punto di vista, uno dei più importanti sviluppi per un eventuale nuovo studio. La ragione per cui non è stato svolto prima è che durante i test sono state fatte molte scoperte; per questo motivo e, soprattutto, per la mancanza di tempo non è stato possibile potenziare ulteriormente questa classe, soprattutto dal punto di vista degli schemi di decisione. Se avessi compreso, con qualche mese di anticipo, il perfetto funzionamento degli agenti *variation* per ricreare andamenti di mercato storici, avrei dotato la classe covered, non solo delle sue capacità di calcolo, ma anche di un sistema per percepire l'andamento futuro di mercato, in base ai segnali ricevuti dalle simulazioni. Questo significa che se avessi programmato funzioni di previsione ancora più avanzate, avrei potuto ottenere ulteriori risultati, sopratutto dal punto di vista di calcolo dei rendimenti e creazione di strategie di trading efficienti.
Dal seguente test mi aspetto di osservare un comportamento simile a quello del test precedente; probabilmente, però, l'effetto sarà meno marcato, poichè l'intensità delle decisioni dei *covered* sarà mitigata dalle decisioni dei *variation*.

Visualizzazione dei risultati ottenuti

In Figura 31 è possibile osservare il risultato, generato dal comportamento aggregato delle tre categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti random: 80
- numero agenti covered: 150
- numero agenti variation: 10
- numero step: 500
- numero cicli: 5

Il fenomeno, presentato in Figura 31, è molto interessante perché è relativamente chiara l'azione dei *covered* agent. I punti di maggiore interesse, evidenziati nel grafico,

sono in particolare tre, in corrispondenza dei quali l'azione dei *covered* è piuttosto evidente.

In particolare, nel primo punto evidenziato non avviene il calo del trend, come, invece, in tutti i precedenti grafici di confronto. La motivazione è legata al fatto che le proposte di acquisto, avanzate in serie da parte dei *covered*, mitigano l'effetto.

Negli altri due punti evidenziati sul grafico si osserva la sommatoria degli effetti generati dagli agenti *variation* e da quelli *covered*. In entrambi i punti, che coincidono con l'inizio di giornate di contrattazione, è visibile un brusco movimento di prezzo. Come si può notare, in simulazione il fenomeno è maggiormente accentuato.

Figura 31 Confronto tra serie di prezzi simulati (agente *random, covered, variation*) e serie di valori Ftse

Dal quarto giorno di contrattazione in avanti, la serie di prezzi in simulazione risulta replicare esattamente i movimenti di mercato, pur essendo costantemente superiore rispetto al livello Ftse. La motivazione, emersa, osservando i dati, risiede nel fatto che da questo punto in avanti non ci sono variazioni di prezzo così brusche. Per questa ragione, il *range* di offerte, proposte dall'agente *variation,* si uniforma alle offerte proposte dagli altri partecipanti al mercato. Solamente in un caso la situazione potrebbe variare e permettere alle due distribuzioni di incrociarsi o di avvicinarsi; tuttavia, questo è possibile solamente se tutto il mercato segue il calo di prezzo dopo il boom, intorno allo step 2000. Per esempio, la situazione che sto descrivendo è avvenuta quando nella simulazione erano presenti anche gli agenti volume a comporre

il mercato (Figura 27) e al crash, successivo al boom, si era unita la quasi totalità degli agenti volume, che avevano accentuato l'effetto e cambiato il corso della distribuzione.

Giudizio conclusivo sperimentazioni

Gli effetti riscontrati dall'interazione dei tre agenti hanno riportato risultati soddisfacenti, perché anche in questo caso l'azione dell'agente, inserito assieme agli agenti *random* e *variation*, ha generato un effetto ben riscontrabile sulla distribuzione. Oltre al movimento di prezzo accentuato, in corrispondenza dell'inizio di ogni giornata di contrattazione, reputo importante il fenomeno osservato dallo *step* 2000 in avanti. L'analisi delle variabili, fatte in questa sezione, mi ha permesso di capire cosa provoca la costante superiorità dei prezzi simulati rispetto ai prezzi Ftse, dallo step 2000 in avanti; fenomeno per altro già osservato in test precedenti (riferimento Figura 23). Come già accennato nella premessa di questo test e nelle conclusioni del precedente test, la particolare categoria *covered* rappresenta una delle maggiori possibilità di sviluppo della mia tesi.

7.9 TEST NUMERO 6: MERCATO POPOLATO DA AGENTI ZERO INTELLIGENCE, BEST OFFER AGENTS E VARIATION PRICE REAL AGENTS

In questo paragrafo saranno presentati i risultati generati dall'interazione tra agenti *random, best offer agent e variation agent*. Come in tutti i precedenti test, le due categorie *variation* e *random* non modificano le loro funzioni. Per quanto riguarda invece gli agenti *best offer* di seguito enuncerò le caratteristiche generiche:

- *best offer agent*: rappresentano una particolare categoria di agenti, che agiscono, acquistano o vendono con certezza massima. La ragione di ciò è che sono programmati in modo da poter visualizzare il massimo prezzo di acquisto presente in lista; inoltre, nel caso in cui essi desiderino acquistare, propongono un prezzo ancora superiore, in modo da avere la certezza di poter concludere una transazione nell'immediato istante in cui la loro offerta giunge al Book. Nel

caso in cui vogliano vendere, essi propongono un'offerta di vendita inferiore alla migliore, elencata in lista. Anche in questo caso, l'offerta sarà così inferiore da permettere, con certezza massima, la vendita del titolo nell'immediato istante in cui l'offerta perviene al Book.

TEST1

Premessa: agenti random, agenti best offer

Ciò che mi aspetto di ottenere dall'interazione di queste due categorie di agenti è una serie di picchi di prezzo sia positivi sia negativi. Considerando l'entità dei possibili prezzi offerti dall'agente *best offer* e il numero di offerte che, all'interno di una simulazione di 2500 *step* sono proposte, l'aspettativa avanzata mi parrebbe più che fondata. La ragione per cui, invece, questo ragionamento è errato riguarda il fatto che, a causa della modalità con cui nel mercato avvengono realmente le transazioni, analogamente, nel mio esperimento, esse si verificano solo se coincidono le condizioni tra domanda e offerta, ma il regolamento di prezzo si basa sulla prima delle due offerte che già era presente in lista. Quindi, il *best offer* concluderà sempre un contratto quando opererà, ma la probabilità che ha di veder conclusa una transazione con prezzo ufficiale l'offerta da lui proposta, è praticamente nulla. Per questa ragione, mi aspetto che le serie di prezzi nel mio mercato simulato si distribuiscano secondo perfetto *random walk,* solamente con una varianza superiore a quella del precedente grafico che rappresentava un *random walk,* mostrato all'inizio di questo capitolo.

Visualizzazione dei risultati ottenuti

In Figura 32 è possibile osservare il risultato, generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti *random*: 90
- numero agenti *best offer*: 10
- numero step: 500

- numero cicli: 5

Come è possibile osservare in Figura 32, ciò che ho ottenuto rappresenta esattamente quanto previsto e una situazione molto simile a quella visibile in Figura 1. Per cercare di mettere in risalto i picchi di prezzo, ho diminuito l'intervallo di scala sull'asse y.

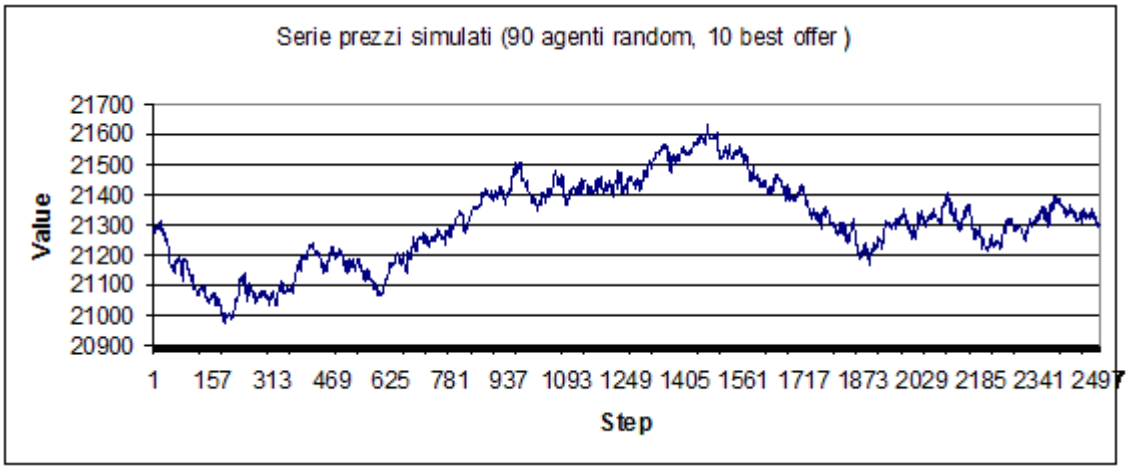


Figura 32 Serie di prezzi generati in un mercato popolato da 90 agenti *random* e 150 agenti *covered*

Poichè il fenomeno osservato è stato perfettamente spiegato nella premessa e verificato nel grafico, credo che per capire e spiegare il fenomeno sia importante verificare il contrario di quanto detto. L'obiettivo, quindi, è quello di ricreare una simulazione, in cui il numero dei *best offet* sia superiore a prima, per aumentare la probabilità che prezzi ufficiali di transazione siano fissati dalle offerte da loro avanzate e di visualizzare una maggiore volatilità di prezzo rispetto a quella presentata nel grafico in Figura 32.

Inserendo, quindi, un numero superiore di agenti *best offer* è possibile che la distribuzione risulti caratterizzata da maggiori picchi, rispetto a quanto già visto. Tale fenomeno è visualizzabile in Figura 33.

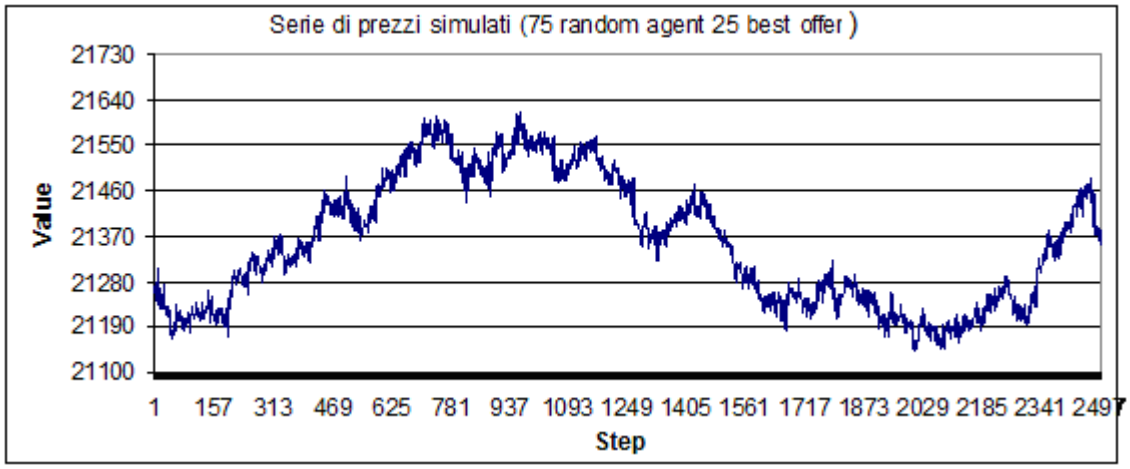


Figura 33 Serie di prezzi simulati generati da 75 agenti *random* e 25 *best offet*.

Giudizio conclusivo sperimentazioni

Dai test proposti in questa sezione, si può concludere che l'effetto generato dall'interazione tra agenti *random* e *best offer* è limitato dal funzionamento del Book di negoziazione. Come anticipato in premessa, in questa particolare serie di test si è verificata una non coincidenza tra il fenomeno che mi aspettavo di osservare e ciò che realmente si è verificato. Considero questo aspetto molto importante riguardante la condizione in cui il pensiero del programmatore, nell'ambito degli ABM, può muoversi in un senso opposto rispetto all'effetto ottenuto dal programma. Pur essendo il mio un semplice errore di valutazione facilmente riconducibile al regolamento del sistema di contrattazione del Book, a livello pratico, credo che il test con i *best agent* mi abbia permesso di comprendere uno dei pregi di questa metodologia quale la non coincidenza tra aspettative e verifiche.

TEST2

Premessa: agenti random, agenti best offer, agenti variation

In questa serie di test si osserveranno gli effetti generati dall'interazione di queste tre categorie di agenti. Ciò che mi aspetto, considerando l'esperienza avuta testando le interazioni tra soli agenti *random* e *best offer* è quella di ottenere una serie di prezzi che ripetono le distribuzioni simulate che già avevo testato facendo interagire i

variation e i *random,* con la differenza di visualizzare una varianza maggiore tra i prezzi.

Visualizzazione dei risultati ottenuti

In Figura 34 è possibile osservare il risultato generato dal comportamento aggregato delle tre categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti *random*: 80
- numero agenti *best offer*: 10
- numero agenti *variation*: 10
- numero step: 500
- numero cicli: 5

Il fenomeno generato dall'interazione di queste tre categorie di agenti risulta essere molto interessante. Cercando di dare una pura descrizione del fenomeno si può dire che le due distribuzioni sono molto simili e vicine nei primi *step,* per poi distaccarsi e ritrovarsi in corrispondenza della terza giornata di contrattazione. Voglio ricondurre l'effetto che ho osservato ad un precedente ragionamento proposto quando si trattavano i risultati delle simulazione generati dall'interazione tra *random*, *volume* e *variation* in cui si è riscontrato un fenomeno analogo.

In quel caso, in corrispondenza dello *step* 1000 si era verificato un allontanamento tra la distribuzione simulata e la distribuzione Ftse, analogo all'allontanamento osservabile in Figura 34, causato dall'attivazione degli agenti *volume* che avevano modificato il corso delle serie dei prezzi e, complice la fase di stabilità dei valori Ftse, avevano impedito ai *variation* di contrastare la forza delle offerte degli agenti antagonisti.

In realtà, ciò che è accaduto è la prova che dimostra che il mio ragionamento è esatto. Ora cercherò di spiegare la mia tesi.

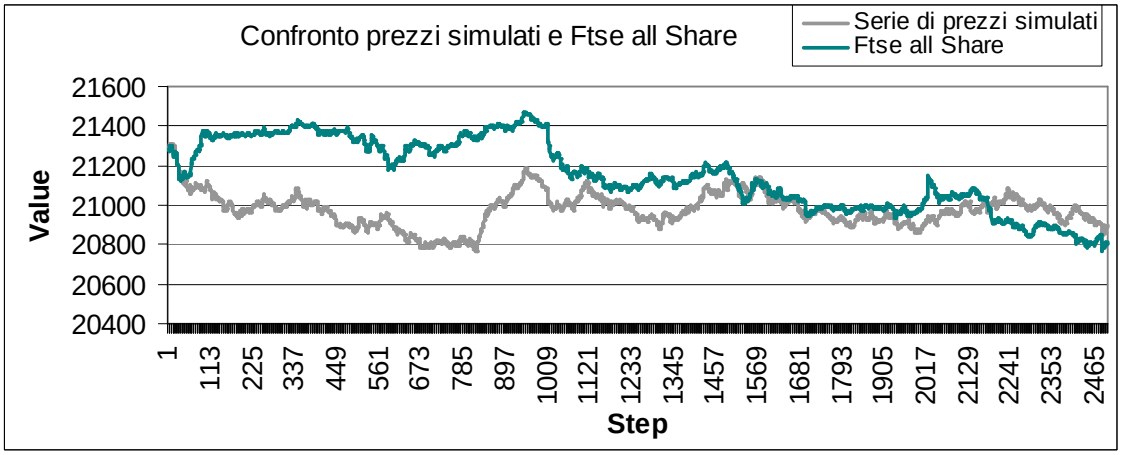


Figura 34 Confronto tra serie di prezzi simulati (agente *random*, *best offer*, *variation*) e serie di valori Ftse

La distribuzione dei valori Ftse può essere suddivisa in tre differenti fasi, come visibile da Figura 35, in cui l'azione dei *variation* condiziona fortemente la distribuzione simulata. Nel caso in cui, in corrispondenza di questi punti, non è l'azione dei *variation* a prevalere sul mercato ma l'azione di qualche altro agente per cause che possono essere le più varie, il corso della distribuzione diventa indipendente dall'agente che basa la sua strategia sulle variazione del Ftse. La ragione è che, pochi sono i punti in cui è avvenuto un brusco cambio di prezzo e quindi di conseguenza le offerte proposte dall'agente *variation* escono dal *range* medio di offerte condizionando la distribuzione.
I rettangoli rappresentati in figura devono essere letti in questo modo, secondo il mio ragionamento:

- i due lati verticali dei tre rettangoli indicano il cambio di prezzo che potrebbe, secondo la mia visione, riportare le due distribuzioni di valori ad essere vicine
- i lato orizzontale dei tre rettangoli rappresenta la fase in cui i prezzi Ftse, non essendo particolarmente volatili, non causano, attraverso gli agenti *variation*, effetti determinanti che fortemente condizionano la distribuzione simulata

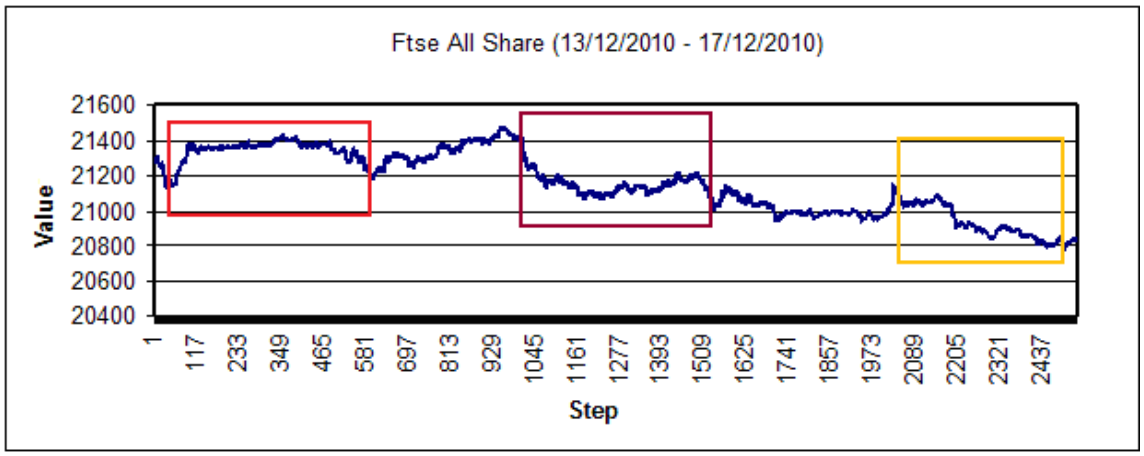


Figura 35 Analisi riguardante i punti di maggiore interesse relativi a cambio di prezzo per serie di valori Ftse

Per dimostrare con dati della simulazione quanto detto è necessario visualizzare la variabile preposta per il conteggio degli spread nella lista opposta a quella visualizzata nel caso dei *volume*. Se prima infatti si era presa di riferimento la BuyList, perché si era verificato un aumento dei prezzi simulati, ora si osserverà la SellList, perché si è osservato una diminuzione. Inoltre, ricordo che il valore degli spread pari a zero, significa che la lunghezza della lista è nulla, quindi, oltre che la teoria degli spread attraverso l'osservanza di questa particolare variabile si incrocia la teoria che attribuisce alla lunghezza delle lista possibili cambiamenti di prezzo.

Il particolare relativo ad un numero limitato di *step*, presenti in Figura 36, è rappresentativo della fase in cui, dopo il primo calo di prezzo registrato anche dalla serie Ftse, le due serie di valori si sono allontanate. Chiaramente, la causa è attribuibile all'operato degli agenti *best offer* i quali intervenendo, hanno permesso l'inversione del corso della distribuzione permettendo transazioni a prezzi fuori dal normale corso del mercato.

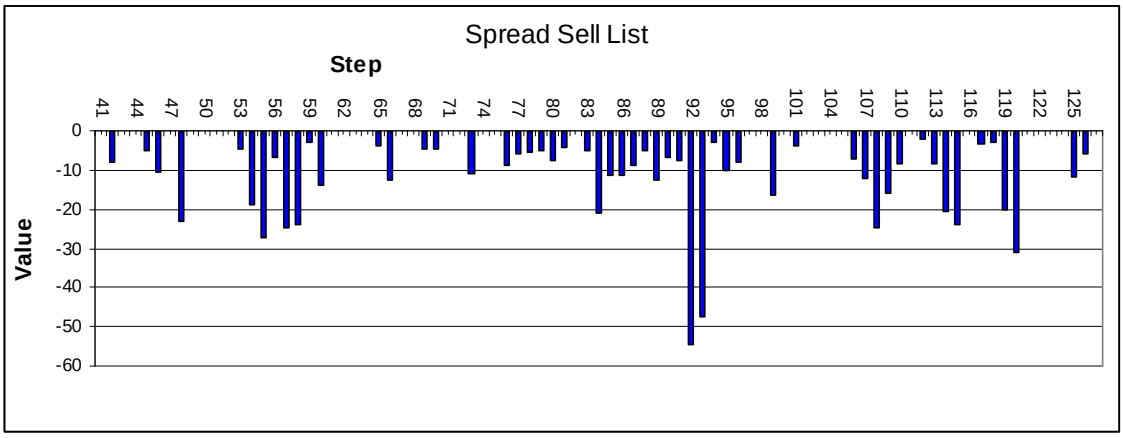


Figura 36 Spread per step calcolato tra il migliore e il peggiore prezzo di vendita, presenti in lista

Giudizio conclusivo sperimentazioni

Dopo le sperimentazioni eseguite sono arrivato a concludere che una delle maggiori caratteristiche messe in risalto dalla categoria di agenti *best offer* riguarda la possibilità che prezzi fuori mercato proposti dagli agenti *random* diventino prezzi ufficiali di transazione. Attraverso questo processo gli agenti *random* assumono il potere di condizionare maggiormente, attraverso le loro offerte, il mercato simulato.

Altra conclusione è relativa alla maggior comprensione delle caratteristiche dell'agente *variation* e della distribuzione dei valori Ftse che mi hanno permesso di dimostrare ulteriormente che solamente gli agenti che propongono offerte fuori dal *range* medio possono condizionare e guidare il mercato. Inoltre, se la "competizione" tra agenti intelligenti non è "vinta" dall'agente *variation* nei punti che ho bene indicato in Figura 35, la serie di prezzi simulati si discosta fortemente dalla distribuzione Ftse originando un *random walk*.

7.9 TEST NUMERO 7: MERCATO POPOLATO DA AGENTI ZERO INTELLIGENCE,TREND AGENT E VARIATION PRICE REAL AGENTS

In questa serie di test si vuole osservare il comportamento, che emerge dalle interazioni di queste tre categorie di agenti. Come è avvenuto per tutti gli esperimenti, fino ad ora proposti, i test saranno principalmente due; in essi la nuova categoria di

agenti sarà prima inserita nel mercato in competizione con i soli agenti *random*, per poi essere posta in competizione sia con i *random* sia con i *variation*. Come in tutti i test, le categorie *random* e *variation* non subiranno modifiche. Di seguito, spiegherò brevemente le caratteristiche della nuova categoria di agenti introdotta.

- *trend agent:* la strategia di azione con cui questi agenti sono programmati si basa sulla capacità di calcolare il trend di mercato nell'istante *t* e di agire di conseguenza, assumendo una posizione sul mercato. Il numero di prezzi storici, a cui l'agente fa riferimento per calcolare il *trend,* è variabile da un minimo di quattro prezzi ad un massimo di tutti i prezzi di transazione avvenuti nel mercato.

TEST1

Premessa: agenti random, agenti trend

L'effetto che mi aspetto di ottenere dall'interazione di queste due categorie di agenti è una serie di prezzi che mostrano distribuzioni caratterizzate da lunghi trend. La ragione che mi spinge a prevedere questo effetto è principalmente la funzione, di cui ho dotato volutamente la classe *trend agent*, e che consente loro di esprimere offerte nello stesso *range* di prezzo dei *random* agent. Questo dovrebbe impedire che si creino degli sbalzi di prezzo e dovrebbe permettere la visualizzazione di trend, in crescita o decrescita, pressoché costanti.

Visualizzazione dei risultati ottenuti

In Figura 37 è possibile osservare il risultato, generato dal comportamento aggregato delle due categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti *random*: 90
- numero agenti *trend*: 10
- numero step: 500
- numero cicli: 5

Come è possibile osservare dal grafico in Figura 37, le previsioni sull'andamento di prezzo, che la simulazione ha generato, non sono state errate. Inoltre, un effetto inaspettato è la nitidezza con cui il grafico mostra un serie di tre trend alternati di segno. Il fenomeno osservato mostra chiaramente il perfetto funzionamento degli agenti *trend,* i quali, inserendo dei prezzi di transazione nell'ordine degli agenti *random*, permettono di non determinare sbalzi di prezzo e andamenti costanti. In realtà, durante i primi *step*, la varianza di prezzo è piuttosto elevata rispetto alle serie di prezzi che si distribuiscono dallo *step* 1000 in avanti. La ragione è legata al fatto che, volutamente, ho programmato una funzione interna ai *trend agent* che consentisse loro di entrare nel mercato solamente dopo lo *step* numero 150. In questo modo, la loro strategia, si sarebbe potuta basare su un numero di prezzi *random,* che avrebbe aumentato per i primi *step*, circa 500, la variabilità delle loro decisioni.

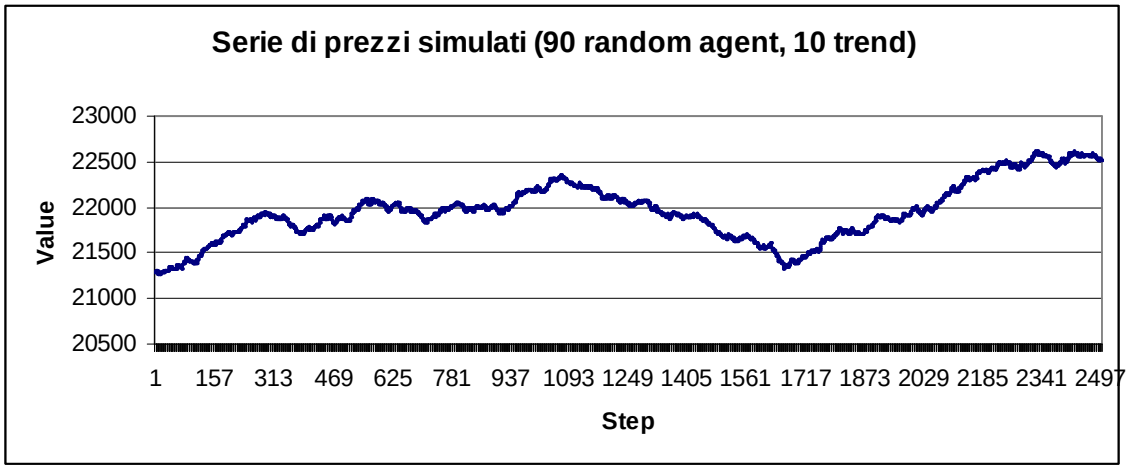


Figura 37 Serie di prezzi generati in un mercato popolato da 90 agenti *random* e 10 agenti *trend* attivi dopo un certo numero di step

In Figura 38 è raffigurato un particolare relativo ad un numero limitato di *step,* in cui gli agenti trend hanno la possibilità di entrare in gioco già dai primi istanti. Impostando quindi una strategia di offerte al rialzo, secondo il trend crescente, essi condizioneranno fortemente la distribuzione. Il motivo riguarda la modalità con cui gli agenti vengono interrogati, proponendo offerte con elevata probabilità di essere del medesimo segno (acquisto o vendita). Questo genera una forte dipendenza nelle serie dei prezzi simulati.

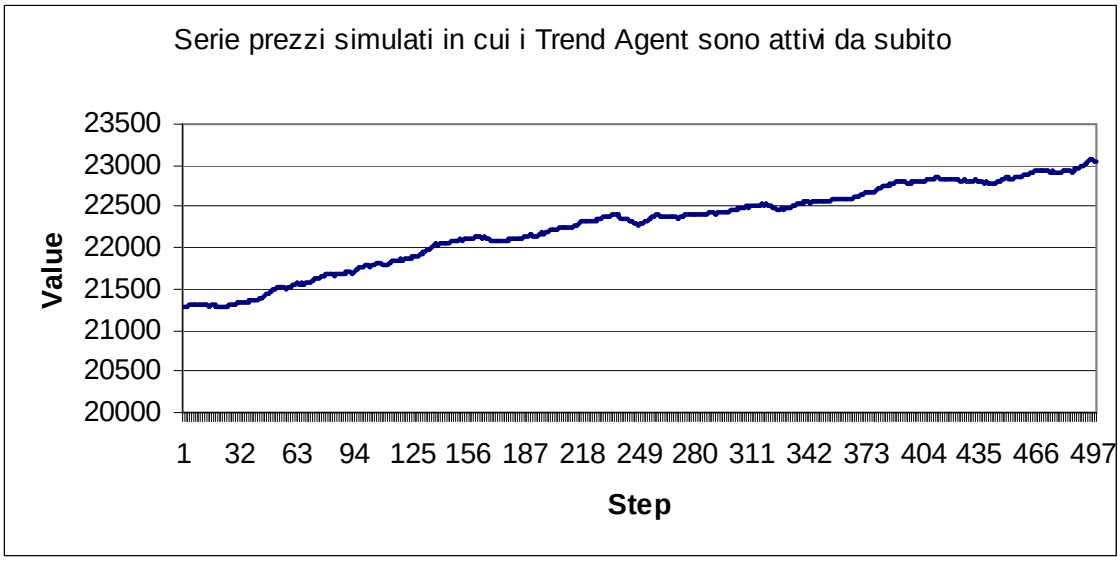


Figura 38 Serie di prezzi generati in un mercato popolato da 90 agenti *random* e 10 agenti *trend* attivi dopo un certo numero di step

Giudizio conclusivo sperimentazioni

Il risultato ottenuto osservando la simulazione, in cui la categoria di agenti *trend* ha agito in competizione con la categoria di agenti *random,* è molto interessante, soprattutto se si considera l'esempio in cui gli agenti intelligenti si attivano dopo un certo numero di *step*. La dipendenza che deriva, in questo caso, dal numero di offerte proposte in un senso, ad esempio l'acquisto, (rispetto alle precedenti dipendenze osservate, causate dalla possibilità di alcuni agenti di presentare prezzi al di fuori del *range* medio), ci permette di apprezzare una nuova qualità emersa dal nostro simulatore, quale il potere che alcuni agenti hanno di condizionare la distribuzione, agendo sempre nel medesimo verso.

TEST2

Premessa: agenti random, agenti trend offer, agenti variation

Dai risultati ottenuti nel test precedente e dal *range* di offerte che possono essere proposte dalla categoria di *trend agent*, mi aspetto che il mercato sia guidato dai *variation agent* i quali, seguendo la mia teoria, hanno più potere di mercato e, quindi,

decideranno le sorti dello andamento dei prezzi. L'unica distorsione, che in realtà mi aspetto, riguarda un aumento della lunghezza dei trend, dovuta all'azione dei *trend agent*.

Visualizzazione dei risultati ottenuti

In Figura 39 è possibile osservare il risultato generato dal comportamento aggregato delle tre categorie di agenti. La simulazione è stata programmata in questo modo:

- numero agenti *random*: 80
- numero agenti *trend*: 10
- numero agenti *variation*: 10
- numero step: 500
- numero cicli: 5

Come previsto, la serie di prezzi simulati segue l'andamento determinato dalle offerte degli agenti *variation.* E' interessante notare il momento in cui gli agenti trend si attivano. In corrispondenza dello step 160, infatti, le offerte espresse dagli *agenti trend* sono ben visibili attraverso l'aumento dei prezzi, rispetto al normale corso seguito dai Ftse. Inoltre, le transazioni in questa fase si sono concluse ad intervalli di prezzo molto ravvicinati.

Per la restante distribuzione sono ben visibili cambi di prezzo, che certamente sono stati determinati dalla volontà dei *variation*; per quanto riguarda le fasi di trend, si sono registrati dei trend allungati, come già indicato nella premessa.

La difficoltà nell'analizzare le variabili, cercando di ottenere spiegazioni dell'effetto ben visibile in Figura 39, ci impedisce di approfondire l'analisi. La causa è legata al fatto che non sono presenti sbalzi di prezzo. Gli unici effetti registrati riguardano i *trend* in simulazione, i quali sono più lunghi rispetto alla distribuzione Ftse. Dalle variabili rilevate e dai dati di cui dispongo, tale effetto non è così tangibile come appare dal grafico.

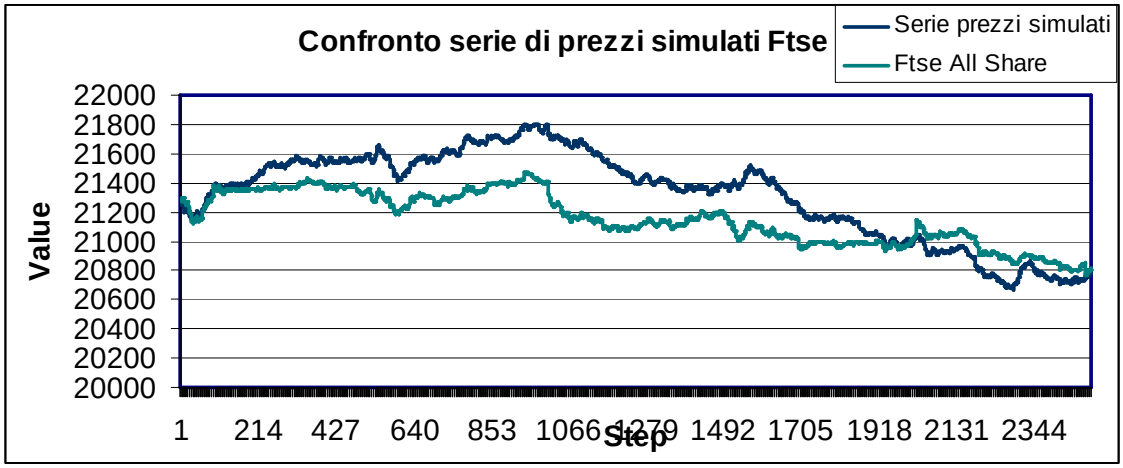


Figura 39 Confronto tra serie di prezzi simulati (agente *random, trend agent, variation*) e serie di valori Ftse

Giudizio conclusivo sperimentazioni

Come è stato dimostrato dal test numero 1 e dal test numero 2, in cui gli agenti *trend* sono entrati in competizione con il mercato, l'impronta data dalla loro azione sulla distribuzione dei prezzi è stata tangibile. La difficoltà nel visualizzare variabili, che effettivamente descrivano l'effetto rappresentato in Figura 39, non ha consentito di approfondire, come nel caso degli altri test, le motivazioni legate agli effetti generati dalle interazioni. L'unica considerazione che si può fare è che l'azione dei *trend agent* rende le fasi di *trend* di mercato più accentuate.

7.10 TEST NUMERO 8: MERCATO POPOLATO DA TUTTI GLI AGENTI PROGRAMMATI NEL SIMULATORE

Questo test, non essendo stato calibrato, appare più una provocazione che un vero e proprio test. Ciò che ci si aspetta, considerando che le due categorie Ftse sono state inserite all'interno del mercato, è che la distribuzione della serie dei prezzi segua quella del Ftse.

- numero agenti *random*: 90
- numero agenti *trend*: 5

- numero agenti *variation*: 5
- numero agenti *level*: 5
- numero agenti *covered*: 5
- numero agenti *volume*: 5
- numero agenti *best*: 5
- numero step: 500
- numero cicli: 5

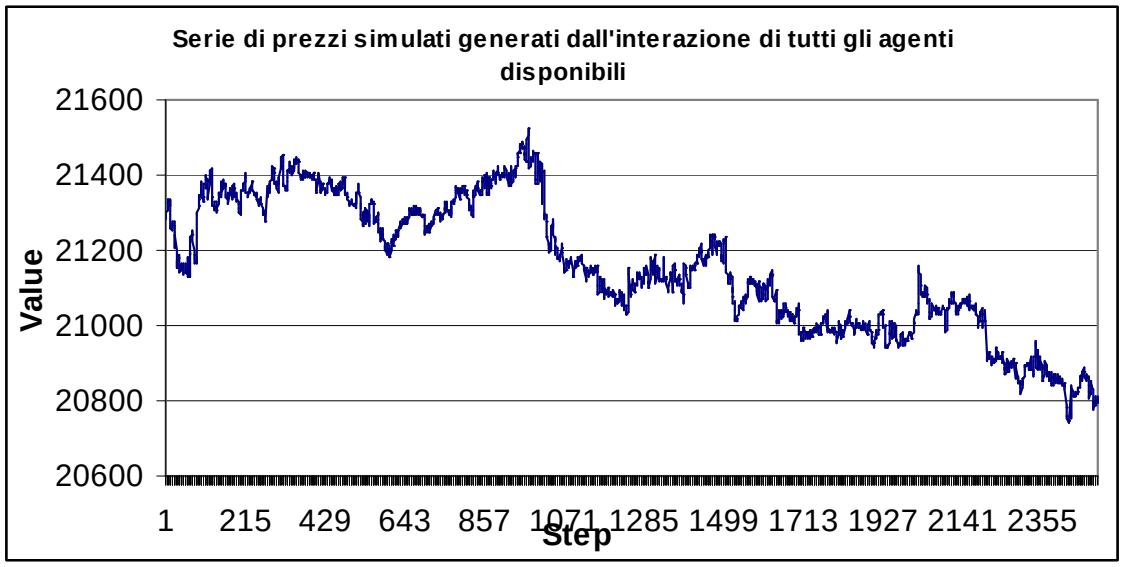


Figura 40 Serie di prezzi simulati in un mercato in cui hanno partecipato tutti gli agenti programmati

Capitolo 8

CODICE DI PROGRAMMAZIONE DEL SIMULATORE

In questo capitolo è riportato il codice Python del programma ufficiale con cui ho svolto i test di simulazione. Per la spiegazione dettagliata relativa alle classi e alle funzioni, fare riferimento al manuale utente riportato nel capitolo sesto. I file sono stati riportati in ordine alfabetico.

8.1 FILE: Action Group

```
#ActionGroup.py

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

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

8.2 FILE: Best Offert Agent

```
#BestOffertAgent.py

import VolumeAgents
import random

class BestOffertAgent(VolumeAgents.VolumeAgent):
    def __init__(self,number,priceList,aBook):

        VolumeAgents.VolumeAgent.__init__(self,number,priceList,aBook)

        self.number = number
        self.assets = 0
        self.type = "BestOffertAgent"
        self.aBook = aBook
        self.priceList = priceList

    def bestTransaction(self):
        self.sellPrice = 0
        self.buyPrice = 0
        self.minimunPrice = self.priceList[-1]
        self.transactionList = self.aBook.getTransactionList()
        self.buyList = self.aBook.getBuyList()
        self.sellList = self.aBook.getSellList()
        self.sellPriceMin = self.aBook.getMaxPriceBuyList()
        self.buyPriceMax = self.aBook.getMaxPriceSellList()

        if len(self.transactionList)>2:
            if self.transactionList[-1] < self.transactionList[-2]:
                if len(self.buyList) > 0:
                    self.sellPrice = self.sellPriceMin - 20
                    #print "self.sellPriceMin",
self.sellPriceMin,self.sellPrice
                    #print
"agent.BestOffert=",self.number,"SellPrice","%.2f" % self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])
            if self.transactionList[-1] > self.transactionList[-2]:
                if len(self.sellList) > 0:
                    self.buyPrice = self.buyPriceMax + 20
                    #print
"self.buyPriceMax",self.buyPriceMax,self.buyPrice
                    #print
"agent.BestOffert=",self.number,"BuyPrice","%.2f" % self.buyPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])


    def setExecutedBuyOrder(self):
        self.assets+=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets
```

8.3 FILE: Book

```
#Book

import math

class Book:
    def __init__(self,priceList):
        self.priceList=priceList
        self.buyList=[]
        self.sellList=[]
        self.finalPriceForStep=[]
        self.transactionList = []
        self.volTransaction=0
        self.lenSellList= []
        self.lenBuyList=[]
        self.spreadBuy = 0
        self.spreadBuyList = []
        self.spreadSell = 0
        self.spreadSellList = []
        self.randomAgentOffert=0
        self.randomAgentOffertList=[]
        self.levelAgentOffert=0
        self.levelAgentOffertList=[]
        self.bestAgentOffert=0
        self.bestAgentOffertList=[]
        self.coveredAgentOffert=0
        self.coveredAgentOffertList=[]
        self.trendAgentOffert=0
        self.trendAgentOffertList=[]
        self.variationAgentOffert=0
        self.variationAgentOffertList=[]
        self.volumeAgentOffert=0
        self.volumeAgentOffertList=[]

    def setBuyDecision(self,information):
        price = information[0]
        identity = information[1]
        categoryAgent = information[2]

        if len(self.sellList)>0:
            self.sellList2 = []
            for i in self.sellList:
                if identity != i[1]:
                    self.sellList2.append(i)
            self.sellList = self.sellList2[:]

        if len(self.buyList)>0:
            self.buyList3 = []
            for i in self.buyList:
                if identity != i[1]:
                    self.buyList3.append(i)
            self.buyList = self.buyList3[:]

        if len(self.sellList)==0:
            self.buyList.append(information)
```

```
            #self.setLenBuyList()
            self.buyList.sort(reverse = True)
        elif len(self.sellList)>0:
                if price < self.sellList[0][0]:
                    self.buyList.append(information)
                    #self.setLenBuyList()
                    self.buyList.sort(reverse=True)
                else:
                    self.getVolTransaction()
                    firstInformationInList=self.sellList.pop(0)
                    self.priceList.append(firstInformationInList[0])
                    #print "TRADING PRICE","%.2f"
firstInformationInList[0]
                    information[1].setExecutedBuyOrder()
                    firstInformationInList[1].setExecutedSellOrder()
                    if firstInformationInList[2] == "RandomAgent":
                        self.randomAgentOffert+=1
                     #   print
firstInformationInList[2],"self.randomAgentOffert",
self.randomAgentOffert
                    if firstInformationInList[2] ==
"LevelPriceRealDataAgent":
                        self.levelAgentOffert+=1
                     #   print firstInformationInList[2],
"self.levelAgentOffert", self.levelAgentOffert
                    if firstInformationInList[2] == "BestOffertAgent":
                        self.bestAgentOffert+=1
                     #   print firstInformationInList[2],
"self.bestAgentOffert", self.bestAgentOffert
                    if firstInformationInList[2] == "CoveredAgent":
                        self.coveredAgentOffert+=1
                     #   print firstInformationInList[2],
"self.coveredAgentOffert", self.coveredAgentOffert
                    if firstInformationInList[2] == "trendAgent":
                        self.trendAgentOffert+=1
                     #   print firstInformationInList[2],
"self.levelAgentOffert", self.trendAgentOffert
                    if firstInformationInList[2] ==
"VariationPriceRealDataAgent":
                        self.variationAgentOffert+=1
                     #   print firstInformationInList[2],
"self.levelAgentOffert", self.variationAgentOffert
                    if firstInformationInList[2] == "VolumeAgent":
                        self.volumeAgentOffert+=1
                    #   print firstInformationInList[2],
"self.levelAgentOffert", self.volumeAgentOffert

      #  print "Buy List", ",".join("%.2f" % price[0] for price in
self.buyList)
      #  print "Sell List", ",".join("%.2f" % price[0] for price in
self.sellList)

    def setSellDecision(self,information):

        price = information[0]
        identity = information[1]
        categoryAgent = information[2]

        if len(self.buyList)>0:
```

```
            self.buyList2=[]
            for i in self.buyList:
                if identity != i[1]:
                    self.buyList2.append(i)
            self.buyList = self.buyList2[:]

        if len(self.sellList)>0:
            self.sellList3 = []
            for i in self.sellList:
                if identity != i[1]:
                    self.sellList3.append(i)
            self.sellList = self.sellList3[:]

        if len(self.buyList)==0:
            self.sellList.append(information)
            self.sellList.sort()
            #self.setLenSellList()
        elif len(self.buyList)>0:
            if price > self.buyList[0][0]:
                self.sellList.append(information)
                #self.setLenSellList()
                self.sellList.sort()
            else:
                self.getVolTransaction()
                firstInformationInList=self.buyList.pop(0)
                self.priceList.append(firstInformationInList[0])
           #     print "TRADING PRICE","%.2f" %
firstInformationInList[0]
                firstInformationInList[1].setExecutedBuyOrder()
                information[1].setExecutedSellOrder()
           #     print firstInformationInList[2]
                if firstInformationInList[2] == "RandomAgent":
                    self.randomAgentOffert+=1
           #         print firstInformationInList[2],
"self.randomAgentOffert",self.randomAgentOffert
                if firstInformationInList[2] ==
"LevelPriceRealDataAgent":
                    self.levelAgentOffert+=1
           #         print firstInformationInList[2],
"self.levelAgentOffert", self.levelAgentOffert
                if firstInformationInList[2] == "BestOffertAgent":
                    self.bestAgentOffert+=1
           #         print firstInformationInList[2],
"self.levelAgentOffert", self.levelAgentOffert
                if firstInformationInList[2] == "CoveredAgent":
                    self.coveredAgentOffert+=1
           #         print firstInformationInList[2],
"self.levelAgentOffert", self.levelAgentOffert
                if firstInformationInList[2] == "trendAgent":
                    self.trendAgentOffert+=1
           #         print firstInformationInList[2],
"self.levelAgentOffert", self.trendAgentOffert
                if firstInformationInList[2] ==
"VariationPriceRealDataAgent":
                    self.variationAgentOffert+=1
           #         print firstInformationInList[2],
"self.levelAgentOffert", self.variationAgentOffert
                if firstInformationInList[2] == "VolumeAgent":
                    self.volumeAgentOffert+=1
```

```
           #         print firstInformationInList[2],
"self.levelAgentOffert", self.volumeAgentOffert

     #   print "Sell List", ",".join("%.2f" % price[0] for price in
self.sellList)
     #   print "Buy List", ",".join("%.2f" % price[0] for price in
self.buyList)

    def cleanLists(self):
        del self.buyList[:]
        del self.sellList[:]
        del self.priceList[:-1]
        #del self.finalPriceForStep[:]
        del self.transactionList[:]

    def cleanStep(self):
        self.volTransaction = 0
        self.randomAgentOffert = 0
        self.levelAgentOffert = 0
        self.bestAgentOffert = 0
        self.coveredAgentOffert = 0
        self.trendAgentOffert = 0
        self.variationAgentOffert = 0
        self.volumeAgentOffert = 0

    def setVolumeAgentOffert(self):
        self.volumeAgentOffertList.append(self.volumeAgentOffert)

    def getVolumeAgentOffert(self):
        return self.volumeAgentOffertList

    def setVariationAgentOffert(self):

self.variationAgentOffertList.append(self.variationAgentOffert)

    def getVariationAgentOffert(self):
        return self.variationAgentOffertList

    def setTrendAgentOffert(self):
        self.trendAgentOffertList.append(self.trendAgentOffert)

    def getTrendAgentOffert(self):
        return self.trendAgentOffertList

    def setCoveredAgentOffert(self):
        self.coveredAgentOffertList.append(self.coveredAgentOffert)

    def getCoveredAgentOffert(self):
        return self.coveredAgentOffertList

    def setBestAgentOffert(self):
        self.bestAgentOffertList.append(self.bestAgentOffert)

    def getBestAgentOffert(self):
        return self.bestAgentOffertList

    def setLevelAgentOffert(self):
        self.levelAgentOffertList.append(self.levelAgentOffert)
```

```
    def getLevelAgentOffert(self):
        return self.levelAgentOffertList

    def setRandomAgentOffert(self):
        self.randomAgentOffertList.append(self.randomAgentOffert)

    def getRandomAgentOffert(self):
        return self.randomAgentOffertList

    def setSpreadBuyOffert(self):
        self.spreadBuy1 = float(self.spreadBuy)
        if len(self.buyList)>0:
            self.spreadBuy1 = self.buyList[0][0] - self.buyList[-1][0]
            self.spreadBuyList.append(self.spreadBuy1)
        else:
            self.spreadBuyList.append(0)

    def getSpreadBuyOffert(self):
        return self.spreadBuyList

    def setSpreadSellOffert(self):
        self.spreadSell1 = float(self.spreadSell)
        if len(self.sellList)>0:
            self.spreadSell1 = self.sellList[0][0] - self.sellList[-
1][0]
            self.spreadSellList.append(self.spreadSell1)
        else:
            self.spreadSellList.append(0)

    def getSpreadSellOffert(self):
        return self.spreadSellList

    def setLenBuyList(self):
        self.lenBuyList.append(len(self.buyList))

    def setLenSellList(self):
        self.lenSellList.append(len(self.sellList))

    def getLenBuyList(self):
        return self.lenBuyList

    def getLenSellList(self):
        return self.lenSellList

    def getStartingPrice(self):
        print "price List", ','.join("%.2f" % price for price in
self.priceList)

    def getBuyList(self):
        return self.buyList

    def getSellList(self):
        return self.sellList

    def getMaxPriceSellList(self):
        self.maxSellPrice = self.priceList[-1]
        if len(self.sellList)>0:
            self.maxPriceSellList = self.sellList[0][0]
            return self.maxPriceSellList
```

```
        else:
            return self.maxSellPrice - 20

    def getMaxPriceBuyList(self):
        self.maxBuyprice = self.priceList[-1]
        if len(self.buyList)>0:
            self.maxPriceBuyList = self.buyList[0][0]
            return self.maxPriceBuyList
        else:
            return self.maxBuyprice + 20

    def setFinalPriceForStep(self):
        self.finalPriceForStep.append(self.priceList[-1])
        #print "end_Of_StepPrice", ','.join("%.2f" % price for price
in self.finalPriceForStep)

    def getFinalPriceForStep(self):
        return self.finalPriceForStep

    def getVolTransaction(self):
        self.volTransaction+=1

    def setTransactionList(self):
        self.transactionList.append(self.volTransaction)
     #   print self.transactionList

    def getTransactionList(self):
        return self.transactionList
```

8.4 FILE: Bs Calculation

```
#BsCalculation.py

import math
import numpy

def erfcc(x):
    z = abs(x)
    t = 1. / (1. + 0.5*z)
    r = t * math.exp(-z*z-1.26551223+t*(1.00002368+t*(.37409196+
        t*(.09678418+t*(-.18628806+t*(.27886807+
        t*(-1.13520398+t*(1.48851587+t*(-.82215223+
        t*.17087277)))))))))
    if (x >= 0.):
        return r
    else:
        return 2. - r

def ncdf(x):
    return 1. - 0.5*erfcc(x/(2**0.5))


def d1(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
```

```
    return (math.log(S01/K1) + (r + sigma**2 / 2) * T) / (sigma *
math.sqrt(T))


def d2(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return (math.log(S01/K1) + (r - sigma**2 / 2) * T) / (sigma *
math.sqrt(T))

def BlackScholesEuropeanCallPrice(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return S0 * ncdf(d1(S0, K, r, sigma, T)) - K * math.exp(-r * T) *
ncdf(d2(S0, K, r, sigma, T))

def BlackScholesEuropeanPutPrice(S0, K, r, sigma, T):
    T0 = float(T)
    S01 = float(S0)
    K1 = float(K)
    return K * math.exp(-r * T) * ncdf(-d2(S0, K, r, sigma, T)) - S0 *
ncdf(-d1(S0, K, r, sigma, T))
```

8.5 FILE: Covered Agent

```
#CoveredAgent.py

import BsCalculation
import random
import math
import numpy

class CoveredAgent:
    def __init__(self,number,priceList,aBook,nSteps,numCycles):
        self.number=number
        self.numCycles = numCycles
        self.nSteps = nSteps
        self.assets = 0
        self.type = "CoveredAgent"
        self.aBook=aBook
        self.priceList=priceList
        self.cash = 100000
        self.active = 0
        self.numberSteps = 0
        self.numberCycles = 0
        self.flag = 0
        self.K = 0
        self.buyPriceCall = 0
        self.buyPricePut = 0
        self.KCall = 0
        self.KPut = 0
        self.buyOffertListCall = []
        self.buyOffertListPut = []
```

```
        self.KCallList=[]
        self.KPutList=[]
        self.numberAgent = 0

    def numberStep(self):
        if self.buyPriceCall > 0:
            self.buyOffertListCall.append(self.buyPriceCall)
        else:
            self.buyOffertListCall.append(self.priceList[0])
            #self.KCallList.append(self.KCall)
        if self.buyPricePut > 0:
            self.buyOffertListPut.append(self.buyPricePut)
        else:
            self.buyOffertListPut.append(self.priceList[0])
            #self.KPutList.append(self.KPut)
        self.returnStrategy=0
        self.active +=1
        valueStrategy = 0
        self.numberSteps+=1
        self.numberAgent +=1
        if self.numberSteps == self.nSteps:
            self.numberCycles +=1
            self.numberAgent = 0
            if self.numberCycles == self.numCycles:
                if self.numberSteps == self.nSteps:
                    if self.flag == "CALL":
                        if self.priceList[-1] > self.KCall:
                            if self.buyPriceCall > self.KCall:
                                valueStrategy = (self.cash +
self.BsCall - self.buyPriceCall + (self.buyPriceCall - self.KCall))
                                self.returnStrategy = (valueStrategy -
self.cash)/self.cash
                             #   print
"returnStrategy","agent.Covered=",self.number,"EsercitaCall",
self.returnStrategy,"self.KCall",self.KCall
                                self.flag = 0
                            else:
                                valueStrategy = (self.cash +
self.BsCall - self.buyPriceCall - (self.KCall - self.buyPriceCall))
                                self.returnStrategy = (valueStrategy -
self.cash)/self.cash
                              #  print
"returnStrategy","agent.Covered=",self.number,"EsercitaCall",
self.returnStrategy,"self.KCall",self.KCall
                                self.flag = 0
                        if self.priceList[-1] < self.KCall:
                            valueStrategy = (self.cash + self.BsCall -
self.buyPriceCall + self.priceList[-1])
                            self.returnStrategy = (valueStrategy -
self.cash)/self.cash
                         #   print
"returnStrategy","agent.Covered=",self.number,"NonEsercitaCall",
self.returnStrategy,"self.KCall",self.KCall
                            self.flag = 0
                    if self.flag == "PUT":
                        if self.priceList[-1] >  self.KPut:
                            valueStrategy=(self.cash + self.BsPut -
self.buyPricePut + self.priceList[-1])
```

```
                            self.returnStrategy=(valueStrategy -
self.cash)/self.cash
                         #   print
"returnStrategy","agent.Covered=",self.number,"NonEsercitaPut",
self.returnStrategy,"self.KPut",self.KPut
                            self.flag = 0
                        if self.priceList[-1] < self.KPut:
                            if self.buyPricePut > self.KPut:
                                valueStrategy= self.cash + self.BsPut
- self.buyPricePut - (self.buyPricePut - self.KPut)
                                self.returnStrategy=((valueStrategy -
self.cash)/self.cash)
                             #   print
"returnStrategy","agent.Covered=",self.number,"EsercitaPut",
self.returnStrategy,"self.KPut",self.KPut
                                self.flag = 0
                            else:
                                valueStrategy= self.cash + self.BsPut
- self.buyPricePut + (self.KPut - self.buyPricePut)
                                self.returnStrategy=((valueStrategy -
self.cash)/self.cash)
                             #   print
"returnStrategy","agent.Covered=",self.number,"EsercitaPut",
self.returnStrategy,"self.KPut",self.KPut
                                self.flag = 0

    def coveredTransaction(self):
        self.randomValue=random.random()
        self.price=self.priceList[-1]
        #self.sellPrice=0
        self.buyPrice=0
        self.BsCall = 0
        self.BsPut = 0
        self.averagePrice = 0
        self.sumPrice = 0
        entrance = 10
        self.will = random.random()

        if len(self.priceList) > entrance:
            param = random.randint(4,len(self.priceList)-2)
            i=2
            while i!= param + 2:
                self.sumPrice = self.priceList[-i] + self.sumPrice
                i=i+1
            self.averagePrice = self.sumPrice / param

            if self.numberAgent<=20:
               # print "self.numberAgent",self.numberAgent
                if self.flag == 0:
                    if self.will < 0.4:
                        if self.averagePrice < self.price:
                        #    print "EVERAGE-
PRICE",self.averagePrice,"SELF.PRICE",self.price
                            self.sigma = numpy.std([self.priceList])
                            self.S0 = self.priceList[-1]
                            self.K = self.S0 + random.randint(0,+10)
                            self.r = 0.03
                            #self.T = 0.5
                            #self.sigma = 0.1
```

```
                            self.T = self.numCycles / 365
                            self.flag = "CALL"
                            self.chooseCall=1
                            self.BsCall =
BsCalculation.BlackScholesEuropeanCallPrice(self.S0,self.K,self.r,self
.sigma,self.T)
                        #   print
"CALL","self.S0",self.S0,"self.K",self.K,"self.r",self.r,"self.sigma",
self.sigma,"self.T",self.T
                            self.buyPrice= self.price +
random.random()
                            self.buyPriceCall = self.buyPrice
                            self.KCall = self.K
                        #   print "agent.Covered=",self.number, "Buy
price","%.2f" % self.buyPrice
                        #   print "self.BsCall", self.BsCall,
"self.K", self.K

self.aBook.setBuyDecision([self.buyPrice,self,self.type])

                       if self.averagePrice > self.price:
                         #  print "EVERAGE-
PRICE",self.averagePrice,"SELF.PRICE",self.price
                            self.sigma = numpy.std([self.priceList])
                            self.S0 = self.priceList[-1]
                            self.K = self.S0 + random.randint(0,+10)
                            self.r = 0.03
                            #self.T = 0.5
                            #self.sigma = 0.1
                            self.T = self.numCycles / 365
                            self.flag = "PUT"
                            self.BsPut =
BsCalculation.BlackScholesEuropeanPutPrice(self.S0,self.K,self.r,self.
sigma,self.T)
                        #   print
"PUT","self.S0",self.S0,"self.K",self.K,"self.r",self.r,"self.sigma",s
elf.sigma,"self.T",self.T
                            self.buyPrice= self.price +
random.random()
                            self.buyPricePut = self.buyPrice
                            self.KPut = self.K
                        #   print "agent.Covered=",self.number, "Buy
price","%.2f" % self.buyPrice
                        #   print "self.BsPut",self.BsPut,"self.K",
self.K

self.aBook.setBuyDecision([self.buyPrice,self,self.type])

    def getPlotStrategyCall(self):
        return self.buyOffertListCall

    def getPlotStrategyPut(self):
        return self.buyOffertListPut

    def setExecutedBuyOrder(self):

        self.assets+=1
     #   print "Agent ",self.type,self.number,"Assets",self.assets
```

```
    def setExecutedSellOrder(self):
        self.assets-=1
     #   print "Agent ",self.type,self.number,"Assets",self.assets
```

8.6 FILE: Function Dictionaries

```
#FunctionDictionaries.py

import ActionGroup
import rpy2.robjects as robjects
import math

class FunctionDictionary:
    def __init__(self,priceForStep,buyList,sellList,steps,numCycles):
        self.buyList = buyList
        self.sellList = sellList
        self.priceForStep = priceForStep
        self.steps = steps
        self.numCycles = numCycles
        self.AgentList = []
        self.x=[]
        self.numMinuts = 500.00 * self.numCycles

    def grafic(self):
        self.actionGroup11 = ActionGroup.ActionGroup("plotPrice")
        def do11(adress):
            time=(adress.numMinuts / adress.steps)
            time1 = round(time,0)

            for i in range(1,adress.steps+1):
                minutes = time1 * i
                adress.x.append(minutes)
        self.actionGroup11.do = do11

        self.actionGroup12 = ActionGroup.ActionGroup("cleanMinuts")
        def do12(adress):
            del adress.x[:]
        self.actionGroup12.do = do12

        self.actionGroup13 = ActionGroup.ActionGroup("plot1")
        def do13(adress):
            print "X", len(adress.x), "Price",
len(adress.priceForStep)
            r = robjects.r
            r.plot(adress.x,adress.priceForStep,xlab = "time", ylab =
"prices")

            r.grid()
            r.title("Price dynamics")
            r.lines(adress.x,adress.priceForStep,col="blue")
            #r.lines(adress.x,adress.buyOffertPut,col="red")
            #r.lines(adress.x,adress.buyOffertCall,color="yellow")
        self.actionGroup13.do = do13

       # self.actionGroup14 = ActionGroup.ActionGroup("plot2")
```

```
       # def do14(adress):
       #     p = robjects.p
       #     p.plot(adress.x,adress.buyList,xlab= "time", ylab =
"numberoffert")

       #     p.grid()
       #     p.title("number offert")
       #     p.lines(adress.x,adress.buyList,col="red")
       # self.actionGroup14.do = do14


        self.actionGroupList = ["plotPrice","plot1","cleanMinuts"]

    def run(self):
        for n in range(self.numCycles):
            for s in self.actionGroupList:
                if s== "plotPrice":
                    self.actionGroup11.do(self)
                if s== "plot1":
                    self.actionGroup13.do(self)
                if s== "cleanMinuts":
                    self.actionGroup12.do(self)
```

8.7 FILE: Level Price Real Data Agents

```
#LevelPriceRealDataAgents.py

from math import *
import random

class LevelPriceRealDataAgent:
    def __init__(self,number,priceList,aBook,nSteps,numCycles):
        self.numCycles=numCycles
        self.nSteps=nSteps
        self.number=number
        self.assets = 0
        self.type = "LevelPriceRealDataAgent"
        self.aBook=aBook
        self.priceList=priceList
        self.variazPerc = 0
        self.step = 0
        self.passo = 0

        self.values=[]
        with open("FTSE_ITALIA_ALL_SHARE.txt") as f:
            for line in f:
                dato=(line.split(';')[3])
                #print dato
                self.values.append(float(dato))

    def numberStep(self):
       self.step+=1

    def levelRealPriceTransaction(self):
```

```
        nstep = self.nSteps
        self.passo = 500 / nstep
     #   print "self.passo", self.passo
        self.randomValue=random.random()
        self.price=self.priceList[-1]
        self.sellPrice=0
        self.buyPrice=0
       # self.finalPriceForStepList =
self.aBook.getFinalPriceForStep()
        j = 0

        if len(self.priceList)>0:
            b =0
            j = self.step*self.passo
            #simulatorPriceLevel= self.finalPriceForStepList[-1]
            realDataLevel = self.values[j]
      #      print
"realDataLevel",realDataLevel,"J",j,"self.step",self.step

            if realDataLevel > self.price:
                self.buyPrice = realDataLevel
       #         print "realDataLevel",realDataLevel
       #         print "agent.LevelPriceRealData=",self.number, "Buy
price","%.2f" % self.buyPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])

            if realDataLevel < self.price:
                self.sellPrice = realDataLevel
        #        print "realDataLevel",realDataLevel
        #        print
"agent.LevelPriceRealData=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])

    def setExecutedBuyOrder(self):
        self.assets+=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets
```

8.8 FILE: Model

```
#Model.py

import Tools
import RandomAgents
import Book
import random
import ActionGroup
import TrendAgents
```

```
import VolumeAgents
import BestOffertAgents
import LevelPriceRealDataAgents
import VariationPriceRealDataAgents
import CoveredAgents

class ModelSwarm:
    def __init__(self,
nRandomAgents,nTrendAgents,nVolumeAgents,nBestOffertAgents,nLevelPrice
RealDataAgents,nVariationPriceRealDataAgents,nCoveredAgents,nSteps,num
Cycles):
        self.numCycles = numCycles
        self.steps = nSteps
        self.nCoveredAgents = nCoveredAgents
        self.nVariationPriceRealDataAgents =
nVariationPriceRealDataAgents
        self.nLevelPriceRealDataAgents = nLevelPriceRealDataAgents
        self.nBestOffertAgents = nBestOffertAgents
        self.nVolumeAgents = nVolumeAgents
        self.nRandomAgents = nRandomAgents
        self.nTrendAgents = nTrendAgents
        self.agentList = []
        self.price= 21276.78
        self.priceList=[self.price]
        self.agentActionList=[]

    def buildObjects(self):
        self.aBook=Book.Book(self.priceList)

        for i in range(self.nRandomAgents):
            anAgentR =
RandomAgents.RandomAgent(i,self.priceList,self.aBook)
            self.agentList.append(anAgentR)

        for i in range(self.nTrendAgents):
            anAgentT =
TrendAgents.TrendAgent(i,self.priceList,self.aBook)
            self.agentList.append(anAgentT)

        for i in range(self.nVolumeAgents):
            anAgentV =
VolumeAgents.VolumeAgent(i,self.priceList,self.aBook)
            self.agentList.append(anAgentV)

        for i in range(self.nBestOffertAgents):
            anAgentB =
BestOffertAgents.BestOffertAgent(i,self.priceList,self.aBook)
            self.agentList.append(anAgentB)

        for i in range(self.nLevelPriceRealDataAgents):
            anAgentL =
LevelPriceRealDataAgents.LevelPriceRealDataAgent(i,self.priceList,self
.aBook,self.steps,self.numCycles)
            self.agentList.append(anAgentL)

        for i in range(self.nVariationPriceRealDataAgents):
            anAgentVar =
VariationPriceRealDataAgents.VariationPriceRealDataAgent(i,self.priceL
ist,self.aBook,self.steps,self.numCycles)
```

```
            self.agentList.append(anAgentVar)

        for i in range(self.nCoveredAgents):
            self.anAgentCov =
CoveredAgents.CoveredAgent(i,self.priceList,self.aBook,self.steps,self
.numCycles)
            self.agentList.append(self.anAgentCov)

    def buildActions(self):

        self.actionGroup1a = ActionGroup.ActionGroup("numberStep")
        def do1a(adress):
            for i in adress.agentList:
                if i.type == "LevelPriceRealDataAgent":
                    i.numberStep()
                if i.type == "VariationPriceRealDataAgent":
                    i.numberStep()
                if i.type == "CoveredAgent":
                    i.numberStep()
        self.actionGroup1a.do=do1a

        self.actionGroup1 = ActionGroup.ActionGroup
("active_inactive")
        def do1(adress):
            adress.agentListCopy = adress.agentList[:]
            random.shuffle(adress.agentListCopy)

            while len(adress.agentListCopy)!=0:
                for i in adress.agentListCopy:
                    action = random.random()
                    ask_will_single_agent =
adress.agentListCopy.pop(0)
                    if action >0.5:

self.agentActionList.append(ask_will_single_agent)
                        #print
"ask_will_single_agent",ask_will_single_agent
                    else:
                        inactive = ask_will_single_agent
                        #print "inactive",ask_will_single_agent

        self.actionGroup1.do=do1

        self.actionGroup2 = ActionGroup.ActionGroup ("bid_ask")
        def do2(adress):
            adress.agentActionListCopy = self.agentActionList[:]
            for i in adress.agentActionListCopy:
                if i.type == "RandomAgent":
                    i.randomTransaction()
                elif i.type == "VolumeAgent":
                    i.volTransaction()
                elif i.type == "BestOffertAgent":
                    i.bestTransaction()
                elif i.type == "LevelPriceRealDataAgent":
                    i.levelRealPriceTransaction()
                elif i.type == "VariationPriceRealDataAgent":
                    i.variationRealPriceTransaction()
                elif i.type == "CoveredAgent":
                    i.coveredTransaction()
```

```
                else:
                    i.trendTransaction()

#Tools.askEachAgentIn(adress.agentActionListCopy,Agents.Agent.randomTr
ansaction,TrendAgents.TrendAgent.trendTransaction)
        self.actionGroup2.do = do2

        self.actionGroup3 = ActionGroup.ActionGroup
("cleanActionList")
        def do3(adress):
            del self.agentActionListCopy[:]
            del self.agentActionList[:]
        self.actionGroup3.do = do3

        #schedule
        self.actionGroupList =
["numberStep","active_inactive","bid_ask","cleanActionList"]


    def step(self):
        for s in self.actionGroupList:
            if s=="numberStep":
                self.actionGroup1a.do(self)
            if s=="active_inactive":
                self.actionGroup1.do(self)
            if s=="bid_ask":
                self.actionGroup2.do(self)
            if s=="cleanActionList":
                self.actionGroup3.do(self)
        print
```

8.9 FILE: Observer

```
#Observer.py

import Model
import ActionGroup
import Book
import RandomAgents
import TrendAgents
import Tools
import random
import VolumeAgents
import FunctionDictionaries
import LevelPriceRealDataAgents
import VariationPriceRealDataAgents
import CoveredAgents

class ObserverAgent:
    def __init__(self):
        self.AgentList = []
        self.price= 1
        self.priceList=[self.price]
```

```
    #create objects
    def buildObjects(self):

        self.numCycles=input("How many cycles?")
        self.nRandomAgents=input("How many RandomAgents?")
        self.nTrendAgents = input ("How many TrendAgents?")
        self.nVolumeAgents = input ("How many VolumeAgents?")
        self.nBestOffertAgents = input ("How many BestVolAgents?")
        self.nLevelPriceRealDataAgents = input("How many
LevelPriceRealDataAgents?")
        self.nVariationPriceRealDataAgents = input("How many
VariationPriceRealDataAgents?")
        self.nCoveredAgents = input("How many CoveredAgents?")
        self.steps = input("How many step?")

        self.modelSwarm =
Model.ModelSwarm(self.nRandomAgents,self.nTrendAgents,self.nVolumeAgen
ts,self.nBestOffertAgents,self.nLevelPriceRealDataAgents,self.nVariati
onPriceRealDataAgents,self.nCoveredAgents,self.steps,self.numCycles)

        self.modelSwarm.buildObjects()

    #actions
    def buildActions(self):

        self.modelSwarm.buildActions()

        self.actionGroup4=ActionGroup.ActionGroup("step")
        def do4(adress):
            for i in range(self.steps):
                adress.modelSwarm.step()
                adress.modelSwarm.aBook.setFinalPriceForStep()
                adress.modelSwarm.aBook.setLenBuyList()
                adress.modelSwarm.aBook.setLenSellList()
                adress.modelSwarm.aBook.setSpreadBuyOffert()
                adress.modelSwarm.aBook.setSpreadSellOffert()
                adress.modelSwarm.aBook.setTransactionList()
                adress.modelSwarm.aBook.setRandomAgentOffert()
                adress.modelSwarm.aBook.setLevelAgentOffert()
                adress.modelSwarm.aBook.setBestAgentOffert()
                adress.modelSwarm.aBook.setCoveredAgentOffert()
                adress.modelSwarm.aBook.setTrendAgentOffert()
                adress.modelSwarm.aBook.setVariationAgentOffert()
                adress.modelSwarm.aBook.setVolumeAgentOffert()
                adress.modelSwarm.aBook.cleanStep()
        self.actionGroup4.do = do4

        self.actionGroup5 = ActionGroup.ActionGroup ("get_prices")
        def do5(adress):
            adress.modelSwarm.aBook.getStartingPrice()
        self.actionGroup5.do = do5

        self.actionGroup6 = ActionGroup.ActionGroup ("print_txt")
        def do6(adress):
            with open("priceForStep.txt","w") as p:

p.write(str(adress.modelSwarm.aBook.getFinalPriceForStep()))
                p.close()
            with open("lenBuyList.txt","w") as f:
```

```
                f.write(str(adress.modelSwarm.aBook.getLenBuyList()))
                f.close()
            with open ("lenSellList.txt","w") as t:
                t.write(str(adress.modelSwarm.aBook.getLenSellList()))
                t.close()
            with open ("spreadBuyOffert.txt","w") as m:

m.write(str(adress.modelSwarm.aBook.getSpreadBuyOffert()))
                m.close()
            with open ("spreadSellOffert.txt","w") as n:

n.write(str(adress.modelSwarm.aBook.getSpreadSellOffert()))
                n.close()
            with open ("randomAgentOffert.txt","w") as r:

r.write(str(adress.modelSwarm.aBook.getRandomAgentOffert()))
                r.close()
            with open ("levelAgentOffert.txt","w") as q:

q.write(str(adress.modelSwarm.aBook.getLevelAgentOffert()))
                q.close()
            with open ("bestAgentOffert.txt","w") as s:

s.write(str(adress.modelSwarm.aBook.getBestAgentOffert()))
                s.close()
            with open ("coveredAgentOffert.txt","w") as c:

c.write(str(adress.modelSwarm.aBook.getCoveredAgentOffert()))
                c.close()
            with open ("trendAgentOffert.txt","w") as x:

x.write(str(adress.modelSwarm.aBook.getTrendAgentOffert()))
                x.close()
            with open ("variationAgentOffert.txt","w") as v:

v.write(str(adress.modelSwarm.aBook.getVariationAgentOffert()))
                v.close()
            with open ("volumeAgentOffert.txt","w") as z:

z.write(str(adress.modelSwarm.aBook.getVolumeAgentOffert()))
                z.close()
               # self.FunctionDictionary =
FunctionDictionaries.FunctionDictionary(adress.modelSwarm.aBook.getFin
alPriceForStep(),adress.modelSwarm.aBook.getLenBuyList(),adress.modelS
warm.aBook.getLenSellList(),self.steps,self.numCycles)
               # self.FunctionDictionary.grafic()
               # self.FunctionDictionary.run()
                print "print_file_txt"
        self.actionGroup6.do = do6


        self.actionGroup7 = ActionGroup.ActionGroup ("clean")
        def do7(adress):
            adress.modelSwarm.aBook.cleanLists()
        self.actionGroup7.do = do7

    #schedule
        self.actionGroupList =
["step","get_prices","print_txt","clean"]
```

```
    #run
    def run(self):
        for n in range(self.numCycles):
            for s in self.actionGroupList:
                if s== "step":
                    self.actionGroup4.do(self)
                if s== "get_prices":
                    self.actionGroup5.do(self)
                if s== "print_txt":
                    self.actionGroup6.do(self)
                if s== "clean":
                    self.actionGroup7.do(self)
            print
```

8.10 FILE: Random Agent

```
#RandomAgent.py

import random

class RandomAgent:
    def __init__(self,number,priceList,aBook):
        self.number=number
        self.assets = 0
        self.type = "RandomAgent"
        self.aBook=aBook
        self.priceList=priceList

    def randomTransaction(self):
        self.randomValue=random.random()
        self.price=self.priceList[-1]
        self.sellPrice=0
        self.buyPrice=0

        if self.randomValue > 0.5:
            self.sellPrice=self.price - 3*random.random()
         #   print "agent.Random=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])
        if self.randomValue < 0.5:
            self.buyPrice= self.price + 3*random.random()
         #   print "agent.Random=",self.number, "Buy price","%.2f" %
self.buyPrice
            self.aBook.setBuyDecision([self.buyPrice,self,self.type])

    def setExecutedBuyOrder(self):
        self.assets+=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets
```

8.11 FILE: Start

```
#Start.py

import Observer


ObserverA = Observer.ObserverAgent()
ObserverA.buildObjects()
ObserverA.buildActions()
ObserverA.run()
```

8.12 FILE: Trend Agent

```
#TrendAgent.py

import random

class TrendAgent:
    def __init__(self,number,priceList,aBook):
        self.number = number
        self.assets = 0
        self.type = "trendAgent"
        self.aBook = aBook
        self.priceList = priceList

    def trendTransaction(self):
        self.randomProbability = random.random()
        self.price = self.priceList[-1]
        self.coefficient = random.gauss(1,0.05)
        self.sellPrice = 0
        self.buyPrice = 0
        self.averagePrice = 0
        self.sumPrice = 0
        entrance = 10

        if len(self.priceList) > entrance:
            param = random.randint(4,len(self.priceList))
            i=2
            while i!= param:
                self.sumPrice = self.priceList[-i] + self.sumPrice
                i=i+1
            self.averagePrice = self.sumPrice / param

            print "Sum prices",self.sumPrice, "Everage prices",
self.averagePrice

            if self.averagePrice < self.price:
                if self.randomProbability > 0.2:
                    self.buyPrice = self.price * self.coefficient
```

```
                    print "agent.Trend=",self.number,"Buy
price","%.2f" % self.buyPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])
                else:
                    self.sellPrice = self.price / self.coefficient
                    print
"agent.Trend=",self.number,"SellPrice","%.2f" % self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])

            if self.averagePrice > self.price:
                if self.randomProbability > 0.2:
                    self.sellPrice = self.price / self.coefficient
                    print
"agent.Trend=",self.number,"SellPrice","%.2f" % self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])
                else:
                    self.buyPrice = self.price * self.coefficient
                    print "agent.Trend=",self.number,"Buy
price","%.2f" % self.buyPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])


    def setExecutedBuyOrder(self):
        self.assets+=1
        print "Agent ",self.type,self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
        print "Agent ",self.type,self.number,"Assets",self.assets
```

8.13 FILE: Variation Price Real Data Agent

```
#VariationPriceRealDataAgents.py

from math import *
import random

class VariationPriceRealDataAgent:
    def __init__(self,number,priceList,aBook,nSteps,numCycles):
        self.numCycles=numCycles
        self.nSteps=nSteps
        self.number=number
        self.assets = 0
        self.type = "VariationPriceRealDataAgent"
        self.aBook=aBook
        self.priceList=priceList
        self.variazPerc = 0
        self.step = 0
        self.passo = 0
        self.finalPriceForStepList = []
```

```
        self.values=[]
        with open("FTSE_ITALIA_ALL_SHARE.txt") as f:
            for line in f:
                dato=(line.split(';')[3])
                #print dato
                self.values.append(float(dato))

    def numberStep(self):
       self.step+=1

    def variationRealPriceTransaction(self):
        self.nstep = self.nSteps
        self.passo = 500 / self.nstep
        self.sellPrice=0
        self.buyPrice=0
        b=0
        j=0

        self.finalPriceForStepList = self.aBook.getFinalPriceForStep()

        if len(self.finalPriceForStepList) > 1:
            self.price=self.priceList[-1]
            j = self.step*self.passo
            b = (self.step - 1)*self.passo
            realDataVariation = ((self.values[j]-self.values[b])/
self.values[b])
            simulatorPriceVariation= ((self.price -
self.finalPriceForStepList[-1])/self.finalPriceForStepList[-1])
          #  print "REAL DATA VARIATION ",realDataVariation,"SIMULATOR
Price Variation",simulatorPriceVariation

            if realDataVariation < simulatorPriceVariation:
                if realDataVariation < 0:
                    if simulatorPriceVariation > 0:
                        self.sellPrice = self.finalPriceForStepList[-
1]*(1 +(realDataVariation+ simulatorPriceVariation))
              #         print
"agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])

            if realDataVariation < simulatorPriceVariation:
                if realDataVariation < 0:
                    if simulatorPriceVariation < 0:
                        self.sellPrice = self.finalPriceForStepList[-
1]*(1 +(realDataVariation-simulatorPriceVariation))
              #         print
"agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])

            if realDataVariation > simulatorPriceVariation:
                if realDataVariation > 0:
                    if simulatorPriceVariation < 0:
                        self.buyPrice = self.finalPriceForStepList[-
1]*(1 + (realDataVariation+simulatorPriceVariation))
```

```
                #       print
"agent.VariationPriceRealData=",self.number, "Buy price","%.2f" %
self.buyPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])

            if realDataVariation > simulatorPriceVariation:
                if simulatorPriceVariation > 0:
                    if realDataVariation > 0:
                        self.buyPrice = self.finalPriceForStepList[-
1]*(1 +(realDataVariation-simulatorPriceVariation))
              #         print
"agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])

            if realDataVariation > simulatorPriceVariation:
                if simulatorPriceVariation < 0:
                    if realDataVariation < 0:
                        self.buyPrice = self.finalPriceForStepList[-
1]*(1 +(realDataVariation-simulatorPriceVariation))
              #         print
"agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])

            if realDataVariation < simulatorPriceVariation:
                if realDataVariation > 0:
                    if simulatorPriceVariation > 0:
                        self.sellPrice = self.finalPriceForStepList[-
1]*(1 +(realDataVariation-simulatorPriceVariation))
              #         print
"agent.VariationPriceRealData=",self.number,"SellPrice","%.2f" %
self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])

    def setExecutedBuyOrder(self):
        self.assets+=1
     #   print "Agent ",self.type,self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets
```

8.14 FILE: Volume Agent

```
#VolumeAgents.py

import random

class VolumeAgent:
```

```
    def __init__(self,number,priceList,aBook):
        self.number = number
        self.assets = 0
        self.type = "VolumeAgent"
        self.aBook = aBook
        self.priceList = priceList
        self.transactionList=[]


    def volTransaction(self):
        self.randomProbability = random.random()
        self.price = self.priceList[-1]
        self.sellPrice = 0
        self.coefficientBuy = random.uniform(1.0,8.0)
        self.coefficientSell = random.uniform(1.0,8.0)
        self.buyPrice = 0
        self.transactionList = self.aBook.getTransactionList()
        self.averagePrice = 0
        self.sumPrice = 0
        entrance = 10

        if len(self.priceList) > entrance:
            param = random.randint(4,10)
            i=2
            while i!= param+2:
                self.sumPrice = self.priceList[-i] + self.sumPrice
                i=i+1
            self.averagePrice = self.sumPrice / param

        if len(self.transactionList)>2:
            if (self.transactionList[-1]) > (self.transactionList[-
2]):
                if self.priceList[-1] > self.averagePrice:
               # print "self.transactionList[-
1]",self.transactionList[-1],"self.transactionList[-
2]",self.transactionList[-2]
                #if self.randomProbability >
(1/((self.transactionList[-1]-self.transactionList[-2])*0.5)):
                    self.buyPrice = self.price + self.coefficientBuy
                  #  print "agent.Vol=",self.number,"Buy price","%.2f"
% self.buyPrice

self.aBook.setBuyDecision([self.buyPrice,self,self.type])
                else:
                    self.sellPrice = self.price - self.coefficientSell
                  #  print "agent.Vol=",self.number,"SellPrice","%.2f"
% self.sellPrice

self.aBook.setSellDecision([self.sellPrice,self,self.type])

    def setExecutedBuyOrder(self):
        self.assets+=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets

    def setExecutedSellOrder(self):
        self.assets-=1
      #  print "Agent ",self.type,self.number,"Assets",self.assets
```

CONCLUSIONI

La complessità del sistema economico, affrontata nel mio lavoro di ricerca, mi ha permesso di studiare i mercati finanziari da un punto di vista differente, cercando di misurare l'effetto aggregato che si genera dalle interazioni tra agenti e che impedisce, ad oggi, la formulazione di leggi certe, che ne descrivano la prevedibilità.

La flessibilità, concessa dagli ABM, mi ha spesso posto dinanzi a problematiche, risolvibili solamente attraverso scelte arbitrarie, che potrebbero far pensare di aver volutamente generato i risultati, che effettivamente sono stati ottenuti e dimostrati.

Tuttavia, la ragione per cui il lettore non dovrebbe essere scettico nei confronti del mio lavoro riguarda l'ottenimento di alcuni risultati contrari alla logica del programmatore. Inoltre, la ricerca di una giusta calibrazione, che eviti di ottenere risultati in maniera forzata, è la prova del fatto che ciò che si è ottenuto non è stato forzatamente voluto.

Cercando di dare un orientamento cronologico alle conclusioni, a cui sono giunto, trovo necessario citare in prima posizione la capacità di aver rigenerato un mercato virtuale che opera come un mercato reale. L'osservanza delle interazioni tra agenti, senza alcun obiettivo specifico legato allo studio delle serie dei prezzi, rappresentava uno dei principali scopi del mio lavoro. E' stato poi il tentativo di rendere sempre più

complesso ed “intelligente” il mio modello a spingermi ad ottenere e ricercare risultati sempre più sofisticati.

Uno tra questi è certamente rappresentato dalla capacità di avere ricreato serie di prezzi simulati, che seguono andamenti e livelli registrati dal Ftse All Share nella settimana di contrattazione dal 13/12/2010 al 17/12/2010. Questo è stato possibile attraverso l’utilizzo di una classe che, programmata in quattro modalità diverse, cercando di aumentare sempre più la distanza tra offerte proposte e valori reali a cui faceva riferimento, mi ha concesso di accettare il reale funzionamento del sistema programmato.

Critiche rivolte all’arbitrarietà di decisione, a cui ho fatto riferimento all’inizio di questo paragrafo, potrebbero sollevarsi da coloro i quali, studiando più approfonditamente le classi Ftse, siano dubbiosi sulla scelta relativa alle funzioni di offerta programmate. Per chiarire ogni dubbio a questo proposito, affermo che la mia decisione si basa su uno studio ragionato, relativo al fatto che se non avessi impostato meccanismi di offerta, fondati su una serie di prezzi fissi a mia disposizione, avrei ottenuto dei risultati, a cui non sarei riuscito a dare una spiegazione precisa. Con questo intendo dire che le classi, a cui sto facendo riferimento, oltre a garantire serie di prezzi, distribuite secondo valori reali, mi permettono un agevole ragionamento sugli effetti generati dalle interazioni con altre classi di agenti. Le evoluzioni, generate dalle simulazioni in cui agenti Ftse sono compresi tra gli investitori virtuali, hanno la possibilità di essere misurate in maniera computazionale, attraverso lo spread tra prezzi simulati e prezzi reali ad ogni istante.

Questo è un aspetto da considerare quale scoperta di un metodo di ricerca con gli ABM, che mi permette di misurare le distorsioni generate dagli agenti intelligenti e causate dalle loro azioni in simulazione.

Per quanto riguarda le specificità di risultato, ottenute da ogni singolo esperimento, può essere posto in evidenza il fatto di essere riuscito a dimostrare che ad ogni effetto corrisponde una causa ben specifica. A questo riguardo, si ricordano le cause legate alle serie di prezzi dipendenti dalle decisioni delle particolari categorie di agenti, capaci di offrire prezzi superiori rispetto ai *range* medio di offerte proposte sul mercato, o

delle particolari categorie di agenti dipendenti a loro volta dagli agenti che operano con maggiore probabilità in uno medesimo verso, calcolando le condizioni di *trend* di mercato al tempo *t*.

La limitazione temporale, che caratterizza un processo di tesi di laurea, lascia inevitabilmente spazio a problematiche non risolte e non approfondite. A questo riguardo, credo che uno dei possibili sviluppi futuri, che la mia tesi può concedere, riguarda la classe *covered,* la quale ha capacità di impostare una strategia di trading con opzioni sul mercato simulato. Sono certo che se avessi avuto la possibilità di migliorare i processi cognitivi, che guidano le decisioni di questa categoria di agenti, avrei ottenuto risultati ancora più sorprendenti. Inoltre, data l'elevata complessità con cui questa categoria di agenti potrebbe intervenire sul mercato, vi sarebbe spazio per la creazione di una nuova tesi.

Altro sviluppo futuro potrebbe riguardare la modalità di calibrazione degli esperimenti, i quali potrebbero produrre ulteriori studi evidenziando nuove correlazioni e, nel caso, contrastare le conclusioni a cui sono giunto.

BIBLIOGRAFIA

ARTHUR, B.W. [1995]. *Complexity in Economics and Financial Markets,* Complexity, 1995, Vol.1, pp. 20-25.

ARTHUR, W. B. et al. [1997]. *Asset Pricing Under Endogenous Expectations in an Artificial Stock Market,* Lane Reading, 1997, Addison-Wesley, MA.

ARTHUR, W. B., et al. [1997]. *The Economy as an Evolving Complex System II,* Lane Reading, 1997, Addison-Wesley, MA.

BAZERMAN, M.H. [1986]. *Judgment in Managerial Decision Making,* John Wiley & Sons, New York.

BEAVER, W.H [1981]. *Market Efficiency*, The accounting Review, Vol.56, No.1, January, 1981, pp 23-37.

BENNETT, S.G. [1991]. *The quest for value: A guide for senior manager,* Harper- Collins, New York, 1991, pp.40-41.

CAMPBELL, J.Y., LO, A.W., MACKINLAY, A.C. [1997]. *The Econometrics of Financial Markets,* Princeton, Princeton University Press, New York, p. 80.

EPSTEIN, J.M., AXTELL, R.L. [1996]. *Growing Artificial Societies: Social Science from the Bottom Up,* MIT Press, Cambridge, MA.

FAMA, E. [1970]. *Efficient Capital Markets: A Review of Theory and Empirical Work,* The Journal of Finance, Vol.25, No.2, May,1970, pp 383-417.

FAMA, E. [1976]. *Foundations of Finance,* Basic Book, New York, Chapters 1 e 2.

FAMA,E. [1991]. *Efficient Capital Market: a Review of Theory and Empirical Work II,* Journal of Finance, Vol. 46, No.5, December, 1991, pp 1575-1617.

JACKWERTH, J.C, RUBINSTEIN, M. [1996]. *Recovering Probability Distributions from Option Prices,* Vol. 51, No. 5, December, 1996, pp 1611-1631.

JENNINGS, R., WOOLDRIDGE, M. [1995]. *Applying Agent Technology,* Journal of Apllied Artificial Intellicence.

JENSEN, M.C [1978]. *Some Anomalous Evidence Regarding Market Efficiency,* Journal of Financial Economics, Vol.6, No.2/3, June/September,1978, pp 93-330.

JOHNSON, N.L. [1999]. *Diversity in Decentralized Systems: Enabling Self-Organizing Solutions,* Los Alamos National Laboratory, NM, 1999, Paper LA-UR-99-6281.

KAHNEMAN, D., TVERSKY, A. [1979]. *Theory: An Analysis of Decision Under Risk,* Econometrica, Vol. 47, No. 2, March, 1979, pp. 263-291.

LATHAM, M. [1986]. *Information efficiency and Information Subsets,* Journal of Finance, Vol. 41, No.1, March, 1986, pp 39-52.

LEBARON, B. *et al.* [2008]. *The Future of Agent Based Research in Economics: A Panel Discussion*, Economic Association Annual Meeting, Boston, March 7, 2008.

LEROY, S.F. [1976]. *Efficient Capital Market: A Comment,* Journal of Finance, Vol. 3, No.1, March, 1976, pp 139-141.

MARSHALL, A. [1890]. *Principles of Economics*, Macmillan and Co., London.

MERTON, H. M. [1991]. *Financial Innovations and Market Volatility,* Blackwell, Cambridge, MA, 1991, pp. 100- 103. Miller refers to BENOIT, B.M., [1964], *The Variation of Certain Speculative Prices,* MIT Press, Cambridge, MA, 1964, pp. 297- 337

OSTROM, T. [1988]. Computer simulation: the third symbol system, Journal of Experimental Social Psychology, Vol.24, No.5, pp. 381-392.

RAPPAPORT, A., MAUBOUSSIN, M.J.[2001]. *Expectations Investing,* Harvard Business School Press, Boston.

RESNICK, M. [1994]. *Turtles, Termites and Traffic Jams,* MIT Press, Cambridge, MA, p. 3.

RICHIARDI, M., LEOMBRUNI, R. [2006]. *Una risposta alle critiche: le metodologie per la definizione dei modelli di simulazione.* In Terna P., Boero R., Morini M., Sonnessa M. [2006]. *Modelli per la complessità,* il Mulino, Bologna

STEWART, G.B.[1991]. *Quest for Value.* HarperCollins, New York, pp 40-41.

TANOUS, P.J [1997]. *Investment Gurus,* New York Institute of Finance.

TERNA, P., BOERO, R., MORINI, M., SONNESSA, M. [2006]. *Modelli per la complessità*, il Mulino, Bologna.

TERNA, P. [2006]. *Costruzione degli agenti: introduzione.* In Terna P., Boero R., Morini, M., Sonnessa M. [2006]. *Modelli per la complessità,* il Mulino, Bologna.

TERNA, P. [2006]. *Modelli ad agenti: introduzione.* In Terna P., Boero R., Morini M., Sonnessa M. [2006]. *Modelli per la complessità,* il Mulino, Bologna.

THALER, R.H. [1992]. *Winner's Curse: Paradoxes and Anomalies of Economic Life,* Free Press, New York, 1992, pp. 50-62.

VAGA, T. [1994]. *Profiting From Chaos,* McGraw Hill, New York, December, 1994, p.294.

RINGRAZIAMENTI

Ringrazio il Professor Terna, che mi ha dato la possibilità di svolgere questo lavoro di ricerca, affiancandomi con entusiasmo e passione. I suoi consigli e le sue puntuali correzioni sono stati indispensabili durante la stesura del mio lavoro.

Ringrazio il Professor Margarita, in veste di correlatore, per la gentilezza dimostratami, i suggerimenti e le domande critiche che ha voluto porgermi e che mi hanno aiutato concretamente anche nella stesura delle conclusioni.

Un grazie doveroso a Borsa Italiana S.p.A. per avermi permesso l'utilizzo di dati fondamentali per la creazione del mio lavoro.

Un grazie sentito e sincero alla mia famiglia e a tutte le persone che mi sono state vicine in questi anni, supportandomi nel mio percorso accademico.