

UNIVERSITA' DEGLI STUDI DI
TORINO



FACOLTA' DI ECONOMIA



Corso di laurea in Economia e Commercio
Tesi di laurea in Economia Matematica

**MODELLO DI SIMULAZIONE AD AGENTI DEL
MERCATO DI BORSA:
REALISMO NEL BOOK.**

Relatore:
Pietro Terna

Correlatore:
Sergio Margarita

Candidato:
Antonio de Ruvo

Anno Accademico 2003-2004

*A mia mamma, mio papà per il loro aiuto e sostegno e
ai miei fratelli. A mia moglie che con pazienza e tenacia
ha permesso il raggiungimento di questo traguardo.
Ed infine ai miei due piccoli figli Riccardo e Alessio.*

INDICE

INDICE	3
CAPITOLO 1.....	7
IL MERCATO DI BORSA COME SISTEMA COMPLESSO	7
1.1 LA COMPLESSITA'	7
1.2 LA BORSA ITALIANA.	8
1.3 IL MERCATO AZIONARIO.....	10
1.3.1 ORARI E FASI DELLA SEDUTA DI BORSA.....	12
1.4 MOTIVI PER CONSIDERARE IL MERCATO AZIONARIO COME SISTEMA COMPLESSO.	13
CAPITOLO 2.....	15
LA SIMULAZIONI AD AGENTI	15
2.1 LE SIMULAZIONI E LO STUDIO DELL'ECONOMIA.	15
2.2 LA COSTRUZIONE DI MODELLI DI SIMULAZIONE AD AGENTI.....	17
2.3 LE RETI NEURALI ARTIFICIALI.....	19
2.3.1 IL METODO DEI CROSS TARGET (CT).....	21
2.4 CENNI SUGLI ALGORITMI EVOLUTIVI.	23
CAPITOLO 3.....	25
ALCUNI MODELLI SULLA BORSA: ASM E AFM	25
3.1 IL MODELLO ASM.....	25
3.1.1 L'INTERFACCIA.....	25
3.1.2 LA STRUTTURA DEL MODELLO.....	27
3.2 IL MODELLO ARTIFICIAL FINANCIAL MARKET.....	29
3.2.1 NETLOGO.....	29
3.2.2 L'INTERFACCIA DI AFM.	31
3.2.3 LA STRUTTURA DEL MODELLO.....	32
CAPITOLO 4.....	35
SWARM E LA STRUTTURA DEI MODELLI SUM E JAVASUM	35
4.1 SWARM.....	35
4.1.1 STRUTTURA DELLE SIMULAZIONI.....	35
4.2 IL MODELLO SUM.....	36

4.2 MODELLO JAVASUM.....	38
CAPITOLO 5	47
IL BOOK DI JAVASUM	47
5.1 PORTING IN JAVA.....	47
5.2 IL BOOK DI SUM.	49
5.2.1 Book.h.....	49
5.2.2 Book.m.....	51
5.3 LA COSTRUZIONE DEL BOOK DI JAVASUM.....	64
5.3.1 IL FILE BOOK.JAVA.....	66
5.3.2 LA GESTIONE DELLE QUANTITA'.....	78
5.4 INSERIMENTO DI METODI E GRAFICI PER LO STUDIO DELLA DINAMICA DI FORMAZIONE DEI PREZZI.....	81
5.4.1 QUANTITA' DI AZIONI NEL <i>BOOK</i> AL TERMINE DELLA FASE DI APERTURA.	81
5.4.2 DIFFERENZA TRA LE QUANTITA' DI AZIONI IN ACQUISTO E IN VENDITA.....	82
5.4.3 DIFFERENZA TRA IL PREZZO MIGLIORE ED IL PEGGIORE DELLE LISTE DI ORDINI.	83
5.4.4 DIFFERENZA TRA MIGLIOR PREZZO LETTERA E MIGLIOR DENATO.	84
5.5 INSERIMENTO DELLE ASTE	84
5.5.1 L'ASTA DI APERTURA.	85
5.5.2 L'ASTA DI CHIUSURA.	90
5.6 INSERIMENTO DEGLI ORDINI AL MEGLIO.	91
 CAPITOLO 6	 95
ALCUNE SIMULAZIONI E STUDIO DEI RISULTATI OTTENUTI ...	95
6.1 ESPERIMENTI SENZA LE ASTE.....	95
6.2 ESPERIMENTI CON LE ASTE.....	100
6.3 ESPERIMENTI CON ORDINI AL MEGLIO.....	104
6.4 APPROSSIMAZIONE DEL PREZZO PROPOSTO DAGLI AGENTI.	107
 APPENDICI	 109
APPENDICE A.....	109
Codice di Generatore.java.....	109
 APPENDICE B	 111
JavaSum: codice del <i>Book</i> prima dell'inserimento delle aste	111
 APPENDICE C	 119
JavaSum: codice del modello più aggiornato (versione 0.8)	119

StartJavaSum.java	119
ObserverSwarm.java	119
ModelSwarm.java	129
Book.java	138
BasicSumAgent.java	160
BasicSumRuleMaster.java	162
APPENDICE D	165
JavaDoc: documentazione del codice	165
Class StartJavaSum	165
Class ObserverSwarm	165
Class ModelSwarm	173
Class Book	180
Class BasicSumAgent	189
Class BasicSumRuleMaster	193
Class RandomAgent	195
Class RandomRuleMaster	197
Class MarketImitatingAgent	198
Class LocallyImitatingAgent	200
Class StopLossAgent	202
Class StopLossAgent	205
Class CurrentAgent	207
Class CurrentIstant	209
Class SwarmUtils	211
Class Matrix	212
RIFERIMENTI BIBLIOGRAFICI	215

CAPITOLO 1

IL MERCATO DI BORSA COME SISTEMA COMPLESSO

1.1 LA COMPLESSITA'.

La realtà in cui siamo immersi è formata da molti sistemi complessi caratterizzati dalla convivenza di tanti fattori eterogenei che, con la loro interazione, creano dinamiche di vario tipo. Le relazioni presenti in questi sistemi, spesso sono caratterizzate da non linearità, per cui risulta molto difficile, se non impossibile, spiegare l'effetto finale partendo dalla conoscenza delle singole cause.

È fondamentale non confondere il termine “complesso” con “complicato”: come anticipato ciò che è complesso non è scomponibile e deve essere considerato nella sua interezza essendo composto da una fitta rete di relazioni che si condizionano a vicenda; complicato invece è qualcosa che può essere scisso nelle sue parti essenziali e dalla somma di queste è possibile capire e prevedere ciò che si ottiene.

Negli ultimi anni lo studio della complessità si è sviluppato maggiormente anche grazie alla grossa spinta impressa dall'evoluzione tecnologica: le innovazioni hanno dato ai ricercatori strumenti più adeguati alla rappresentazione e alla comprensione di fenomeni complessi che interessano varie discipline tra cui anche l'economia.

Col passare del tempo la complessità è stata definita in diversi modi tra cui quella di Richard H. Day (1994) che definisce complesso un sistema dinamico se da un punto di vista endogeno non tende asintoticamente ad un punto fisso, a un ciclo limitato o a un'esplosione. Nella sua visione un fenomeno di questo tipo può avere comportamenti discontinui e può essere descritto da equazioni differenziali o equazioni alle differenze finite, possibilmente comprendendo anche elementi stocastici.

Barckley e Rosser (1999), argomentando su questa definizione, mettono in luce come sia difficile associare le parole di Day alla complessità riferita all'economia, in quanto in questo ambito l'interazione tra gli individui crea complicazioni non prevedibili e quindi difficilmente rappresentabili attraverso una metodologia puramente matematico-statistica.

Questo porta ad una visione completamente innovativa dello studio dell'economia che passa da quella classica, in cui per giungere a delle conclusioni è necessario fare determinate semplificazioni (sotto forma di ipotesi che allontanano dalla realtà, tra le quali razionalità degli individui e informazione perfetta), ad una più innovativa che tende a modellizzare la realtà individuando regole base degli elementi costitutivi del modello. Questo secondo metodo di studio dell'economia punta, attraverso l'interazione delle parti che compongono il modello, all'osservazione delle dinamiche che si creano e non alla loro comprensione puntuale. Se così fosse cadrebbe la definizione di complessità. Come accade nella realtà, a volte, si ottengono

risultati inspiegabili dovuti all'incorporazione della componente non lineare di cui si è parlato prima.

Ormerod (1998) afferma che l'economia e la società non possono essere considerate al pari di una macchina che, pur essendo complicata, in fondo è prevedibile. Tutti i giorni nel modo economico si assiste a fenomeni che a priori non era possibile prevedere, caratteristica principale dei sistemi complessi. Nel libro presenta una serie di esperimenti in cui alla base vi sono semplici regole ma che generano fenomeni inattesi. Uno di questi è effettuato su una colonia di formiche posta alla stessa distanza da due mucchietti identici di cibo. L'esperimento è creato in modo tale che le formiche non abbiano assolutamente motivo di preferire un mucchietto piuttosto che un altro, per cui, non appena una formica ne porta via una piccola quantità, se ne aggiunge in modo da tenerli costantemente uguali. Ovviamente una formica che ha avuto successo nella ricerca è incentivata a tornare, in quanto troverà sempre cibo. Altra cosa importante nella valutazione dei risultati è il fatto che le formiche tornando al formicaio influenzano i loro simili indicando la strada per mezzo di una secrezione lasciata sul terreno. In termini economici questo significa che il comportamento di ciascun agente è influenzato direttamente da quello degli altri. Durante il suo percorso la formica però è condizionata da tutte quelle che le passano accanto per cui è difficile prevedere a priori dove arriverà. Comunque maggiore è il numero di formiche che va verso lo stesso mucchio, superiore è la probabilità che le altre siano spinte verso quella direzione. Ovviamente sulla base di queste considerazioni, si pensava che le prime formiche che uscivano alla ricerca del cibo, avrebbero determinato un'influenza su tutte le altre e, dopo alcune fluttuazioni casuali, si sarebbe dovuto osservare un numero sempre maggiore verso il mucchietto scelto all'inizio da poche formiche.

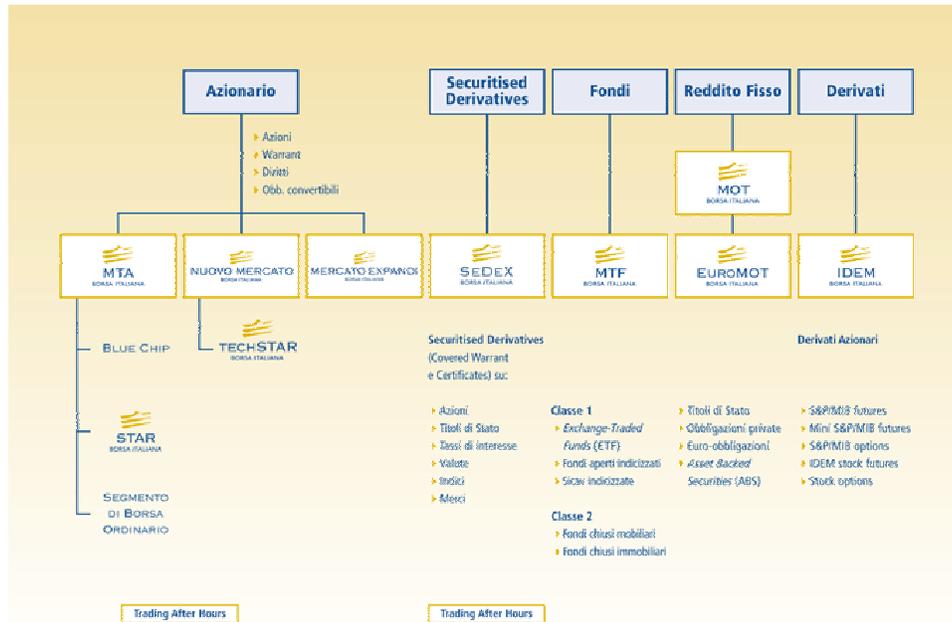
Il risultato che si ottenne fu del tutto inatteso: le formiche pur raggiungendo in massa un mucchio in un certo istante dopo qualche tempo la maggioranza si sgretolava a favore dell'altro mucchio, a volte in modo lento e a volte in modo repentino. Un risultato di questo tipo mette in luce, in modo molto chiaro, come, anche in presenza di regole semplici di base, l'interazione tra agenti e l'influenza reciproca dei comportamenti, possano creare dinamiche complesse che rendono impossibile prevedere il comportamento della collettività sulla base dei comportamenti dei singoli.

Tutto il campo dell'economia è del tutto assimilabile a quanto appena detto ed è quindi facilmente comprensibile perché un mercato come quello borsistico possa essere inteso come sistema complesso.

1.2 LA BORSA ITALIANA.

In generale con il termine borsa si intende il luogo in cui sono scambiati valori mobiliari. Nel nostro paese la società che gestisce i mercati regolamentati è Borsa Italiana s.p.a. che ha ricevuto l'autorizzazione il 2 gennaio 1998 da parte

della CONSOB (COMmissione Nazionale per le SOcietà e la Borsa, l'organo di vigilanza dei mercati e delle società in esse quotate (Damilano e altri, 2002). L'organizzazione del sistema di borsa italiana oggi è così strutturato:



L'immagine è scaricata direttamente dal sito ufficiale della società che gestisce i vari mercati e comparti www.borsaitaliana.it.

Nei prossimi paragrafi concentrerò la trattazione sul mercato azionario, in quanto l'oggetto principale della tesi è il modello di simulazione di borsa in cui è scambiato un titolo azionario

Dalla legge n. 272 del 1913, di costituzione delle borse valori, al T.U.F. (testo unico della finanza) si sono susseguite alcune leggi che hanno segnato l'evoluzione normativa dell'intermediazione mobiliare. Nel 1974 è stata istituita la CONSOB e sono state inserite nuove tipologie di azioni (risparmio e convertibili). Nel 1983 si sancisce la costituzione dei primi fondi comuni di investimenti mobiliari. Nel 1991 sono istituite le SIM (Società di Intermediazione Mobiliare), sono riorganizzati i mercati mobiliari e nasce il mercato dei derivati. Sempre in questo anno è emanata la legge *sull'insider trading*: è colpevole di questo reato chi utilizza informazioni privilegiate per effettuare transazioni di strumenti finanziari, indipendentemente dalla realizzazione del profitto. L'anno successivo nasce la normativa sulle offerte pubbliche di acquisto, vendita e scambio (rispettivamente denominate OPA, OPV e OPS). Nel 1993 sono disciplinati, con due leggi distinte, i fondi pensione e i fondi mobiliari chiusi. L'anno successivo è la volta dei fondi immobiliari. Ultima tappa fondamentale prima del 1998 anno di emanazione del testo unico è il 1996 che con il decreto legislativo n. 415 recepisce la

direttiva comunitaria n. 93/CEE sui servizi di investimento (detto decreto EUROSIM).

Sul mercato mobiliare italiano gli organi di vigilanza sono:

- La CONSOB
- La Banca d'Italia
- Il Ministero dell'Economia e delle Finanze
- ISVAP (Istituto per la Vigilanza sulle Assicurazioni Private)
- COVIP (Commissione di Vigilanza sui fondi Pensione)
- Borsa Italiana

Alla CONSOB sono affidate molteplici funzioni: quella normativa che si rivolge agli emittenti, agli intermediari e ai mercati, quella di vigilanza per fare in modo che gli emittenti operino attenendosi alle regole imposte dal legislatore e da lei stessa, ed infine quella amministrativa, perché concorre all'organizzazione dei mercati regolamentati rilasciando autorizzazioni di vario tipo (iscrizioni agli albi, pubblicazione dei prospetti informativi, all'esercizio dei servizi di investimento).

Banca d'Italia emana norme e vigila sul mercato mobiliare solo in un'ottica di stabilità del sistema finanziario nel suo complesso. Disciplina a riguardo dell'adeguatezza patrimoniale degli intermediari, sulle partecipazioni detenibili in essi, sulla loro organizzazione e il loro controllo interno. Ha anche un potere ispettivo con la facoltà di richiedere informazioni e documenti ai soggetti abilitati per eventuali indagini.

1.3 IL MERCATO AZIONARIO.

Il mercato azionario fa parte del più ampio mercato dei capitali in cui sono scambiati strumenti finanziari con scadenza superiore ai 12 mesi (quelli di durata inferiore sono scambiati sul mercato monetario). Fa parte di questo mercato anche quello obbligazionario su cui sono trattati titoli di stato e obbligazioni emesse sia da Enti che da imprese private.

Sul mercato azionario (comunemente chiamato borsa) sono scambiati titoli rappresentativi di quote di società: chi le acquista può avere due obiettivi, il primo di rendimento per cui rappresenta una forma d'investimento, oppure può essere effettuato al fine di controllo della società (Forestieri e Mottura, 2002).

Il mercato italiano azionario è distinto in tre segmenti:

- 1) MTA sul quale sono quotati i titoli delle società operanti in settori consolidati; a seconda della capitalizzazione, data dal numero delle azioni esistenti per il corso dei titoli quotati, si divide in tre sotto segmenti:
 - a) Blue Chip su cui non trattate le *large cap stocks* che hanno una capitalizzazione superiore agli 800 milioni di euro

- b) Star dove vengono scambiate le *mid cap o medium cap stocks* con capitalizzazione tra gli 800 e i 300 milioni di euro
 - c) Segmento ordinario di borsa sul quale sono trattate le *small e micro stocks* con capitalizzazione inferiore ai 300 milioni di euro.
- 2) Il Nuovo Mercato su cui sono trattate le High growth stocks cioè i titoli appartenenti a società con alto potenziale di crescita (operanti in settori tecnologici, società di nuova costituzione, le start-up di società operanti in mercati nuovi e innovativi.
 - 3) Il Mercato Espandi in cui sono trattati titoli di piccole aziende che non hanno i parametri per essere negoziati sul MTA. I requisiti di ammissione sono meno stringenti.

I mercati gestiti da Borsa Italiana s.p.a., per cui anche il mercato azionario, utilizzano un circuito telematico per le negoziazioni. Questo metodo di contrattazione è stato introdotto a partire dal 1991, precedentemente le negoziazioni avvenivano per lo più al di fuori della borsa non essendoci ancora l'obbligo di concentrazione degli scambi. La parte che passava attraverso la borsa era trattata in dieci borse valori situate nel nostro territorio nazionale e più precisamente a Roma, Milano, Napoli, Palermo, Torino, Genova, Bologna, Firenze, Venezia e Trieste. In queste sedi, smantellate con il decreto di attuazione della direttiva CEE del 1996, le negoziazioni erano fatte con il meccanismo delle grida (asta a chiamata), che consisteva nel ritrovo in una specie di recinto, detto *corbeille*, di tutti coloro che erano interessati ad un certo titolo. Il banditore partendo dal prezzo *fixing* del giorno precedente, procedeva alzando e abbassando il prezzo fino ad arrivare a quello che permetteva di uguagliare domanda ed offerta.

Questa logica di funzionamento è stata ripresa anche in via telematica in alcune fasi della negoziazione del mercato telematico, e più precisamente nelle aste di apertura e di chiusura.

Il sistema di negoziazione utilizzato da Borsa Italiana è del tipo *order-driven* o più semplicemente ad asta¹: questo tipo di negoziazione permette una maggiore probabilità di formazione di prezzi efficienti facendo confluire tutti gli interessati nello stesso luogo (fisico o telematico che sia), creando quindi una certa concorrenza.

Altra caratteristica essenziale del meccanismo di contrattazione dettato da Borsa Italiana s.p.a. è l'assenza di lotti minimi. Dal 2002 è infatti possibile acquistare o vendere anche un solo titolo azionario per volta.

Se non vi sono limiti alla quantità trattata non è così per le variazioni di prezzo: il prezzo prima di tutto può essere espresso in valori multipli di *tick* stabiliti seduta per seduta in base al prezzo di riferimento del giorno precedente², ed oltre tutto non può essere immesso un prezzo che abbia una variazione superiore a $\pm 5\%$ dall'ultimo contratto concluso.

¹ Nel capitolo 5 viene preso in considerazione tutto il meccanismo di conclusione dei contratti.

² Si veda il capitolo 6 in cui è riportata una recente tabella.

Il *book* è il sistema centrale delle contrattazioni: in esso sono inserite tutte le proposte che non hanno la possibilità di essere eseguite immediatamente. La struttura interna di questo “libro” è divisa in due lati: da una parte sono inserite proposte si acquisto in ordine di prezzo decrescente (nel caso di ordini di pari valore si tiene conto della priorità temporale), dall’altra parte le proposte di vendita in ordine crescente di prezzo.

Le informazioni che l’operatore deve indicare in ogni proposta sono:

- Il segno dell’operazione per individuare se in acquisto o in vendita;
- Il valore mobiliare a cui si riferisce (tutti i titoli sono codificati attraverso il codice ISIN);
- Il prezzo (limitato o di mercato) che, a seconda della fase in cui si è, può avere connotazioni differenti;
- La quantità totale oggetto di vendita o acquisto;
- Parametri di negoziazione che indicano la modalità di conclusione;

Di seguito sono indicati tutti i parametri di negoziazione inseribili:

- ERP esposti al raggiungimento del prezzo: la proposta è accettata ma sarà esposta solo al raggiungimento di un determinato prezzo. È utilizzato spesso per impostare i cosiddetti *stop-loss*³
- ECO esegui comunque⁴: la proposta è abbinata a qualsiasi con quelle nel book con qualsiasi prezzo sino alla sua soddisfazione totale
- EOC esegui o cancella: è eseguito anche parzialmente, la parte che resta ineseguita è cancellata automaticamente.
- EQM esegui quantità minima: è eseguito solo se sul mercato è disponibile una quantità minima in linea con il prezzo proposto.
- TON tutto o niente: la proposta è eseguita solo se è raggiunto tutto il quantitativo richiesto, se così non è, automaticamente si elimina la proposta
- VSC valido fino alla cancellazione: rimane valido fino alla fine della contrattazione
- VSD valido fino alla data specificata: rimane valido nei giorni di contrattazione seguenti fino ad un massimo di 90 giorni.

1.3.1 ORARI E FASI DELLA SEDUTA DI BORSA.

Il mercato azionario italiano presenta tre tipi di orari a seconda del segmento o del mercato in cui si opera. In ogni fascia di tempo è svolta una determinata fase: questi possono essere due o tre in base al segmento in cui si è.

I segmenti *Blue-chip* e *Star* ed il Nuovo Mercato hanno tre fasi così articolate:

³ In JavaSum e in Sum vi sono agenti che hanno questo tipo di strategia pur non utilizzando il parametro in fase di inserimento degli ordini.

⁴ È l’unico introdotto nel modello JavaSum

- Asta di apertura dalle ore 8.00 alle 9.30
- Negoziazione continua dalle 9.30 alle 17.25
- Asta di chiusura dalle 17.25 alle 17.40

L'MTA segmento di borsa ordinario ha le stesse identiche fasi ma con orari differenti:

- Asta di apertura dalle ore 8.00 alle 11.00
- Negoziazione continua dalle 11.00 alle 16.25
- Asta di chiusura dalle 16.25 alle 17.40

Il Mercato ristretto ha solo aste di apertura e chiusura dalle 8.00 alle 11.00 e dalle 11.00 alle 16.40.

Durante le fasi di asta di apertura e di chiusura si sviluppano tre distinti momenti. Il primo è la pre-asta in cui sono raccolte le proposte degli operatori e in cui, sulla base di regole precise, è stabilito un prezzo teorico di apertura⁵. Finito il primo, momento il secondo, detto validazione, è utilizzato per stabilire se effettivamente il titolo potrà aprire. Il prezzo teorico è validato se non ha avuto una variazione superiore a $\pm 10\%$ del prezzo di riferimento del giorno precedente. L'ultimo momento è quello in cui i contratti sono conclusi. Nel caso in cui tale limite sia superato il titolo ritenta l'asta, per cui teoricamente e, a volte anche praticamente, vi è la possibilità che il titolo non apra per tutta la giornata. Vi sono anche casi in cui il prezzo teorico non è determinato: il titolo apre lo stesso, inviando gli ordini al *book* di continua secondo determinate regole imposte da Borsa Italiana s.p.a..

L'asta di chiusura è svolta secondo le stesse regole, sopra riportate, per quella di apertura. Il motivo principale di queste due fasi sono per la prima, il voler dare una certa continuità degli scambi tra le varie giornate, mentre per la seconda si punta ad una sorta sintesi della giornata trascorsa.

1.4 MOTIVI PER CONSIDERARE IL MERCATO AZIONARIO COME SISTEMA COMPLESSO.

Principalmente il motivo per cui la borsa dovrebbe essere considerata alla stregua di un fenomeno complesso è la presenza di migliaia di agenti che operano, secondo una propria strategia, in modo indipendente gli uni dagli altri. Pur conoscendo in modo dettagliato quali potrebbero essere le motivazioni o le regole di ogni agente per operare sul mercato, i fenomeni e le dinamiche che si generano non sempre sono prevedibili e nel momento in cui si verifica è pressoché impossibile stabilire per quale motivo si siano verificati.

⁵ Nel capitolo 5 è spiegato puntualmente l'algoritmo di determinazione di tale prezzo.

L'Istituto per gli studi sulla complessità di Santa Fe nel New Mexico ha indicato 6 qualità specifiche della complessità a cui è possibile ricondurre le caratteristiche del mercato di borsa:

- Interazione dispersa e diffusa.
- Nessuna capacità di gestione globale.
- Organizzazioni gerarchiche che si intersecano.
- Adattamento continuo degli agenti in grado di imparare ed evolvere.
- Innovazione continua.
- Dinamica senza equilibrio

Nel prossimo capitolo sarà affrontato il problema di rappresentazione e studio di fenomeni complessi.

CAPITOLO 2 LA SIMULAZIONI AD AGENTI

2.1 LE SIMULAZIONI E LO STUDIO DELL'ECONOMIA.

Nel capitolo 1 ho messo in luce come, l'assenza di indipendenza tra gli elementi dei sistemi complessi, faccia incorporare nei fenomeni che ne risultano componenti che non sono né lineari né additivi. Questa caratteristica non consente di scindere le parti che interagiscono nel sistema al fine di comprendere, valutare e prevedere i fenomeni che si generano.

Le scienze sociali sono definite da Simon come *hard*, nel senso che molti processi che ne fanno parte sono di natura complessa: i sub processi che compongono i fenomeni (appartenenti a vari campi di ricerca tra cui quella economica) non aiutano a comprendere gli effetti che si generano a livello aggregato (Terna, 2000c). La difficoltà di questa scienza ritrova le sue cause nell'impossibilità di effettuare delle sperimentazioni controllate, in quanto è pressoché irrealizzabile la comprensione e la gestione della fitta rete di interazione che si crea tra gli individui.

Molto spesso, nello studio dei fenomeni sociali, si tende ad eliminare l'eterogeneità degli individui che fanno parte della società, creando semplificazioni che allontanano i modelli dalla realtà rendendoli per questo poco attendibili (molte volte si inserisce l'agente razionale molto lontano da quelli che in realtà agiscono nella vita quotidiana).

Terna (2002c) indica tre modi per poter creare modelli:

- Letterario-descrittivi definiti flessibili ma con il limite di non avere la possibilità di essere verificati per mezzo della matematica;
- Statistico-matematici per mezzo dei quali si possono costruire modelli semplificati che si allontanano dalla realtà. Risultano molto difficili da utilizzare in campi come le scienze sociali in cui il grande numero di variabili e la loro interdipendenza potrebbe portare a soluzioni indeterminate.
- La simulazione al computer (Ostrom, 1988).

Con questa ultima alternativa è possibile incorporare le componenti non lineari in quanto sono generate dal modello stesso durante l'interazioni degli agenti che lo popolano.

Negli ultimi anni le simulazioni hanno avuto una maggiore attenzione da parte dei ricercatori operanti nelle scienze sociali. Il motivo principale è la possibilità di creare degli agenti artificiali che popolano l'ambiente virtuale ricreando sistemi complessi.

L'idea alla base è di produrre dei programmi che descrivano gli agenti, e gli ambienti in cui operano, per poi farli interagire ottenendo modelli che emulano la realtà.

Molto apprezzato è anche il fatto che con queste rappresentazioni è possibile osservare ciò che potrebbe accadere nel caso in cui venisse modificato lo stato di partenza del mondo virtuale, il comportamento di alcuni agenti, o singole variabili alla base del modello.

Un grande vantaggio che questi modelli possiedono è la necessità di utilizzare il computer: questo permette di superare alcuni limiti umani (per esempio la memoria e la precisione nei calcoli). La simulazione mette a disposizione un laboratorio virtuale (Parisi, 1999) per il ricercatore per osservare sistemi che altrimenti non potrebbero essere rappresentati.

A differenza dei modelli classici, sono necessarie solo alcune semplificazioni che non pregiudicano e non mettono in dubbio la validità del modello nel ricreare fenomeni complessi.

Con un simile strumento non si ha più la necessità di scindere le varie componenti che creano un sistema complesso, avendo la possibilità di osservare i fenomeni, e la loro dinamica di formazione, nella loro interezza.

Con i modelli tradizionali è possibile definire correttamente i legami che vi sono tra le diverse parti e tra le parti ed il sistema stesso; tuttavia è estremamente difficile, se non impossibile, determinare come le parti siano influenzate dall'ambiente che le circonda e soprattutto come lo stato del sistema sia modificato dai rapporti che si creano tra gli agenti. Con le simulazioni ad agenti, questo problema è facilmente aggirato potendo descrivere attraverso codice informatico tutte le varie parti del sistema in modo dettagliato, incorporando quindi anche l'influenza che le diverse parti hanno reciprocamente.

Con il metodo delle simulazioni è poi possibile inserire le diverse parti in modo graduale in modo da rendere più facile e controllabile la ricerca di errori nel codice e di conseguenza una verifica costante del funzionamento. Questo è molto importante perché, non appena il modello sarà ultimato, i risultati che si otterranno potranno essere "bizzarri" e inattesi per cui sarà difficile stabilire se sono attendibili o frutto di errori di programmazione.

Le principali critiche che sono state avanzate a questa metodologia sono:

- Risultano troppo semplificate rispetto alla realtà. Più volte nella trattazione si è detto come anche le teoriche classiche presentano vari gradi di semplificazione. A differenza di altri metodi qui è possibile contenerle, ma è comunque importante osservare che queste siano quelle poche necessarie ad una più facile comprensione del fenomeno, e che allo stesso tempo, non compromettano la validità dei risultati che si otterranno.
- Sembrano non dire nulla di nuovo sul fenomeno che si simula. Questa obiezione è facilmente contrastabile vista la flessibilità che ha una simulazione. Infatti, grazie ai modelli è possibile osservare come i risultati si modificano apportando variazioni alle variabili iniziali. Oltretutto attraverso l'osservazione dei risultati è possibile testare la validità della teoria sottostante al modello.

- Non siamo in grado di simulare un qualcosa di cui non siamo profondi conoscitori. Anche questa critica è facilmente contrastabile. Lo scopo delle simulazioni è arrivare ad una migliore conoscenza, altrimenti non ci sarebbe la necessità di doverla simulare.
- La riproduzione con successo di un fenomeno non sempre consente di facilitarne la comprensione. Questa affermazione è vera nel caso in cui la simulazione è osservata solo dall'esterno. Non è più così se la si considera, come detto prima, un laboratorio sperimentale che permette al ricercatore di verificare cosa comportano determinate scelte di modifiche da apportare al modello.

Oltre a queste critiche di non grosso spessore vi sono alcuni problemi che effettivamente delineano alcuni limiti di questa metodologia. Alcuni di questi problemi sono molto probabilmente dovuti alla scarsa conoscenza del metodo, per cui ad esempio si rischia, come accennato prima, di cadere in semplificazioni che escludono aspetti rilevanti al fine della simulazione.

Un altro problema è che si tende a non verificare adeguatamente e in modo sistematico i risultati empiricamente. Spesso ci si basa sulle intuizioni pur essendo comunque fondamentale un confronto dettagliato e ampio tra i risultati delle simulazioni e le evidenze empiriche conosciute.

2.2 LA COSTRUZIONE DI MODELLI DI SIMULAZIONE AD AGENTI.

Parisi (2001) fornisce una definizione di simulazione ad agenti: è un particolare tipo di simulazione che cerca di riprodurre fenomeni collettivi facendo interagire tra di loro un determinato numero di agenti, i quali hanno determinate regole, ossia reagiscono agli stimoli che ricevono in modo determinato. Ovviamente una metodologia di questo tipo risulta molto utile nello studio dei fenomeni sociali. In economia questo tipo di simulazioni offrono uno strumento flessibile per lo studio dei comportamenti all'interno di un sistema.

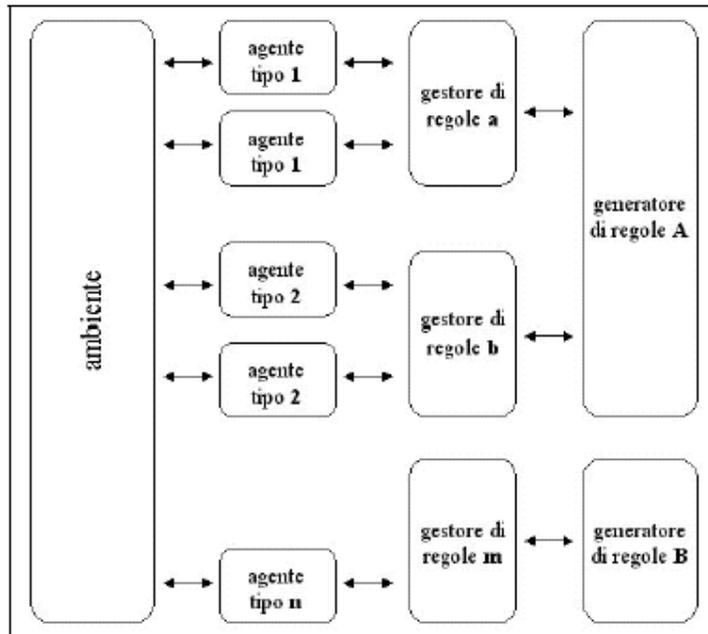
Negli ultimi anni, al fine di rendere più agevole e standard la costruzione dei modelli di simulazione, si sono sviluppate alcune metodologie e alcuni strumenti in continua evoluzione. Tra questi Swarm, che consiste in un insieme di librerie di funzioni, e in protocolli di costruzione dei modelli in modo standardizzato⁶.

Il dettare regole comuni di costruzioni aiuta a creare modelli che possono essere modificati anche da chi non è l'inventore, ma da chi ha intenzione di adattare il modello alle proprie esigenze per lo studio di altri fenomeni o per integrare quello preesistente.

Swarm come già accennato non è l'unico strumento esistente ma è sicuramente tra i più diffusi.

⁶ Si veda il capitolo 4 .

Terna (2002c) propone di utilizzare uno schema di costruzione dei modelli che permette alcune semplificazioni della programmazione. Tale schema è detto ERA (*Environment-Rules-Agents*⁷) che riporto nella figura sottostante:



L'ambiente all'interno del protocollo di Swarm è il *Model*, ovvero la parte di codice che si occupa della creazione degli agenti, e di tutto ciò che è necessario per poter gestire in modo adeguato la simulazione (gestione anche del tempo). La seconda colonna individua le varie tipologie degli agenti: questi possono essere di vario tipo (agenti con mente, che hanno metodi di apprendimento, o agenti senza mente che operano in modo casuale). La terza colonna rappresenta la parte di programmazione che gestisce le regole. Prima di operare ogni agente interroga il proprio gestore (detto *RuleMaster*) il quale sulla base dei dati che riceve indica all'agente come comportarsi. L'ultima parte della figura rappresenta il *RuleMaker*, ovvero il generatore di regole. Questo è utilizzato per poter rendere gli agenti più evoluti e capaci di modificare la loro strategia di azione durante la simulazione.

Gli scenari che si possono sviluppare attraverso modelli simulativi sono (Terna, 2001):

- Agenti "senza mente" operanti in ambiente non strutturato;
- Agenti "senza mente" operanti in ambiente strutturato;
- Agenti "con mente" operanti in ambiente non strutturato;
- Agenti "con mente" operanti in ambiente strutturato;

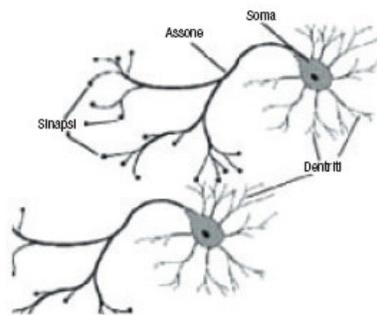
⁷ Nel capitolo di programmazione del *Book* questo schema è adattato al modello di borsa.

Da quanto appena elencato si può dedurre che, con i modelli ad agenti, è possibile riprodurre anche simulazioni che possono essere definite “s sofisticate”, in cui vi è la presenza di un ambiente strutturato (ad esempio il meccanismo di contrattazione della borsa), ed allo stesso tempo agenti con capacità di scelta, capacità di adattamento ed anche di apprendimento dagli eventi che sorgono nella simulazione.

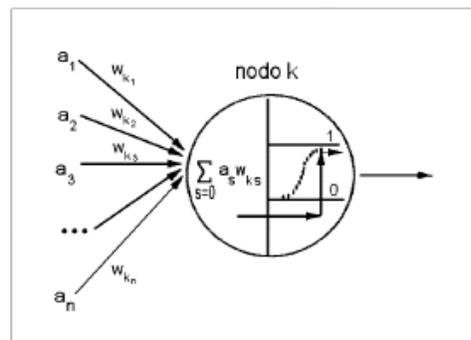
Nei prossimi paragrafi presento alcune strutture informatiche che permettono di fornire una “mente” agli agenti artificiali dandogli un comportamento adattivo all’interno della simulazione.

2.3 LE RETI NEURALI ARTIFICIALI.

Le reti neurali artificiali nascono dall’osservazione di processi naturali: in questo caso, quello esaminato, fu il funzionamento del cervello umano attraverso i neuroni e le sinapsi.



Terna (1995) definisce le reti neurali artificiali come una funzione che produce un vettore di output da un vettore di input sulla base di un insieme di parametri, detti pesi, la cui determinazione non si discosta dalle consuete stime statistiche in ambito multivariato. Di seguito riporto lo schema di un neurone artificiale che compone la rete neurale:



Le reti neurali artificiali maggiormente utilizzate sono quelle con il meccanismo *feedforward* in cui l'informazione non ha modo di tornare indietro. Questo tipo di reti è costituito, in genere, da tre strati:

- Nodi di *input*
- Nodi *hidden*
- Nodi di *output*

Sostanzialmente ogni neurone riceve varie attivazioni che possono essere derivanti dai nodi precedenti o direttamente dagli *input* forniti alla rete, nel caso del primo strato.

Come si può vedere dalla figura precedente, le attivazioni arrivano al neurone artificiale moltiplicate per dei pesi che inizialmente sono generati casualmente. All'interno del neurone è calcolata la sommatoria dei prodotti precedentemente descritti e il risultato è trasformato per mezzo di una funzione sigmoide.

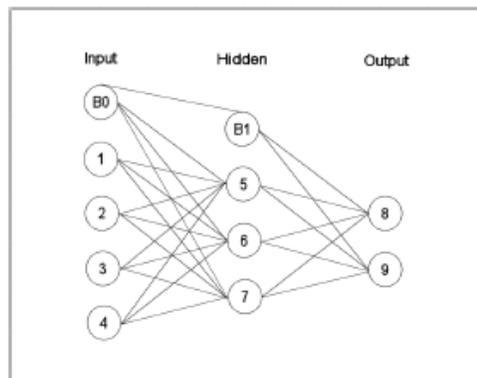
La funzione più comunemente utilizzata è la logistica:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Gli *input*, generalmente, come anche le attivazioni dei neuroni degli altri strati sono generalmente compresi nell'intervallo 0-1.

Avendo pesi iniziali casuali, la rete deve effettuare in primo luogo un apprendimento: attraverso un *training set*, ovvero un insieme di dati determinati, la rete deve produrre in uscita valori prestabiliti in modo tale che sia possibile, attraverso un adeguato metodo, modificare i pesi per ridurre al minimo l'errore che inizialmente è prodotto. Successivamente si passa dalla fase di verifica attraverso un insieme di dati di controllo detti *validation set*, e solo dopo ha senso utilizzarla su dati nuovi.

Lo schema di una rete neurale può essere così rappresentato:



La determinazione dei pesi appartenenti ad una rete neurale artificiale si basa sulla minimizzazione dell'errore misurato come differenza tra i valori y di *output* ed i valori utilizzati come *target* dai *pattern* del *training set*. Per ogni caso (*pattern*), si disporrà di un vettore x di valori in *input* ed un vettore t di risultati attesi. L'errore da minimizzare è calcolato mediante una doppia sommatoria:

$$\sum_n \sum_k (t_k - y_k)^2$$

dove n va da 1 al numero totale dei *pattern* e mentre k individua i singoli nodi di *output*.

Con questo metodo, quello che si ottiene è uno speciale tipo di elaborazione delle informazioni che, se associato ad agenti artificiali, gli permettono di comportarsi in base a ciò che accade nell'ambiente (*input*). Gli *output* che ottengono dalla rete, sono le azioni che l'agente decide di effettuare.

Terna (1995) individua almeno cinque ragioni perché il ricercatore è spinto ad utilizzare le reti neurali artificiali:

1. consentono di avere una riproduzione semplificata del cervello umano;
2. perché si allontanano dalla precedente visione dell'intelligenza artificiale basata sulla rappresentazione simbolica e subsimbolica: si tenga presente che l'intelligenza è per lo più basata su scelte piuttosto che su simboli;
3. dimostrano una particolare rilevanza matematico-statistica, avendo come principale caratteristica la capacità di approssimare qualsiasi tipo di funzione;
4. sono una struttura adatta a lavorare in parallelo, cioè su più macchine;
5. specificità nella costruzione connessionistica, intesa come paradigma di rappresentazione della coscienza e di determinazione del comportamento dei singoli agenti o organizzazioni.

Attraverso le reti neurali artificiali è possibile costruire tre distinti modelli: il primo è fondato su agenti che utilizzano la rete per fare previsioni; un secondo in cui gli agenti basano le proprie regole comportamentali; l'ultimo tipo strettamente econometrico e quello di utilizzare la rete come una funzione che produce un output da un input interpolando i dati per generalizzarli e produrre previsioni.

2.3.1 IL METODO DEI CROSS TARGET (CT)

Il nome *cross target* deriva dal metodo con cui vengono costruiti i target alla base dell'apprendimento della rete neurale sottostante. Questo metodo consente

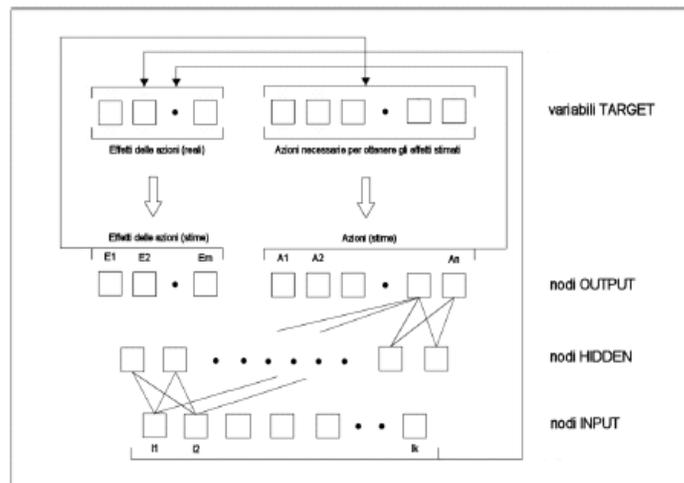
di creare comportamenti coerenti con le congetture effettuate dall'agente, sia sulle azioni da intraprendere, sia sugli effetti che queste produrranno.

Gli *output* della rete si suddividono in due parti: quelli che si riferiscono alle azioni e quelli che invece indicano gli effetti. Con questo metodo i *target* sono creati in modo incrociato: quelli delle azioni sono calcolati in coerenza con gli *output* relativi agli effetti (in modo da creare una capacità di decidere quali azioni fare per raggiungere determinati effetti attesi); i *target* degli effetti, viceversa sono costruiti in coerenza con gli *output* relativi alle congetture sulle azioni (sviluppando la capacità dell'agente di stimare gli effetti relativi alle azioni che sta intraprendendo). In generale gli input sono provenienti dall'ambiente e dalle azioni svolte dal soggetto. Il metodo dei *cross target* è in grado di produrre simultaneamente azioni e apprendimento: il motivo per cui questo capita, è da ricercarsi nel fatto che le azioni sono necessarie per costruire le informazioni su cui basare l'apprendimento.

Terna (1995) individua 4 fasi per l'apprendimento della rete neurale:

1. Produzione degli output della rete: rispettivamente congetture sulle azioni da compiere e congetture sugli effetti di tali azioni;
2. Formazione dei *target* relativi alle congetture sugli effetti sulla base degli output della rete riguardante le congetture di azione;
3. Formazione dei *target* relativi alle congetture sugli azioni da compiere che si basano sulle stime degli effetti delle azioni prodotte dalla rete;
4. *Backpropagation*: si effettua l'apprendimento, correggendo i pesi della rete (anche in questo caso inizialmente scelti casualmente in un *range* stabilito dall'utilizzatore) per ottenere stime degli effetti in accordo con le conseguenze delle azioni congetture, e congetture di azioni più coerenti con le stime degli effetti.

La figura sottostante rappresenta lo schema generale di una rete basata sul metodo dei *cross target*:



Si possono effettuare due tipi di apprendimento: uno di breve periodo in cui non si ha la modifica significativa di tutti i pesi (soprattutto quelli tra *input* e *hidden* si modificano poco); ed uno di lungo periodo effettuato ex post e non in modo concomitante con le azioni, in cui realmente i pesi sono modificati e formati.

2.4 CENNI SUGLI ALGORITMI EVOLUTIVI.

Questo tipo di algoritmo è nato dall'osservazione del processo di evoluzione della natura. Ferraris (2000) afferma che in queste tecniche le regole sono assoggettate a procedimenti, di riproduzione ed estinzione, al fine di garantire che le buone norme rimangano in vita, al posto di quelle che non generano i risultati migliori.

Questa tecnica prevede un'attenta analisi delle regole che ci sono in un dato momento, in modo da decidere quali siano buone e possano essere utilizzate per generarne di nuove, e quali invece debbano essere eliminate.

Ovviamente con una metodologia di questo genere, le evoluzioni del sistema sono subito catturate e incorporate nella creazione delle nuove regole. Come per le reti neurali inizialmente i pesi erano casuali, anche negli algoritmi evolutivi si parte da una popolazione di regole prodotte casualmente dal programma.

Per stabilire quali elementi della popolazione debbano avere più probabilità di evolvere (individui migliori), e quali invece siano destinati all'estinzione (individui peggiori), si utilizza una misura di adattabilità detta *fitness*.

Un tipo di algoritmo evolutivo sono gli algoritmi genetici. Con questo metodo tutti gli individui della popolazione sono costituiti da una stringa di caratteri uguali a 0 o a 1. La lunghezza della stringa deve poter contenere un numero sufficiente a incorporare tutta la conoscenza necessaria per descrivere una qualche strategia.

Si possono distinguere 5 momenti fondamentali:

1. creazione della popolazione;
2. valutazione degli individui;
3. la riproduzione;
4. applicazione degli operatori genetici;
5. ripetizione del ciclo n numero di generazioni;

Ad ogni iterazione sulla base della *fitness* sono scelti gli individui idonei alla riproduzione e quelli destinati ad estinguersi. La riproduzione avviene mediante copia ed incrocio in modo da avere una compartecipazione delle due regole alla nascita della nuova. L'incrocio, detto anche *crossover*, è effettuato per mezzo dell'estrazione di un numero di posizione casuale della stringa, dopodiché è scambiata, tra i due individui coinvolti, la parte a sinistra di tale posizione. Gli algoritmi genetici si prestano molto bene per rappresentare un agente che si evolve durante la simulazione.

CAPITOLO 3

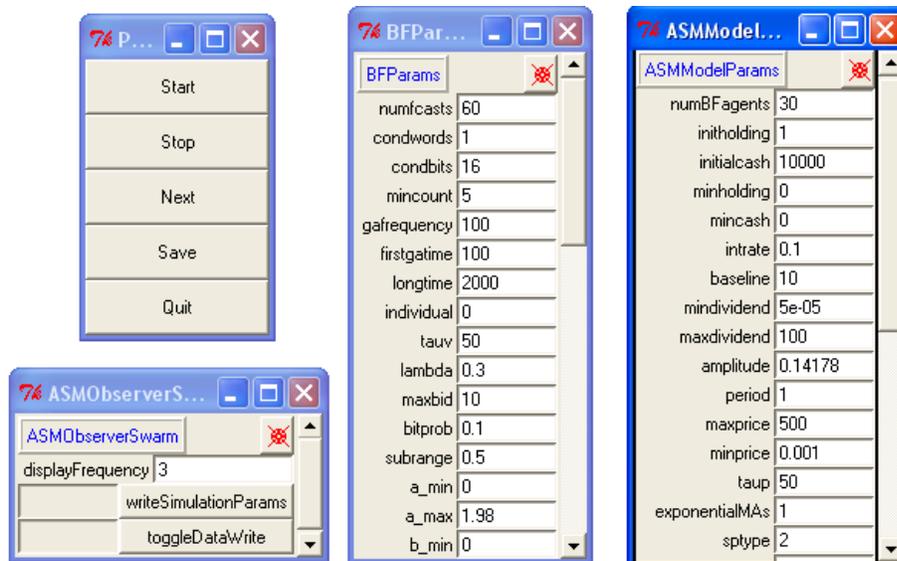
ALCUNI MODELLI SULLA BORSA: ASM E ARTIFICIAL FINANCIAL MARKET

3.1 IL MODELLO ASM.

È uno tra i primi modelli ad agenti artificiali applicato ai mercati finanziari. Nasce, indicativamente tra il 1980 e il 1990 da un'idea di Brian Arthur e John Holland a cui si aggiungeranno in seguito Paul Tayler e Richard Palmer. Il progetto è sviluppato al Santa Fe Institute del New Messico ed è costruito utilizzando i protocolli e le librerie di Swarm⁸. Durante gli anni trascorsi si sono susseguite alcune versioni che possono essere scaricabili gratuitamente da internet (<http://artstkmkt.sourceforge.net/>). Il linguaggio utilizzato è l'Objective C anche se oggi è possibile trovare in linea anche una versione scritta in Java.

3.1.1 L'INTERFACCIA.

Qui di seguito riporto le finestre che sono a disposizione dell'utilizzatore con la versione 2.2 di ASM, seguite poi da una breve descrizione:



Utilizzando Swarm, l'interfaccia del modello ha esattamente la stessa struttura presentata nei prossimi capitoli per i modelli Sum e JavaSum.

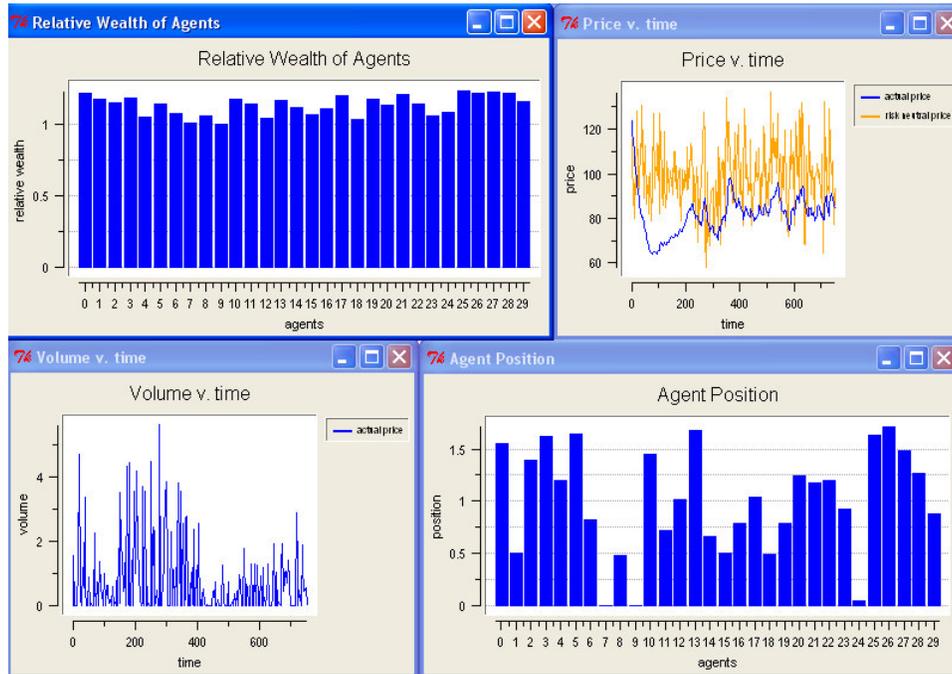
⁸ Si veda il capitolo 4.

Sono presenti: un pannello di controllo che permette di avviare, arrestare o di uscire dal programma; il *Model* attraverso il quale si stabiliscono i parametri della simulazione prima di avviarla; un *Observer* con cui si decide con che frequenza osservare i dati generati dalla simulazione; l'ultima finestra presente, chiamata *BFParms*, imposta alcune variabili utilizzate durante la simulazione internamente dagli agenti, dai *classifier system* e dagli algoritmi genetici⁹. Scaricando il programma si ottiene una ricca documentazione in formato html, che consente di capire il significato di ogni parametro modificabile dall'utente. Di seguito ne riporto alcuni facenti parte del *Model* (l'utilità di queste variabili sarà spiegata nel paragrafo successivo in cui si vedrà il funzionamento interno del modello):

- *numBFagents*: il numero degli agenti dell'esperimento;
- *initholding*: la dotazione iniziale di attività rischiose;
- *initcash*: la dotazione iniziale di titoli *risk free*;
- *minholding*: il limite di vendite allo scoperto;
- *mincash*: il limite di indebitamento;
- *intrate*: il tasso di interesse sull'impiego esente da rischio r_f ;
- *baseline*: il valore centrale attorno al quale viene calcolato il dividendo;
- *mindividend*, *maxdividend*: i minimi e massimi per dividendi;
- *amplitudine*: l'ampiezza delle deviazioni da *baseline*;
- *period*: il periodo di autocorrelazione del processo;
- *minprice*, *maxprice*: i minimi e massimi per i prezzi;
- *taup*, *eta*, *etamin*, *etamax*, *maxiterations*, *minexcess*, *rea*, *reb*: i parametri da utilizzare nel calcolo del prezzo di equilibrio;
- *exponentialMAs=1*, *exponentialMAs=0*: il tipo di media utilizzare nel calcolo dell'accuratezza delle regole; una media mobile pesata esponenzialmente (1) o una semplice media mobile di n periodi (0);
- *randomseed*: l'innesco dei numeri casuali (randomseed = 0 indica l'innesco casuale);
- *tauv*: il parametro τ che definisce l'orizzonte temporale preso in considerazione dall'agente;
- *maxbid*: l'offerta massima di attività rischiose;
- *initvar*: la varianza iniziale;
- *maxdev*: la massima deviazione di una previsione nella stima della varianza;
- *lambda*: il coefficiente di avversione al rischio.

Avviando la simulazione il ricercatore può visionare alcuni grafici che permettono di avere da subito un'idea dei risultati che la simulazione sta generando:

⁹ Nel prossimo paragrafo è spiegato il funzionamento interno per cui si comprenderà la loro utilità.



3.1.2 LA STRUTTURA DEL MODELLO.

La struttura ricostruita è un mercato in cui vi sono solo due titoli negoziabili (LeBaron, 2002):

- *risk free bond* che può essere inteso come un titolo a reddito fisso, o può essere considerato un deposito bancario (la scelta effettuata dall'agente è solo sull'altro titolo, ciò che rimane inallocazione della sua disponibilità è interamente investito in questa attività finanziaria);
- *risky stock* ovvero un'azione che paga un dividendo stocastico.

Il processo autoregressivo seguito dal dividendo è di questo tipo:

$$d_t = \bar{d} + \rho(d_{t-1} - \bar{d}) + \mu_t$$

In una delle prime versioni questi parametri erano fissi, mentre nelle ultime versioni, come ad esempio quella riportata nel paragrafo precedente, si ha la possibilità modificare i valori all'inizio della simulazione.

La ricchezza di ogni agente è data dalla somma investita nel titolo risk free e dal numero delle azioni possedute per il prezzo di mercato. Una delle principali critiche del modello è stata, come vedremo in seguito, fondata sul fatto che gli

agenti non sono influenzati, nel loro operato, dalla ricchezza posseduta. Essi hanno solo il limite di un massimo di negoziazione per *tick* di 10 titoli che si abbassa però a 5 nel caso in cui vendano allo scoperto.

Altro limite che ha generato critiche è la possibilità di indebitamento illimitato e la presenza di un solo titolo rischioso.

Gli agenti non dialogano tra di loro per acquistare o vendere, questi sono messi in relazione attraverso il mercato stesso: un valore di prezzo è annunciato dal banditore a tutti gli agenti del modello; dopo aver effettuato le proprie previsioni, gli operatori inseriscono le loro proposte per quantità unitarie di azioni e, sulla base dell'eccesso di domanda e di offerta, si procede al calcolo del nuovo prezzo facendo in modo di mantenere in equilibrio il mercato (eccesso di domanda il banditore alza il prezzo, viceversa abbassa).

Questo meccanismo è utilizzato in quanto, secondo LeBaron (2002), per comprendere meglio i mercati finanziari è necessario ideare modelli attorno ad una determinata condizione di equilibrio, pur non essendo una condizione necessaria nei modelli di simulazione ad agenti. Questo tipo di rappresentazione però allontana il modello dalla realtà.

Gli agenti effettuano le loro proposte sulla base delle aspettative razionali elaborate per mezzo di una selezione di informazioni che ottengono dal mercato. La caratteristica principale degli agenti che operano è di essere avversi al rischio in modo costante: la loro funzione di utilità da massimizzare è del tipo CARA (avversione assoluta al rischio costante).

La domanda, ipotizzando la normalità di distribuzione dei rendimenti (con media il valore atteso della somma tra il prezzo e il dividendo, per il periodo successivo, e varianza pari a $\sigma_{i,p+d}^2$), è definita come segue:

$$x_t^i = \frac{\widehat{E}(p_{t+1} + d_{t+1}) - (1 + r_f)p_t}{\gamma \sigma_{i,p+d}^2}$$

dove r_f è il tasso fisso, γ è l'indice di avversione al rischio costante.

Dalla sommatoria delle varie quantità richieste il banditore stabilirà il prezzo, come già detto, al fine di riequilibrare domanda ed offerta.

Considerando la formalizzazione data fino ad ora non vi è nulla che cambia rispetto ai modelli economici tradizionali. La vera caratteristica che lo differenzia, è il metodo adottato per rendere capaci di formulare previsioni sul futuro da cui far dipendere le decisioni. I prezzi e i dividendi attesi sono calcolati per mezzo di un *classifier system*, ovvero un algoritmo evolutivo capace di creare regole eterogenee per ogni agente della simulazione.

Questo metodo, sulla base di un sistema binario, mette a disposizione dell'agente le varie condizioni di mercato. Le regole di classificazione sono costituite da due parti, la prima è una stringa di bit, caratterizzata dai simboli 1,

0, #, in cui lo zero e l'uno descrivono la presenza o meno di uno stato, mentre il carattere # indica uno stato di indifferenza, la seconda parte della regola, invece, ha il compito di convertire il set di bit in una previsione del prezzo e del dividendo.

Le attese sono stimate utilizzando la regola che meglio si adatta allo stato in cui si trova il sistema nel momento in cui si effettua la decisione. Le varie regole (*classifier rules*) sono costantemente monitorate in modo da stabilire per mezzo di un algoritmo genetico quali sono adatte alla riproduzione e quali invece devono essere scartate¹⁰.

Il set di regole posseduto da ogni agente è costituito da regole basate sull'analisi tecnica (confrontando il prezzo con la media mobile di varia lunghezza) e regole di analisi fondamentale (dove vengono prese in considerazione le opportunità che il dividendo o l'interesse fisso possono dare). Le regole che gli agenti hanno a loro disposizione, oltre a consentire la previsione permettono, di stimare la varianza che sarà utilizzata poi per calcolare il quantitativo da trattare (si veda la formula precedente).

Per concludere, metto in evidenza come in questo modello sia totalmente mancante la microstruttura del mercato di borsa e come, pur essendo un modello di simulazione ad agenti, sia comunque presente un solo tipo di agente razionale con comportamento ottimizzante. Questo rende il modello molto lontano dal mondo in cui viviamo e non sfrutta al meglio le possibilità date dalle simulazioni (ad esempio la gestione di più famiglie di agenti diverse fra loro e con proprie regole di comportamento) (Terna, 2003).

3.2 IL MODELLO ARTIFICIAL FINANCIAL MARKET.

Sempre nel campo delle simulazione dei mercati finanziari, un secondo modello di simulazione è *Artificial Financial Market*.

Pur non avendo anch'esso la microstruttura sottostante del mercato di borsa per il calcolo del prezzo, rispetto ad ASM possiede alcune differenze sostanziali che lo rende molto interessante.

Prima di descrivere come è strutturato il modello, sia a livello di interfaccia che di struttura interna è necessario parlare dell'ambiente informatico utilizzato per la costruzione.

3.2.1 NETLOGO.

Questo ambiente nasce negli Stati Uniti con lo scopo della rappresentazione dei fenomeni complessi creati dall'interazione tra agenti eterogenei.

È un prodotto derivante dalla fusione di StarLisp (da cui acquisisce la gestione di più agenti) e dal Logo (da cui prende in eredità la grafica).

Il risultato ottenuto è un ambiente in cui si ha a disposizione un insieme di strumenti adatti alla gestione di simulazioni in cui prendono parte centinaia di

¹⁰ Un cenno sugli algoritmi genetici è riportato alla fine del capitolo precedente.

agenti. Data l'eterogeneità che si può creare nei comportamenti degli attori è possibile, attraverso l'interazione di essi, riprodurre sistemi complessi al fine di poterli comprendere in un modo più adeguato.

Il bello di questo ambiente è che pur essendo scritto in Java, il ricercatore, dopo aver capito la metodologia e le istruzioni sottostanti all'ambiente, ha la possibilità di creare in modo più rapido i propri modelli di simulazione rispetto a quelli creati con altri strumenti (ad esempio Swarm). Con questo non intendo affermare che sia un prodotto migliore, ma può essere vantaggioso per creare semplici simulazioni ad agenti che se fatte in altro modo potrebbero risultare più difficili.

L'implementazione di un nuovo modello passa dalla creazione degli agenti, alla definizione delle azioni che ognuno deve compiere, per poi arrivare alla definizione delle istruzioni che impongano agli agenti di effettuare le azioni prestabilite.

NetLogo permette di effettuare una moltitudine di simulazioni che possono appartenere a varie discipline (economia, biologia, matematica e tante altre).

Anch'esso, come Swarm, è un prodotto *open source* direttamente scaricabile dal sito <http://education.mit.edu/starlogo> (in esso è anche possibile trovare esempi di modelli di simulazione tra cui Artificial Financial Market).

La struttura dei modelli costruiti attraverso questi ambienti conta tre tipi di strumenti:

- i *turtles* ovvero le tartarughe che si muovono e interagiscono (gli agenti). Sono così definiti per la discendenza dal Logo in cui si rappresentava un tartaruga che muoveva secondo regole all'interno del mondo;
- i *patch* che sono i punti che compongono lo spazio in cui si muovono le tartarughe. Il mondo è bidimensionale, per cui è possibile individuare ogni *patch* attraverso una coppia di coordinate (il centro è individuato da [0,0]). Il mondo, che a prima vista sembra una superficie limitata, è invece un toroide, per cui se l'agente sconfinava da una parte rientra dal lato opposto. Anche le *patch* hanno la possibilità di eseguire azioni e quindi si può fare in modo che influenzino gli agenti e, di conseguenza, i risultati che si ottengono dalla simulazione;
- l'Observer che, al pari di Swarm, permette di osservare ciò che avviene durante la simulazione. Questa operazione avviene per mezzo di un'interfaccia grafica riportata nel prossimo paragrafo.

Un'altra analogia, che l'ambiente possiede con Swarm, è la possibilità di creare famiglie omogenee di agenti, inseriti in liste, con la limitazione però di non poter inviare messaggi alla lista in modo che essa informi coloro che ne fanno parte. Il modo con cui lo si può fare è l'utilizzo di cicli e procedure ricorsive. Ultima caratteristica che merita di essere menzionata è la possibilità di far interagire gli umani con l'ambiente simulato. Questo è consentito dagli Hubnet

che sono costruiti per essere usati su più postazioni in modo da influenzare il comportamento contemporaneo di più agenti.

3.2.2 L'INTERFACCIA DI AFM.

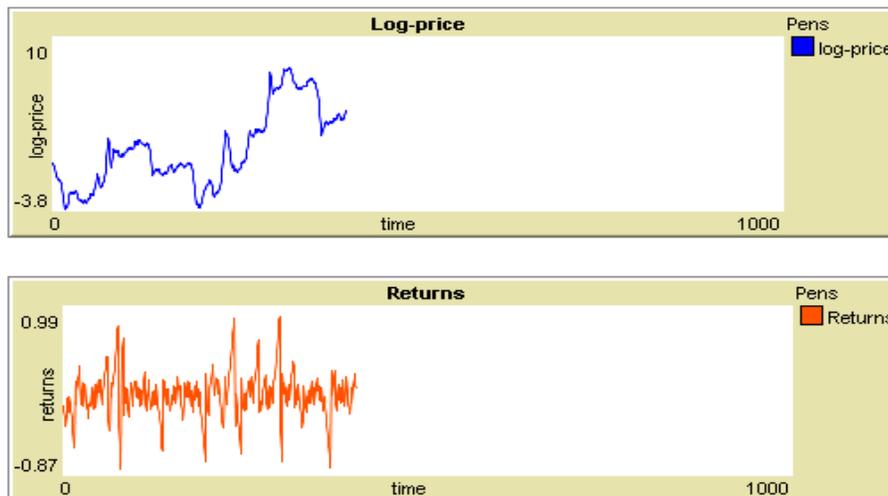
Per accedere al programma di simulazione è sufficiente selezionare il *file* di *Artificial Financial Market*, ovviamente dopo aver installato il programma NetLogo.

L'utente ha a disposizione la possibilità di modificare solo 4 parametri della simulazione:

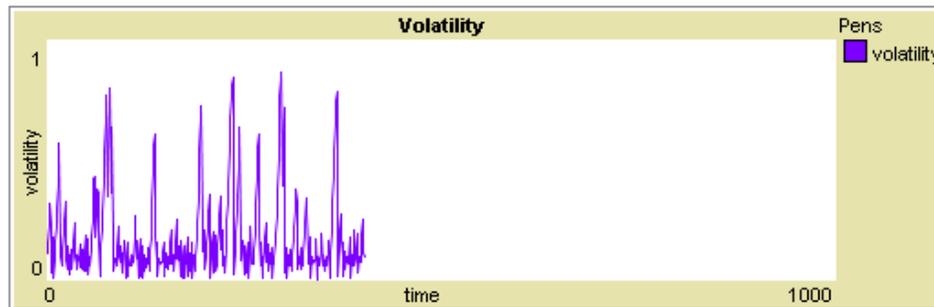
- *miu* utilizzata per stabilire le regole del comportamento degli agenti. Con essa viene modificata la volatilità con la quale l'agente interpreta le informazioni (agisce sulla distribuzione casuale modificandone la deviazione standard¹¹);
- *sigma* serve per calcolare la volatilità con la quale l'agente interpreta l'informazione;
- *max-news-sensitivity* è il massimo valore che può raggiungere la sensibilità dell'agente all'informazione;
- *max-base-propensity-to-sentiment-contagion* che rappresenta il valore massimo di propensione ad essere influenzati dagli altri agenti.

Vi sono poi tre grafici che permettono all'utilizzatore di avere sotto controllo ciò che avviene durante la simulazione.

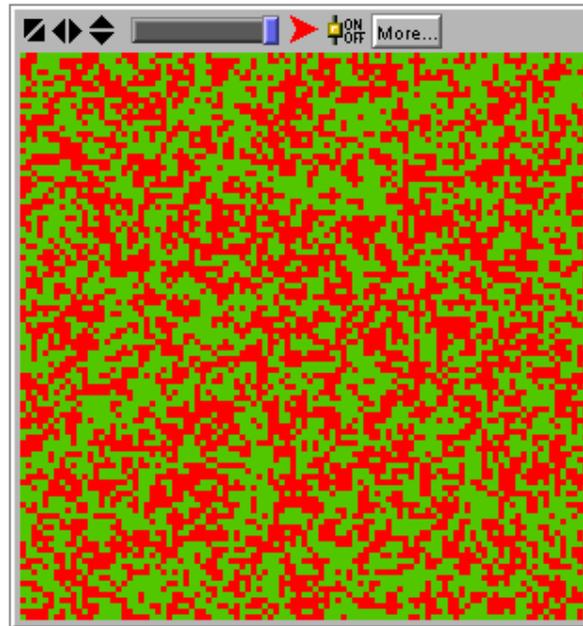
Il primo riporta il logaritmo del prezzo (più avanti ne riporto il metodo di calcolo), il secondo permette la visualizzazione dei rendimenti ed il terzo misura la volatilità dei rendimenti. I grafici sono così strutturati:



¹¹ Nel prossimo paragrafo verrà spiegato in modo più rigoroso.



L'ultima figura dell'interfaccia è il mondo su cui ci sono disegnati gli agenti. Questi assumono diverso colore a seconda del tipo di decisione che hanno preso: verde se acquistano, rosso se vendono.



3.2.3 LA STRUTTURA DEL MODELLO.

Il modello è creato in modo da mettere in luce l'importanza dell'iterazione che vi è tra gli agenti, mettendo in questo caso l'accento sull'influenza che può avere una notizia che arriva al mercato. L'informazione è una variabile importantissima nel mercato reale: tutti i giorni i corsi dei titoli risentono delle notizie che sono rese pubbliche sia in senso positivo, generando *trend* rialzisti, che negativo, facendo stornare il corso del titolo oggetto di notizia.

Il modello è costruito con l'utilizzo di agenti eterogenei, limitatamente razionali, che non massimizzano la loro funzione di utilità ma basano i loro comportamenti in base all'interpretazione data alla notizia che ogni giorno

arriva nel mondo simulato dall'esterno. Oltre alla propria interpretazione l'agente può essere influenzato dagli agenti che gli stanno vicino, creando così un'interdipendenza tipica della realtà e dei sistemi complessi.

A mio parere il carattere vincente di questo modello risiede proprio nell'incorporazione di una variabile fondamentale nelle dinamiche dei prezzi, qual 'è l'informazione.

Questa è rappresentata da una variabile casuale distribuita normalmente con media 0 e varianza pari ad 1 (normale standard). Ad essa è assegnato un valore: se è una notizia positiva (variabile aleatoria maggiore di 0) il valore attribuito è uguale a 1 mentre se negativa (negli altri casi) assumerà valore -1. Prendendo spunto dall'*information* presente nel *file* di *Artificial Financial Market*, il tutto può essere così formalizzato:

$$I(t) \sim N(0,1);$$

$$Q(t) = 1 \text{ se } I(t) > 0, Q(t) = -1 \text{ altrimenti.}$$

Dove $I(t)$ è l'informazione e $Q(t)$ è il valore ad esso assegnato ed è l'unico comune a tutti gli agenti.

La relazione seguente invece definisce le regole di comportamento in cui, ad eccezione di $Q(t)$, tutti i parametri differiscono per ogni agente:

$$S_i(t) = \text{sign}(K_i * N S_i(t) + n s_i * Q(t) + e_i(t)),$$

dove:

- K_i è la propensione dell'individuo ad essere influenzato dalle opinioni altrui;
- $N S_i(t)$ è la somma delle sensazioni degli agenti limitrofi;
- $n s_i$ è la sensibilità dell'agente al significato qualitativo delle informazioni;
- $e_i(t)$ è il termine di errore casuale che tiene conto dell'interpretazione delle informazioni da parte del singolo.

Se $S_i(t) > 0$ l'agente acquista; in caso contrario vende.

Tutti gli agenti hanno quindi una propria rappresentazione dell'ambiente in cui operano e conoscono solo le opinioni di coloro che gli stanno vicino, eliminando l'ipotesi di informazione perfetta utilizzata nella teoria classica.

Il prezzo è calcolato partendo da quello del giorno precedente; ad esso si aggiunge l'indice dell'opinione media (rapporto tra la sommatoria delle opinioni e il numero degli agenti). I rendimenti invece sono calcolati come posizione netta dell'agente rapportata al numero totale di agenti operanti sul mercato, e sono utilizzati internamente per aumentare o diminuire i coefficienti che indicano la propensione all'influenza: se una notizia è confermata dal mercato il coefficiente sarà aumentato di un importo pari al rendimento, nel caso contrario sarà diminuito.

CAPITOLO 4

SWARM E LA STRUTTURA DEI MODELLI SUM E JAVASUM

4.1 SWARM.

L'interazione tra numerosi agenti semplici, e quindi l'influenza che il comportamento degli uni potrebbe avere nei confronti degli altri, fanno in modo che siano generate dinamiche non lineari capaci di creare fenomeni complessi che non possono essere studiati matematicamente. Sulla base di questa affermazione, negli anni si è sviluppata la necessità di studiare tali sistemi per mezzo delle simulazioni ad agenti.

Nel 1994 da un'idea di Chris Langton, presso il Santa Fe Institute, nacque *Swarm*, un ambiente studiato per fornire strumenti standard in continua evoluzione, che permettessero la riproduzione di fenomeni complessi al computer.

Secondo obiettivo, ma non per questo meno importante, era l'individuazione di un protocollo da seguire, nella costruzione dei modelli che potesse rendere più agevole da parte dei ricercatori sia la comprensione del codice informatico di programmi preesistenti, sia l'adattabilità a diverse discipline.

Il software fino ad oggi creato comprende una serie di librerie che permettono la realizzazione di simulazioni multi-agenti, con la possibilità di essere utilizzate con due tipi di linguaggi, l'Objective C e, a partire dal 1999, il Java.

Swarm è un software *open source*, scaricabile dal sito www.swarm.org, nato per essere utilizzato con sistemi operativi *Unix*. Da qualche anno è comunque possibile utilizzarlo anche con sistemi operativi Microsoft Windows attraverso un emulatore di ambiente *Linux* chiamato Cygwin, anch'esso scaricabile gratuitamente dal sito www.cygwin.com.

4.1.1 STRUTTURA DELLE SIMULAZIONI.

Una delle principali caratteristiche dei modelli creati attraverso l'ausilio di *Swarm*, e quindi con linguaggi di programmazione ad oggetti, è quella di avere un programma composto da più *file* che durante la simulazione interagiscono tra di loro attraverso una fitta rete di attivazione di metodi in essi contenuti.

La struttura della simulazione può essere divisa in due livelli distinti:

- Il *Model* che descrive le regole della simulazione. Con esso sono creati gli oggetti necessari per effettuare gli esperimenti (agenti e ambiente in cui operano), ed anche la sequenza di azioni da effettuare per poter avere un esatto susseguirsi degli eventi (gestite dallo *schedule*).
- L'*Observer* è il livello più elevato e consente l'osservazione di ciò che avviene all'interno del mondo virtuale. Risulta essere il primo oggetto creato dal metodo *main*, ed è quello che permette di creare il livello

inferiore rappresentato dal modello stesso (*Model*). Anch'esso possiede un proprio metodo di scansione degli eventi, che può differire da quello del *Model*, e consente di immagazzinare i dati prodotti dalla simulazione per un eventuale studio *ex-post* dei risultati che si ottengono.

Il modello JavaSum, oggetto di questa tesi, riprende in modo fedele i concetti appena presentati. Nei paragrafi successivi sarà illustrato per primo il modello "genitore" Sum-0.66, scritto in Objective C, nel suo insieme per poi passare ad una descrizione dettagliata del nuovo scritto in Java.

4.2 IL MODELLO SUM.

Sum (*Surprising(Un)realistic Market model*) è inizialmente ideato e sviluppato da Terna. Successivamente alcune tesi di ricerca hanno continuato il progetto, raggiungendo un grado di approssimazione della realtà molto elevato.

L'obbiettivo è stato quello di ricreare, con il minor numero di semplificazioni, un ambiente virtuale per la simulazione delle sedute di borsa, in cui agenti artificiali propongono di acquistare e vendere un ipotetico titolo azionario.

La versione 0.66, da cui si parte per effettuare il *porting* in Java, prevede la possibilità, da parte di ogni agente, di inserire nel *book* proposte per il singolo titolo trattato sul mercato virtuale. L'utilizzatore del modello attraverso la finestra del *Model* ha la possibilità di stabilire molti parametri della simulazione tra cui il numero e il tipo di agenti che operano sul mercato:

1. agenti *random*: è il primo tipo di agenti introdotto nel modello. Sono i meno strutturati a livello di codice, dovendo semplicemente proporre ordini in modo casuale sia intermini di quantità, sia di prezzo che di genere (acquisto e vendita). Rappresentano la categoria di *trader* che non seguono un metodo di scelta razionale o basato su una qualche regola di decisione, e che acquistano e vendono in base al loro istinto.
2. agenti imitativi: si dividono in due sottotipi, quelli che imitano il mercato e che quindi basano le loro decisioni sull'andamento dei prezzi medi dei giorni precedenti, e quelli che imitano localmente il comportamento della maggior parte degli agenti. Nel mercato reale questo tipo di operatori coincide con coloro che si fanno influenzare da eventi esterni o da decisioni prese da un certo numero di operatori. Un esempio eclatante è la bolla speculativa scoppiata nel 2000-2001: negli anni in cui si sviluppò la bolla, molte persone si affacciarono al mercato di borsa attratti dai forti guadagni che amici e conoscenti o anche il negoziante di fiducia avevano fatto in breve tempo sui titoli azionari. La corsa all'*equity* si diffuse come una vera epidemia che continuò per qualche tempo ad alimentare la tendenza al rialzo dei corsi azionari. Stessa cosa accadde nel momento in cui il mercato si accorse che forse la valutazione dei titoli non era equa.

3. agenti *stop-loss* sono coloro che hanno come strategia quella di limitare le perdite ad un certo livello. Il modello prevede due tipi, quelli che non osservano se hanno posizioni lunghe o corte sull'azione e quelli che invece osservano la loro posizione speculativa.
4. agenti *ANNForecastApp* sono agenti che operano sulla base delle previsioni di prezzo calcolate dall'agente *Forecasting*, il quale fa solo previsioni senza operare sul mercato. Quest'ultimo, sulla base degli indici di variazione dei giorni precedenti, attraverso una rete neurale artificiale (RNA), formula previsioni sul prezzo del giorno successivo. In base all'indice di variazione del prezzo ottenuto come output della rete neurale, gli agenti *ANNForecastApp* propongono ordini sul mercato.
5. agenti BP-CT operano sul mercato attraverso una strategia che nasce da una rete neurale artificiale che ha come metodo di apprendimento quello dei *Cross Target*.

Peculiarità del modello Sum sono la possibilità di far operare un solo agente per tick (unità di tempo di simulazione) e l'inserimento di soli ordini unitari (se l'agente ad esempio volesse acquistare 3 azioni, inserisce tre ordini da 1). La versione 0.66 non prevede aste, ma solo una fase di pre-apertura in cui vengono accumulati gli ordini per non iniziare le giornate con il *book* vuoto¹². L'utilizzatore del programma, attraverso una *probe*, può stabilire la percentuale di agenti che operano in questa prima fase: tutta la lista degli agenti viene interpellata e a seconda dei parametri impostati, solo alcuni o tutti propongono ordini (è previsto anche il caso in cui nessuno opera, iniziando ogni giorno con *book* vuoto). Le *probe* sono un tipico strumento di *Swarm* che consente, a chi utilizza, il programma di modificare i dati iniziali della simulazione dalle finestre dell'*Observer* o de *Model* (tra questi ad esempio la già citata possibilità di stabilire quanti e quali agenti far operare).

In versioni più evolute del modello Sum sono state inserite contrattazioni su più titoli, calcolo di un indice sul quale costruire un derivato, aste e da ultimo la possibilità di far interagire col modello agenti umani con esperimenti in aula e in rete (Mezzera 2003 e Cappellini 2003).

Il risultato a cui si è arrivati è notevole potendo avere a disposizione un simulatore molto strutturato e flessibile, capace di produrre dinamiche di formazione dei prezzi simili a quelli reali (bolle e crash), ma soprattutto che ha eliminato una delle maggiori semplificazioni dei modelli di simulazione di borsa, consistente nel banditore che stabilisce i prezzi (eliminato grazie all'introduzione di alcune regole di formazione dei prezzi dettate da Borsa Italiana s.p.a.). Rispetto ad altri modelli precedentemente creati¹³, seppur con qualche semplificazione, i meccanismi del modello di simulazione Sum si

¹² Si veda il capitolo 5 per una spiegazione dettagliata del funzionamento del *book*.

¹³ Nel capitolo 3 vengono presentati due di questi modelli.

avvicinano in maniera più adeguata a quelli reali del mercato di borsa gestito da Borsa Italiana s.p.a.

Il motivo principale per cui si è pensato di effettuare un *porting* in Java è sicuramente la maggiore diffusione di questo linguaggio di programmazione e di conseguenza la maggiore probabilità di un continuo e costante aggiornamento delle librerie di funzione messe da esso a disposizione. Tutto ciò potrebbe risultare utile per un ulteriore sviluppo del progetto di simulazione.

4.2 MODELLO JAVASUM.

In questo paragrafo non si ha l'obiettivo di descrivere il nuovo modello JavaSum sotto il profilo della programmazione, ma si vuole presentare in modo dettagliato l'insieme dei *file* che compongono il programma e soprattutto come si presenta all'utilizzatore.

Come anticipato all'inizio del capitolo, il programma è utilizzabile con computer su cui vi è installato *Swarm*, e per chi possiede il sistema operativo Windows della Microsoft, è necessario avviare il programma di simulazione da una *shell* di Cygwin (emulatore di ambiente Linux).

Il codice sorgente per poter essere utilizzato deve essere compilato cioè tradotto in un linguaggio che possa essere eseguito dall'interprete di Java. In altre parole i *file* “.java” vengono trasformati in *bytecode* e salvati in altrettanti *file* omonimi ma con estensione “.class”. Questa operazione è effettuata utilizzando il comando “*make*”, il quale permette di attivare le operazioni di compilazione contenute all'interno del *file Makefile*. In esso sono presenti altri due tipi di comando: “*make clean*” che cancella i *file bytecode* e “*make cleanall*” che oltre ad essi cancella eventuali *file* con estensione “.java~” e “.stackdump”. Questi nascono in due differenti circostanze: i primi, nel caso in cui si utilizzi come programma per generare codice sorgente Xemacs scaricabile gratuitamente dal sito www.xemacs.org, sono generati nel momento in cui sono apportate modifiche ai *file* “.java”, in modo tale da avere sempre una versione precedente; i secondi invece sono generati da Cygwin nel momento in cui il programma è interrotto da qualche errore in fase di esecuzione.

Un altro *file* di servizio del programma è *cleanDos*, il quale consente di cancellare i file di testo creati dall'*Observer* durante la simulazione per salvare i dati su cui effettuare eventuali studi e considerazioni ex-post.

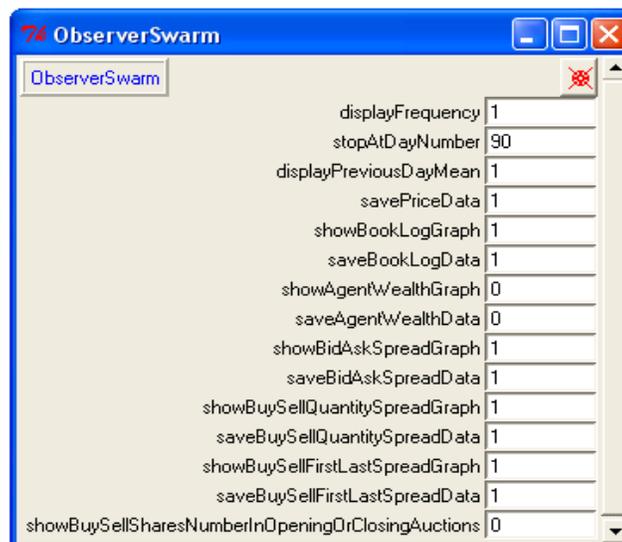
Sempre di servizio risulta essere *JavaSum.scm* che consente di impostare valori differenti ad alcune variabili del modello senza dover ogni volta modificare il codice sorgente e ricompilare il tutto.

Dando il comando “*javaswarm StartJavaSum*” il programma inizia a produrre le prime tre finestre che permettono al ricercatore di impostare i dati iniziali della simulazione, e gli consentono di stabilire quali grafici disegnare e quali dati salvare.

La prima di queste finestre è un pannello di controllo della simulazione contenente cinque tasti con funzioni distinte:



1. il primo avvia la simulazione in modo continuo;
2. il secondo la ferma dando la possibilità di riavviarla;
3. il terzo permette di avanzare un *tick* alla volta;
4. il quarto permette di salvare il layout delle finestre che vengono create. In genere al primo avvio del programma è opportuno schiacciare *next* per poter aprire tutte le finestre del programma e fermare subito la simulazione. Questa azione permette di spostare le finestre in modo che siano tutte visibili. Successivamente il tasto *save* consente di non effettuare, nei prossimi avvii, i passaggi appena illustrati.
5. il quinto arresta e chiude tutte le finestre.



La prima *probe* dell'*Observer* permette di stabilire con che frequenza si osservano i dati prodotti dalla simulazione: se il valore immesso è 1 ad ogni *tick* di simulazione sono aggiornati sia i grafici sia i dati salvati su *file*.

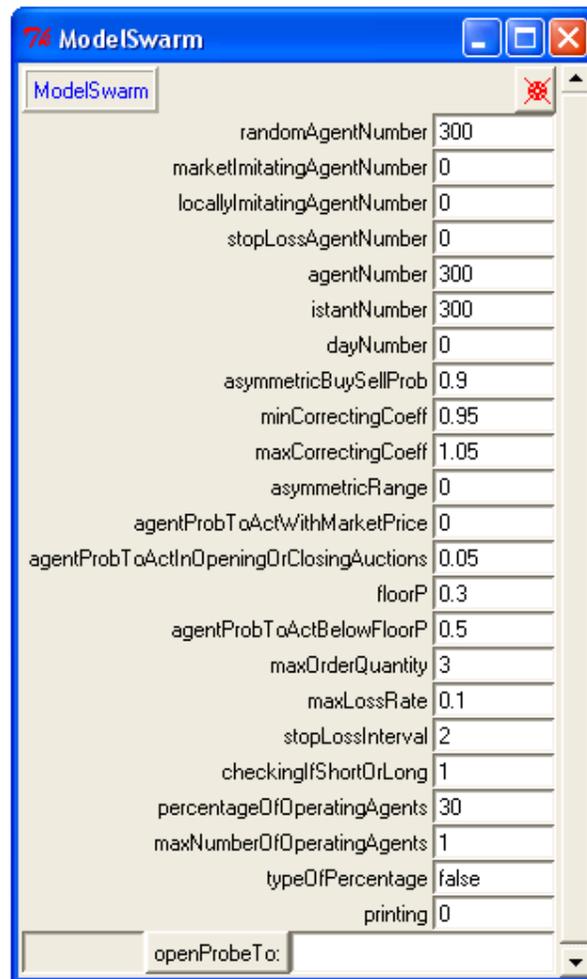
I valori che possono essere inseriti devono essere multipli della variabile *istantNumber* modificabile dalla finestra del *Model*. Questo consente di aggiornare i dati alla fine della giornata ad intervalli di giorni prestabiliti. Se per esempio *istantNumber* è uguale a 10 e *displayFrequency* è pari a 20, il programma aggiornerà i dati ogni due giorni di contrattazione prendendo i dati generati alla fine dell'ultimo *tick* delle giornate.

La seconda sonda permette di stabilire la durata della simulazione indicando il giorno in cui arresterà.

Tutte le sonde successive a seconda che siano impostate uguali a 1 o 0, permettono o meno di effettuare determinate operazioni:

- 1) *displayPreviousDayNumber* fa in modo che sia indicato o no, nella finestra del grafico del prezzo corrente, anche il valore del prezzo medio del giorno precedente. Quest'ultimo non è un prezzo ufficiale in quanto il programma effettua una media aritmetica dei prezzi e non una media ponderata con le quantità a cui si riferiscono.
- 2) *savePriceData*, se attivato, crea due *file* di testo in cui vengono riportati i prezzi di ogni singolo contratto concluso e la media dei prezzi del giorno precedente. Il nuovo modello, a differenza di Sum, permette di far operare più di un agente per ogni istante di simulazione il che comporta che i dati salvati e evidenziati, nel caso in cui venga attivata questa possibilità, siano quelli dell'ultimo contratto concluso perdendo l'informazione di ciò che è avvenuto durante lo stesso *tick*. Un ulteriore sviluppo del modello potrebbe prendere in considerazione il recupero anche di questi dati intermedi.
- 3) *showBookLogGraph* fa sì che sia creato un grafico riportante le quantità di azioni che sono in attesa nei due lati *del book* in un dato istante. In questo grafico, attraverso l'ultima *probe* dell'*Observer* (*showBuySellSharesNumberInOpeningOrClosingAuctions*) è possibile evidenziare anche il quantitativo di azioni, in attesa di controparte, alla fine della fase di apertura; *saveBookLogData* crea quattro *file* di testo in cui sono salvati i dati descritti precedente.
- 4) *showAgentWealthGraph* crea un grafico riportante sia la ricchezza media sia la minima e la massima di ogni famiglia di agenti presenti nella simulazione. Ogni agente è dotato di un proprio sistema di contabilità per cui a fine giornata è a conoscenza del numero di azioni che possiede e di una loro valorizzazione ai prezzi di mercato. Si tenga presente che come in Sum ogni agente ha la possibilità di indebitarsi illimitatamente ed anche di vendere allo scoperto le azioni trattate sul mercato. La *probe saveAgentWealthData* permette il salvataggio dei dati rappresentati graficamente su file di testo denominati come il tipo dell'agente a cui si riferiscono.

- Le ultime *probe* presenti nella figura precedente, sono quelle inserite per fare in modo che l'utente del programma abbia a disposizione dati che permettono lo studio della dinamica di formazione dei prezzi. È così possibile sia riprodurli graficamente, sia salvarli su *file* di testo¹⁴.



Dalla finestra del *Model* è possibile stabilire quanti agenti interagiscono col *book*, e soprattutto quali tipi e in che proporzioni.

Nell'ultima versione di JavaSum sono presenti alcune delle famiglie di agenti di Sum-0.66, tenendo quindi aperte ulteriori possibilità di sviluppo in questa direzione del *porting*.

Il primo tipo è stato introdotto da Agaglate (2004) e sono i *random agents* già citati nella descrizione del modello in Objective C.

¹⁴ Nel paragrafo 4 del capitolo 5 si parla in modo approfondito di questo argomento.

Questi agenti hanno come unica informazione ultimo prezzo formatosi sul mercato.

Il prezzo che propongono è calcolato moltiplicando l'ultimo prezzo osservato per un numero casuale compreso in un *range* stabilito dall'utente attraverso le probe *minCorrectinCoeff*, *maxCorrectinCoeff* e *asymmetricRange*. È plausibile impostare il primo valore non inferiore a 0.95 e il secondo non superiore a 1.05, potendo così automaticamente rispettare la regola, imposta sul mercato azionario, che vieta una differenza di prezzo superiore al 5% tra un contratto e quello successivo.

L'inserimento di questa flessibilità nel decidere il campo di variazione, può essere utilizzato, ad esempio, per osservare come cambia la variabilità dei prezzi nel caso in cui Borsa Italiana decidesse di aumentare o diminuire il parametro imposto.

Se il prezzo è positivo, l'ordine è in acquisto, mentre se è negativo è in vendita: la scelta tra uno e l'altro ha probabilità 0.5. La quantità da proporre è anch'essa stabilita in modo casuale, semplicemente tirando a sorte un numero intero compreso tra 0 e *maxOrderQuantity*, parametro stabilito dall'utilizzatore di JavaSum attraverso la rispettiva *probe* nella finestra del *Model*. La prima differenza di questo modello rispetto a Sum, è il sistema di inserimento degli ordini: con JavaSum gli agenti propongono un singolo ordine per una quantità pari a n azioni, invece di inserire n ordini unitari¹⁵.

Le prossime tre famiglie di agenti sono state introdotte da Mencarelli (2004), e rispettivamente sono:

- i *market imitating agents*;
- i *locally imitating agents*;
- gli *stop loss agents*.

I primi non hanno come informazione solo l'ultimo prezzo formatosi sul mercato, ma conoscono anche i prezzi medi delle due giornate precedenti. Questi agenti non hanno la possibilità di acquistare e vendere con probabilità 0.5, la scelta viene ancorata ad un valore indicato nella finestra del *Model*. Nel caso in cui il prezzo medio del giorno precedente sia superiore a quello antecedente ad esso, segnalando una tendenza all'acquisto, questo tipo di agente, propone un ordine in acquisto con probabilità pari a *asymmetricBuySellProb*. Viceversa la probabilità diminuisce divenendo pari a $1 - \text{asymmetricBuySellProb}$.

La seconda tipologia utilizza indicativamente lo stesso metodo di scelta, basando però la propria decisione sul comportamento degli altri agenti, nel giorno in cui inserisce l'ordine e non nei precedenti. Se il metodo *getLocalHistory* del *book* fornisce un valore positivo, per cui vi è una maggiore quantità di proposte in acquisto, l'agente ha una probabilità di acquisto pari a

¹⁵ Si veda il capitolo 5.

asymmetricBuySellProb; se invece è negativo, segnalando il caso opposto la probabilità sarà $1 - \textit{asymmetricBuySellProb}$.

Il prezzo, come per tutti gli altri tipi di agenti finora descritti, è calcolato moltiplicando l'ultimo prezzo per un valore casuale compreso tra $\textit{minCorrectinCoeff} + \textit{asymmetricRange}$ e $\textit{maxCorrectinCoeff} + \textit{asymmetricRange}$. Nel caso in cui l'ultimo prezzo sia sceso al di sotto di *floorP*, stabilito dall'utente, il modello imposta la probabilità di un acquisto da parte degli agenti a *agentProbToActBelowFloorP*, fissata attraverso la finestra del *Model*.

L'ultima tipologia che partecipa al mercato borsistico virtuale sono gli *stop loss agents*. I parametri che ne influenzano il comportamento, e che l'utilizzatore del programma può variare, sono:

- *maxLossRate*;
- *stopLossInterval*;
- *ceckingIfShortOrLong*.

Per questi agenti il valore del prezzo e della quantità è stabilito esattamente come i *random agents*. L'unica differenza si trova nella scelta tra proposta in acquisto o in vendita. Il meccanismo di decisione è così strutturato:

1. nel caso in cui *ceckingIfShortOrLong* sia pari a 0, ad indicare agenti che non considerano se hanno quantità positive o negative di azioni, il prezzo sarà positivo (acquisto) se l'ultimo prezzo formatosi sul mercato è maggiore o uguale a $\textit{stopLossMeanPrice} * (1 + \textit{maxLossRate})$. Il primo valore si riferisce al prezzo medio di *stopLossInterval* giorni prima, mentre il secondo indica il tasso di perdita sopportato dall'agente. Viceversa il prezzo sarà negativo (vendita) se l'ultimo prezzo è minore o uguale a $\textit{stopLossMeanPrice} * (1 - \textit{maxLossRate})$.
2. nel caso in cui *ceckingIfShortOrLong* sia pari a 1, l'agente considera la propria posizione lunga o corta sull'azione trattata. Il meccanismo è esattamente quello descritto precedentemente con l'unica variante che per poter essere un prezzo positivo (acquisto), la quantità di azioni possedute deve essere minore di 0. Viceversa sarà una proposta in vendita se si avrà anche una quantità positiva di azioni, oltre ad un ultimo prezzo pari al valore descritto al punto precedente.

La probe *agentNumber* anche se modificata, non comporta nessuna variazione nel modello di simulazione, in quanto non appena si avvia il programma, ad essa è assegnato il valore della somma delle *probes* che la precedono (indica il numero totale degli agenti). È inserita solo a titolo informativo come *dayNumber* il cui compito è indicare il giorno di simulazione corrente.

La sonda *istantNumber*, indica il numero di istanti che compongono una giornata. A differenza di Sum-0.66 gli istanti possono differire dal numero degli agenti. Come già detto all'inizio del capitolo, nel modello scritto in

Objectiv C, gli agenti operavano uno alla volta e la giornata finiva non appena a tutti fosse stato chiesto se volevano inserire una proposta nel *book*.

Grazie ad Agagliate (2004), nel nuovo modello, è possibile far operare per ogni istante un solo agente, una parte oppure tutti, scegliendo anche se il numero di agenti che opera per *tick* è fissa o variabile entro un *range* prestabilito.

Le *probe* interessate sono:

- *percentageOfOperatingAgents*, che indica la percentuale di agenti che devono operare per *tick*;
- *maxNumberOfOperatingAgents*, che indica il numero massimo di agenti che opera se la percentuale è considerata variabile;
- *typeOfPercentage* se uguale a *false* rende la percentuale di agenti fissa, viceversa se uguale a *true* la modifica in variabile tra 0 e *maxNumberOfOperatingAgents*.

Il parametro *agentProbToActInOpeningOrClosingAuctions* permette di stabilire a priori con che probabilità gli agenti operano durante le fasi d'asta. Molto simile la *probe agentProbToActWithMarketPrice* in quanto stabilisce con che probabilità l'agente può piazzare ordini a prezzo d'asta nelle fasi d'asta, e ordini al meglio, con la caratteristica di essere della tipologia ECO (esegui comunque in modo che l'ordine sia chiuso a qualsiasi prezzo) durante la contrattazione continua.

L'ultima *probe* rende possibile la scrittura a video di alcune informazioni relative agli ordini che vengono inseriti. Può avere per ora tre valori:

- se uguale a 0 non è scritto nulla ovviamente aumentando la velocità di esecuzione del programma;
- se uguale a 1 è il *book* che scrive vari dati (l'ordine che arriva, la composizione dei due lati, la tabella per il calcolo del prezzo teorico ed il prezzo di riferimento);
- se è uguale a 2 è l'agente che scrive alcuni dati (il tipo di agente, il suo indirizzo di memoria, l'ordine che sta inserendo ed altri valori a seconda della tipologia di agente che è stato interpellato).

Altra innovazione rispetto a Sum è la possibilità di visualizzare tutte le variabili di un determinato agente durante la simulazione. Per poter aprire le finestre che permettono di osservare internamente cosa sta elaborando l'agente, è sufficiente indicare il numero dell'agente che si vuole osservare e successivamente cliccare su *openProbeTo:* . Il tutto può essere attivato anche durante la simulazione.

L'ultima finestra messa a disposizione dal modello è quella che riporta il prezzo corrente. Il motivo per cui è citata per ultima non è assolutamente la minore importanza rispetto a tutto quello che finora si è detto.

L'unico motivo è l'indipendenza dalle scelte dell'utilizzatore del modello. Pur essendo uno degli strumenti che l'*Observer* mette a disposizione, questa è l'unica finestra che sarà sempre visibile. Il prezzo e la sua dinamica sono la sintesi di tutto ciò che avviene durante la simulazione, ed averlo sempre a disposizione, consente al ricercatore di riflettere non solo ex-post ma anche mentre il programma è in esecuzione.

La finestra, sotto riportata, non presenta alcuna differenza strutturale rispetto agli altri grafici prodotti dall'*Observer*.



Nel prossimo capitolo sarà descritto in modo dettagliato, a livello di codice sorgente, i *book* dei due modelli di simulazione di borsa Sum e JavaSum.

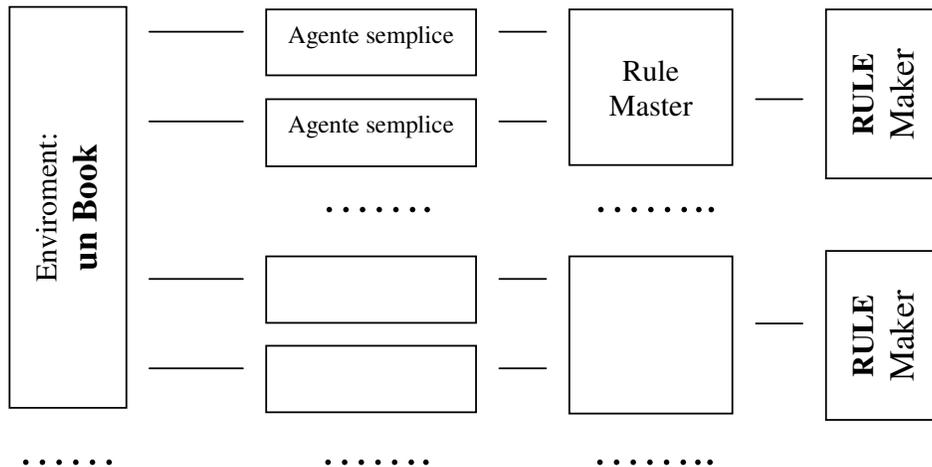
CAPITOLO 5 IL BOOK DI JAVASUM

5.1 PORTING IN JAVA.

Il nuovo programma di simulazione del mercato di borsa nasce sulla base del modello Sum scritto in Objective C. Il metodo che utilizzo per effettuare il *porting* in Java, passa da una prima fase di comprensione del codice di Sum in modo tale da comprendere ciò che internamente il programma effettua. In questo modo si hanno due tipi di vantaggi: uno è quello di avere delle linee guida da seguire che dovrebbe rendere la creazione del programma più agevole e soprattutto ordinata, ed il secondo sta nella più facile individuazione di possibili miglioramenti da apportare.

Per rendere il codice meno complesso, si segue nuovamente lo schema ERA (Enviroment-Roules-Agents) già adottato nella realizzazione di Sum e studiato per essere utilizzato come schema di riferimento per la costruzione di simulazione per le scienze sociali (Gilbert e Terna, 2000).

Lo schema, inizialmente, risulta essere il seguente:



Utilizzando questo schema si facilita la scrittura del codice perché si fa in modo che gli agenti non comunichino tra loro, ma attraverso l'ambiente, rappresentato nel nostro caso dal *book*.

Il modello inizialmente ha un solo *book*, ma fin dall'inizio il programma sarà strutturato in modo da poter inserire contrattazioni su più titoli, cercando di avvicinarsi sempre di più al mercato reale. Il primo obiettivo da raggiungere è la riscrittura del codice del *book* di SUM in linguaggio Java in cui non sono previste né aste di apertura né aste di chiusura. Una prima riflessione che deve

essere fatta è se considerare i *book* come oggetti a se stanti oppure come facenti parte di una classe che racchiuda i vari elementi comuni tra i *book*, ad esempio le regole di contrattazione e di formazione dei prezzi. Utilizzando un linguaggio di programmazione ad oggetti come Java la scelta non può che essere di costruire il *book* come esemplare di una classe in modo da sfruttare nel modo migliore una delle caratteristiche principali di questo tipo di programmazione: l'ereditarietà. Stabilite quali sono le caratteristiche generali, che la classe *book* deve avere, saranno poi generati i vari oggetti *book* che permetteranno l'effettiva contrattazione tra gli agenti.

Quando si costruiscono modelli di simulazione, è necessario poter tenere sotto controllo gli errori che possono verificarsi nella stesura del codice. È quindi opportuno poter testare passo dopo passo ciò che è stato prodotto, a livello di codice, e non solo dopo aver scritto gran parte del programma. A differenza dei modelli matematico-statistici, con le simulazioni si tende ad incorporare anche componenti non lineari, che potrebbero rendere difficile, se non impossibile, l'individuazione di errori dai risultati ottenuti (Gilbert e Terna, 2000). Per ovviare a questa problematica fin dall'inizio sarà necessario creare degli agenti fittizi che interagiscano col *book* in modo tale da verificare se tutto procede come desiderato.

L'oggetto delle tesi di Marco Agagliate e di Bruno Mencarelli è stato invece lo sviluppo delle prime classi di agenti che operano nel mercato di borsa simulato. Anch'essi risultano essere esemplari di un'unica classe dalle quale ereditano le parti di codice a loro necessari per operare. Come previsto dallo schema ERA, gli agenti sono costruiti in modo che prima di effettuare le loro scelte interrogano il *Rule Master*, contenente le regole di decisione, e che a sua volta riceve i suoi dati dal *Rule Maker* che viene utilizzato per modificare le regole di decisione (molto importante ad esempio per gli agenti cognitivi che apprendono da ciò che accade con l'evolversi del tempo). Una fondamentale caratteristica degli agenti è la capacità di tenere una loro contabilità in modo tale da conoscere sempre ciò che hanno effettuato nel passato e la loro ricchezza.

Questo tipo di organizzazione della struttura interna della simulazione permette di rendere molto flessibile il programma, in modo tale che in futuro, se si vorrà, si potrà inserire nuovi tipi di agenti o modificare in modo agevole quelli esistenti.

Il *book*, nella programmazione, risulta essere un oggetto in attesa che un agente attivi un suo metodo. Una volta ricevuto l'ordine di acquisto o di vendita deve essere in grado di controllare se vi è contropartita, nel caso in cui non fosse possibile accoppiare l'ordine appena ricevuto, accoda l'ordine lasciandolo in attesa che ne arrivi uno compatibile seguendo le regole dettate dal regolamento di Borsa Italiana s.p.a.

Internamente l'oggetto *book* possiede due vettori all'interno dei quali inserisce i vari ordini:

1. quello degli acquisti è ordinato in modo decrescente rispetto al prezzo, in modo tale che in prima posizione vi sia sempre il miglior ordine (in

caso di parità di prezzo si considera il momento di ricevimento dell'ordine);

2. in modo analogo il secondo vettore delle vendite è ordinato in senso crescente per avere in prima posizione l'ordine di vendita con prezzo inferiore.

5.2 IL BOOK DI SUM.

In questo paragrafo sono trattate in modo approfondito le istruzioni del codice riguardante il *book* della versione di Sum-0.66.

Le classi, con la programmazione in Objective C, possono essere costruite interamente in un *file*, oppure possono essere costituite da due *file*: uno di interfaccia (nome_classe.h), nel quale sono dichiarate tutte le variabili istanza e i metodi della classe che si sta costruendo (definisce come un oggetto deve interagire con gli altri elementi del programma); ed uno di implementazione (nome_classe.m) all'interno del quale si scrive il codice che effettivamente costituisce il programma.

In Sum la via scelta è quella della scissione in due *file* e sono quindi presenti *book.h* e *book.m* di cui ora commento le parti di codice in modo dettagliato.

5.2.1 BOOK.H

```
// Book.h  
  
#import <objectbase/SwarmObject.h>  
#import <collections.h>  
#import "Matrix2.h"
```

Le parti di codice precedute dalla doppia *slash*, non sono considerate in fase di compilazione e vengono utilizzate per inserire dei commenti: in questo caso indica semplicemente il nome del *file* che si sta osservando.

Le righe successive servono per poter usare all'interno della nostra classe metodi definiti in altre classi o librerie. “*objectbase/SwarmObject.h*” serve per utilizzare le *probe*, cioè le sonde che permettono all’*Observer* di vedere i dati, ed è anche importante per la creazione degli oggetti (grazie a questo è possibile utilizzare il metodo *createEnd*). La seconda riga è utilizzata per importare una classe di *Swarm*, “*collections*” che serve per gestire le collezioni di oggetti (liste di oggetti). “*Matrix2.h*” è, invece, l’interfaccia di una classe, creata in Sum, per la gestione delle matrici e dei vettori.

```
@interface Book: SwarmObject
```

Questa riga indica l’inizio dell’interfaccia del *book*. Dopo i “:” è indicata la classe da cui si eredita in modo tale che le variabili istanza e i metodi, della classe ereditata, possano essere utilizzati all’interno della nuova classe. Se

viene indicata la classe da cui si eredita è necessario, come visto poco prima, che l'interfaccia di quest'ultima sia importata.

```
{  
    id <Index> agentArrayIndex;  
    Matrix2 * sellOrderStorehouse, * buyOrderStorehouse, *  
    meanPriceHistory,  
        * localHistory;  
    int sellOrderNumber, buyOrderNumber, maxOrderQuantity,  
        agentNumber, printing, meanPriceHistoryLength,  
    localHistoryLength,  
        nAheadForecasting;  
    float price, executedPrice, meanPrice, currentMeanPrice,  
        previousClosingPrice;  
    int count;  
}
```

Segue poi un corpo unico, compreso fra graffe, nel quale vengono dichiarate le variabili istanza, indicando di che tipo sono, utilizzate all'interno della classe *book*. Nella definizione delle variabili, "id" risulta essere un puntatore ad un oggetto, o più precisamente ai dati dell'oggetto, per cui *agentArrayIndex* viene definito come oggetto, ma non si fornisce nessuna informazione supplementare sul tipo dei suoi metodi o delle sue variabili istanza. "<Index>" indica che possono essere utilizzati tutti i metodi della classe *Index*.

La struttura Nome_classe * nome_variabibile serve per la tipizzazione statica delle variabili. In definitiva sono dei puntatori alla classe *Matrix2* e questo serve per poter fornire al compilatore maggiori informazioni sul tipo di oggetto definito.

Vi è poi ancora un tipo di dichiarazione delle variabili istanza (tipo_variabibile nome_variabibile) che fornisce di che tipo è la variabile.

La parte successiva del *file* dichiara tutti i metodi che possono essere utilizzati.

```
- createEnd;  
  
- setAgentArrayIndex: i;  
- setAgentNumber: (int) n;  
- setMaxOrderQuantity: (int) m;  
- setMeanPriceHistoryLength: (int) l;  
- setLocalHistoryLength: (int) l;  
- setPrinting: (int) p;  
- setMeanPrice;  
  
- setClean;  
- setOrderBeforeOpeningFromAgent: (int) n atPrice: (float) p;  
- setOrderFromAgent: (int) n atPrice: (float) p;  
- setNAheadForecasting: (int) na;  
  
- (float) getPrice;  
- (float) getPreviousClosingPrice;
```

```

- (float) getMeanPrice;
- (float) getLaggedMeanPrice: (int) lag; // the methods
                                         //
'getLaggedMeanPrice: 1' and
                                         // 'getMeanPrice give
the same
                                         // result
- (float) getMeanPriceIndex;
- (int) getLocalHistory; // the method returns
                        // a positive int
if locally
                        // see localHistoryLength) the # of
buying decisions
                        // is greater than the # of selling
decisions
                        // a negative int
in the
                        // opposite case
- (float) getSellOrderNumber;
- (float) getBuyOrderNumber;

@end

```

Le righe precedute dal segno “-” indicano metodi che possono essere utilizzati solo dalle istanze della classe.

Il metodo *createEnd* appartiene a *Swarm* e serve per creare un oggetto. Se il nome del metodo è preceduto dalla parola “set” indica che è utilizzato per assegnare un valore ad un oggetto; quelli preceduti dal “get” invece forniscono un valore del tipo indicato tra parentesi tonde. Cosa effettivamente fanno i vari metodi è spiegato nell’implementazione del book.

La fine dell’interfaccia è indicata da “@end”.

5.2.2 BOOK.M

```

// Book.m

#import "Book.h"
#import "BasicSumAgent.h" // this is placed here to avoid a
circular call

```

Come per il *file* precedente, la prima riga non è considerata dal compilatore in quanto è un commento che riporta semplicemente il nome del *file*. Subito dopo sono importate due interfacce: la prima è quella della classe di cui si effettuerà l’implementazione, mentre la seconda si riferisce alla classe *BasicSumAgent* che è ereditata da tutti gli agenti e che serve per comunicare col *book* (ad esempio per comunicare il prezzo di esecuzione del contratto ovvero *executedPrice*).

Successivamente vengono definite tre funzioni:

```
// functions

void message(id <Index> agentArrayIndex, int n, float p)
{
    BasicSumAgent * anAgent;

    [agentArrayIndex setOffset: n-1];
    anAgent=[agentArrayIndex get];
    [anAgent setConfirmationOfExecutedPrice: p];
    return;
}
```

La prima funzione è utilizzata per informare l'agente dell'esito del suo ordine. Gli argomenti che si passano alla funzione sono:

- *id <Index> AgentArrayIndex* che serve al *book* per sapere dove si trova in memoria l'agente, altrimenti non sarebbe possibile fargli arrivare il messaggio. Questo oggetto è necessario in quanto il programma crea una lista di agenti che è riordinata casualmente di giorno in giorno in modo tale che non sia sempre lo stesso agente ad operare per primo. L'*array* degli agenti, invece, rimane sempre nello stesso ordine in cui è creato all'inizio della simulazione, e quindi, per poter ritrovare l'agente tramite il suo numero identificativo, è necessario conoscere l'indice dell'*array*.
- *n* è il numero identificativo dell'agente
- *p* è il prezzo di conclusione del contratto.

Nel corpo della funzione si definisce subito una variabile locale *anAgent* come puntatore alla classe *BasicSumAgent*. È necessario poi, come precedentemente detto, ricercare l'indirizzo dell'agente ricordando di "bilanciare" gli indici, col metodo *setOffset*, visto che l'agente 1 si trova in posizione 0 (motivo del passaggio da *n* a *n-1*). Trovato l'indirizzo lo si assegna alla variabile *anAgent* al quale verrà inviato il messaggio di assegnazione del valore *p* attraverso il metodo *setConfirmationOfExecutedPrice* della classe *BasicSumAgent*.

La seconda funzione è così strutturata:

```
void setLocal(Matrix2 * loc, float p)
{
    int i;
    if (p==0)return;
    for (i=[loc getRows]-2;i>=0;i--)[loc R:i+1 C:0 setFrom: [loc
R:i C:0]];
    [loc R:0 C:0 setFrom: p];
    return;
}
```

ed è utilizzata per aggiornare il vettore, di lunghezza definita dall'utente tramite l'interfaccia utente del *Model* alla voce *localHistoryLength*, dei prezzi proposti: se il prezzo passato alla funzione è uguale a 0 non capita nulla ed esce dalla funzione. Con *p*≠0 viene utilizzato il metodo *getRows* (della classe

Matrix2) per conoscere il numero di righe del vettore a cui l'argomento tipizzato *loc* punta (si vedrà in seguito che è il vettore *localHistory*) e lo si utilizza per creare un ciclo *for* che sposti tutti i valori di una posizione in modo da inserire il nuovo valore nella serie storica. Il metodo *setFrom* (di *Swarm*) indica il valore da inserire e [loc R:i C:0] fornisce il riferimento di riga e di colonna, in questo caso del vettore, ma può essere utilizzato anche con le matrici.

L'ultima funzione non è di tipo *void* come le due precedenti, ma restituisce un valore intero:

```
int getLocal(Matrix2 * loc)
{
    int i, tot;
    tot=0;
    for (i=0;i<[loc getRows];i++)
    {
        if([loc R:i C:0]>0)tot++;
        if([loc R:i C:0]<0)tot--;
    }
    return tot;
}
```

Alla funzione si passa un solo argomento tipizzato *loc* (puntatore alla classe Matrix2) e, nel suo corpo, sono definite due variabili locali, di cui una inizializzata a 0. L'unico compito che assolve, è quello di incrementare la variabile "tot" di uno se il valore che trova è maggiore di 0 e lo decremента della stessa quantità se il valore è negativo. Più avanti nel *file* si vedrà che il vettore che gli viene passato come argomento è *localHistory*, e serve per lo più per fornire un dato locale (solo su una parte di serie di prezzi proposti) che indica se i prezzi in acquisto sono in quantità superiori a quelli di vendita o viceversa. In altri termini il valore che si ottiene indica se gli ordini di acquisto sono superiori a quelli di vendita o viceversa e questo dato serve agli agenti che imitano localmente e in modo casuale il mercato (*LocallyImitatingAgent*).

```
@implementation Book
```

Questa riga indica che inizia la realizzazione del *book*. La prima parte è soltanto la definizione di alcuni metodi che servono per l'assegnazione di valore alle variabili, e ritornano il loro indirizzo di memoria (return self).

```
- setAgentArrayIndex: i
{
    agentArrayIndex = i;
    return self;
}
```

Il primo metodo assegna il valore "i" alla variabile *agentArrayIndex* e restituisce il suo indirizzo di memoria. In questo modo assegnando il numero dell'agente possiamo comunicare con lui.

```
- setAgentNumber: (int) n
{
  agentNumber=n;
  return self;
}
```

In questo caso viene restituito l'indirizzo di memoria dove si trova il numero degli agenti. Questo metodo serve per dimensionare le due matrici che conterranno gli ordini di acquisto e di vendita, visto e considerato che ogni agente può operare una sola volta sul mercato per ogni giorno. Al momento della costruzione della matrice si utilizza il metodo per conoscere dove andare a trovare questo dato.

```
- setMaxOrderQuantity: (int) m
{
  maxOrderQuantity=m;
  return self;
}
```

Anche questo ha lo stesso scopo del precedente, e fornisce l'indirizzo di memoria per trovare la quantità massima trattata da ogni agente (sempre per dimensionare le matrici)

```
- setMeanPriceHistoryLength: (int) l
{
  meanPriceHistoryLength=l;
  return self;
}
```

Con questo metodo, *il book*, è in grado di fissare la lunghezza del vettore dei prezzi medi. A fine giornata viene calcolato il prezzo medio da inserire nel vettore dei prezzi medi che sarà lungo *meanPriceHistoryLength*.

```
- setLocalHistoryLength: (int) l
{
  localHistoryLength=l;
  return self;
}
```

localHistoryLength fornisce l'indirizzo di memoria dove trovare il valore della lunghezza del vettore utilizzato per calcolare se vi sono preponderanza di prezzi in acquisto rispetto a quelli di vendita o viceversa. Serve, come visto in precedenza, per le decisioni degli agenti che imitano localmente il mercato.

```
- setPrinting: (int) p
{
  printing=p;
  return self;
}
```

```
}

```

L'ultimo metodo di assegnazione è un'opzione di stampa. Nel *book* è considerato solo il caso che sia uguale a 1, e se così fosse, scrive a video che ha ricevuto un ordine.

A questo punto si costruiscono degli oggetti necessari al *book*, utilizzando il metodo *createEnd* di *Swarm*.

```
- createEnd
{
    int i;
    [super createEnd];

    executedPrice=1; // this is the starting price; it seems to
be                               // not relevant at all for the behavior of
the model
    meanPrice=1;
    previousClosingPrice=executedPrice;
    currentMeanPrice=0;
    count=0;

    // the book works on the basis of two matrixes containing
sell
// order in increasing order or buy order in decreasing
order
// (in col 1 we have the orders; in col 2 the number of the
agent
// placing the order)

// if an order obtains an immediate matching, it is not
filed

// the worse situation is that of having all the order on
one side of
// the market and all the agents ordering; so the rows of
the two
// matrixes must be equal to the number of the agents

    sellOrderStorehouse=[Matrix2 createBegin: [self getZone]];
    [sellOrderStorehouse
agentNumber*maxOrderQuantity
                                setDimensionRows:
                                Cols: 2 Code: 1];
    sellOrderStorehouse=[sellOrderStorehouse createEnd];

    buyOrderStorehouse=[Matrix2 createBegin: [self getZone]];
    [buyOrderStorehouse
agentNumber*maxOrderQuantity
                                setDimensionRows:
                                Cols: 2 Code: 2];
    buyOrderStorehouse=[buyOrderStorehouse createEnd];

```

```

    meanPriceHistory=[Matrix2 createBegin: [self getZone]];
    [meanPriceHistory setDimensionRows: meanPriceHistoryLength
                        Cols: 1 Code: 3];
    meanPriceHistory=[meanPriceHistory createEnd];
    // mean prices will be stored by rows; now we fill all the
r. with
    // the starting mean price
    for (i=0;i<=meanPriceHistoryLength-1;i++)
        [meanPriceHistory R:i C:0 setFrom: meanPrice];

    localHistory=[Matrix2 createBegin: [self getZone]];
    [localHistory setDimensionRows: localHistoryLength
                    Cols: 1 Code: 4];
    localHistory=[localHistory createEnd];
    // local actions will be stored by rows; we fill all the r.
with
    // 0, i.e. 'no action', automatically, as a byproduct of the
    // setDimensionRows:Cols: method

    return self;
}

```

Nella prima parte è dichiarata una variabile locale di tipo intero “i” e si fa in modo, con la seconda riga di codice del metodo, che *createEnd* sia cercato nella super classe. Vengono poi inizializzate le variabili che il *book* utilizza per calcolare alcuni dati.

Il passo successivo è quello che crea la struttura vera e propria del *book*, che come già affermato, risulta essere composto da due matrici di raccolta degli ordini: una per gli acquisti, caratterizzati nella simulazione da un prezzo positivo e una per le vendite, identificate da un prezzo negativo. Vengono anche creati due vettori: quello del prezzo medio storico e quello di una parte della serie storica dei prezzi proposti, che, come già ricordato, servirà per gli agenti che imitano localmente il mercato.

Tutti questi oggetti sono costruiti con la medesima sequenza di istruzioni: al nome dell’oggetto, creato dalla classe *Matrix2*, viene assegnata una zona di memoria. Al nuovo oggetto viene poi inviato un messaggio contenente un metodo di *Matrix* che dimensiona la matrice e assegna ad essa un codice identificativo.

Per gli ordini, il numero delle righe è calcolato tramite il prodotto tra il numero degli agenti per la quantità massima di ordini che possono eseguire, mentre il numero delle colonne è 2: la prima ospita i numeri identificativo degli agenti, mentre la seconda i prezzi proposti.

Dopo il dimensionamento delle matrici viene conclusa l’inizializzazione con la riga:

```
nome_oggetto = [nome_oggetto createEnd]
```

Al vettore del prezzo medio storico, a differenza degli altri casi analizzati ora, si assegna ad ogni suo elemento un valore iniziale, pari a *meanPrice*

(inizializzato prima a 1), utilizzando il metodo di *Swarm setFrom* abbinato ad un ciclo *for*.

Dopo l'inizializzazione degli oggetti, sono realizzati altri metodi divisi per periodo di utilizzo durante la giornata di borsa.

```

    // at the end of each day
- setMeanPrice{
    if (count>0) meanPrice=currentMeanPrice/count; // otherwise
we keep                                     // previous
value
    return self;
}

- setNAheadForecasting: (int) na
{
    nAheadForecasting = na;
    return self;
}

```

Due metodi alla fine di ogni giornata:

- uno per il calcolo del prezzo medio, dividendo *currentMeanPrice* (la sommatoria dei prezzi formatisi nella giornata) per la variabile *count* (contatore dei contratti conclusi).
- e uno di assegnazione di valore alla variabile *nHaeadForecasting*, necessaria per il calcolo dell'indice di prezzo medio.

```

    // at the beginning of each day
- setClean{
    int i;
    sellOrderNumber=0; buyOrderNumber=0;

    // meanPriceHistory in row 0 contains the t-1 meanPrice;
    //                    in row 1 contains the t-2 meanPrice;
    //                    etc.
    for (i=meanPriceHistoryLength-2;i>=0;i--)
        [meanPriceHistory R:i+1 C:0 setFrom:
            [meanPriceHistory R:i
C:0]];
    [meanPriceHistory R:0 C:0 setFrom: meanPrice];

    previousClosingPrice=executedPrice; // the last one of
'yesterday'
    currentMeanPrice=0;
    count=0;
    return self;
}

```

All'inizio di ogni giorno si attiva il metodo *setClean* utilizzato per pulire il *book*.

Definita una variabile locale "i", si procede con l'azzeramento del numero degli ordini in acquisto e in vendita. Le istruzioni successive fanno in modo che il vettore della serie storica dei prezzi sia aggiornato: vengono spostati gli elementi verso il basso di una posizione, perdendo così l'ultimo dato ed acquisendo, in prima posizione, il dato calcolato in chiusura col metodo *setMeanPrice* visto in precedenza.

Le ultime righe servono per assegnare il valore del prezzo dell'ultimo contratto concluso alla variabile del prezzo di chiusura precedente (non essendoci in questa versione di Sum l'asta di chiusura), l'azzeramento della sommatoria dei prezzi formati e il contatore dei contratti conclusi.

Prima che la giornata passi alla fase di asta continua, è prevista una fase di pre-apertura durante la quale alcuni degli agenti, e con una certa probabilità, hanno la possibilità di inserire nel *book* ordini di vendita o di acquisto. I contratti, in questa fase non sono conclusi perché, come già detto, non vi è una vera e propria asta di apertura come nel mercato reale. Questa fase serve solo per far sì che il *book* ad inizio giornata non sia vuoto, in modo da creare continuità nella formazione dei prezzi.

Osserviamo ora il codice di questa fase:

```
// receiving an order before opening from an agent
- setOrderBeforeOpeningFromAgent: (int) n atPrice: (float)
  p

{
  int number;

  number = n;
  price = p;

  if(printing==1)
  printf(
  "The book received a before opening order from agent #%3d at
price %7.4f\n", number, price);
```

Dopo una riga di commento, vi è il codice relativo al metodo utilizzato dall'agente che opera in questa fase, e sono indicati gli argomenti che gli vengono passati: numero identificativo dell'agente e prezzo proposto. Vengono poi definite alcune variabili locali e, nel caso in cui sia attiva una certa opzione di stampa, si prevede la scrittura di una frase a video. Il messaggio che si scrive a video è la descrizione dell'ordine appena ricevuto.

```
// local history
setLocal(localHistory, price);
```

Utilizzando la seconda funzione definita all'inizio del *file*, viene inserito il valore del prezzo nel vettore contenente una parte della serie storica dei prezzi proposti, utilizzato dagli agenti che imitano localmente il mercato.

```
// if price==0 no action required, but sending a 0.0 message
to the agent
if(price==0) message(agentArrayIndex, number, 0.0);

// the agent is selling at min price '-price'
if(price<0) {
    message(agentArrayIndex, number, 0.0);
    // filing the sell order, in increasing
order
    sellOrderNumber++;
    [sellOrderStorehouse fileIncreasingP: -
price
                                andN: (float)
number
                                usingAsNumberOfRows:
sellOrderNumber];
    if(printing==1)
    [sellOrderStorehouse printNRows:
sellOrderNumber];
}

// the agent is buying at max price 'price'
if(price>0) {
    message(agentArrayIndex, number, 0.0);
    // filing the buy order, in decreasing
order
    buyOrderNumber++;
    [buyOrderStorehouse fileDecreasingP: price
number
                                andN: (float)
                                usingAsNumberOfRows:
buyOrderNumber];
    if(printing==1)
    [buyOrderStorehouse printNRows:
buyOrderNumber];
}

return self;
}
```

Vengono poi previsti tre casi:

1. prezzo uguale a 0
2. prezzo maggiore di 0
3. prezzo minore di 0

a seconda che sia verificata una di queste condizioni, sono effettuate operazioni diverse.

Nel primo caso non capita nulla ma si invia ugualmente un messaggio all'agente, per mezzo della funzione "message" definita all'inizio, con il valore del prezzo uguale a 0.

Quando il prezzo è negativo significa che l'agente vuole vendere al prezzo minimo del valore di "-price". Il *book* aggiorna il contatore degli ordini di vendita (*sellOrderNumber*) e la proposta viene archiviata nella matrice *sellOrderStorehouse*, per mezzo del metodo *fileIncreasingP* della classe *Matrix2*, in modo che sia ordinata per valori crescenti di prezzo. La medesima cosa capita nell'ultimo caso, solo che il contatore aggiornato è quello degli ordini di acquisto (*buyOrderNumber*) e la matrice interessata è *buyOrderStorehouse*. In tutti e due gli ultimi casi, il messaggio inviato, all'agente che piazza l'ordine, ha prezzo = 0 perché non si concludono i contratti. Sia per il prezzo maggiore che per quello minore di 0, è prevista la scrittura a video delle N righe delle matrici degli ordini, sempre se l'opzione di stampa *printing* è uguale a 1.

Nella fase di asta continua le istruzioni non sono molto diverse da quelle appena illustrate.

```
// receiving an order when the market is open
- setOrderFromAgent: (int) n atPrice: (float) p
{
    int number;

    number = n;
    price = p;

    if(printing==1)
        printf("The book received an order from agent #%3d at price
%7.4f\n",
            number, price);

    // local history
    setLocal(localHistory, price);

    // if price==0 no action required, but sending a 0.0 message
to the agent
    if(price==0) message(agentArrayIndex, number, 0.0);

    // the agent is selling at min price '-price'
    if(price<0) {if (buyOrderNumber>0 &&
        [buyOrderStorehouse R: 0 C: 0] >= -price)
        {executedPrice=[buyOrderStorehouse R: 0 C:
0];
            currentMeanPrice+=executedPrice;
            count++;
```

```

        message(agentArrayIndex, number, -
executedPrice);
message(agentArrayIndex, (int) [buyOrderStorehouse R: 0 C: 1],
executedPrice);
        buyOrderNumber--;
        [buyOrderStorehouse shiftRowsDown:
buyOrderNumber];
        if (printing==1)
        [buyOrderStorehouse printNRows:
buyOrderNumber];
    }
    else {
        message(agentArrayIndex, number, 0.0);
        // filing the sell order, in increasing
order
        sellOrderNumber++;
        [sellOrderStorehouse fileIncreasingP: -
price
        andN: (float)
number
        usingAsNumberOfRows:
sellOrderNumber];
        if (printing==1)
        [sellOrderStorehouse printNRows:
sellOrderNumber];
    }
}

// the agent is buying at max price 'price'
if (price>0) {if (sellOrderNumber>0 &&
[sellOrderStorehouse R: 0 C: 0] <= price)
{executedPrice=[sellOrderStorehouse R: 0 C:
0];
        currentMeanPrice+=executedPrice;
        count++;
        message(agentArrayIndex, number,
executedPrice);
message(agentArrayIndex, (int) [sellOrderStorehouse R: 0 C: 1],
executedPrice);
        sellOrderNumber--;
        [sellOrderStorehouse shiftRowsDown:
sellOrderNumber];
        if (printing==1)
        [sellOrderStorehouse printNRows:
sellOrderNumber];
    }
    else {
        message(agentArrayIndex, number, 0.0);

```

```

        // filing the buy order, in decreasing
order
        buyOrderNumber++;
        [buyOrderStorehouse fileDecreasingP: price
        andN: (float)
number
                usingAsNumberOfRows:
buyOrderNumber];
        if (printing==1)
        [buyOrderStorehouse printNRRows:
buyOrderNumber];
        }
    }
    return self;
}

```

Fino al caso in cui il prezzo è uguale a 0 non vi è alcuna differenza, se non il nome del metodo attivato dagli agenti in questa fase (*setOrderFromAgent*). Se il prezzo è minore di 0, cioè prezzo in vendita al prezzo minimo “-price”, e il numero degli ordini di acquisto non è uguale a 0, viene fatto un controllo immediato sulla prima riga della matrice *buyOrderStorehouse* in modo da vedere se vi è un ordine con prezzo maggiore o uguale a quello appena proposto dall’agente. Questo meccanismo serve per utilizzare il prezzo migliore per chi ha fatto l’ultima offerta: supponendo che l’ordine di vendita sia al prezzo di 1 e che gli ordini di acquisto, presenti nel *book*, siano tre con rispettivi prezzi di 1, 2 e 3, il contratto è concluso al prezzo di 3.

Se è possibile accoppiare due ordini si aggiorna il prezzo di esecuzione, guardando quello in prima riga della matrice, si aggiorna la sommatoria dei prezzi per il calcolo del prezzo medio a fine giornata e il contatore dei contratti conclusi. Fatto ciò si inviano i messaggi ai due agenti interessati per mezzo della funzione *message*. Avendo accoppiato un ordine della matrice, il contatore delle proposte deve essere decrementato e la prima riga della matrice viene sovrascritta utilizzando il metodo della classe *Matrix2* che sposta le righe (*shiftRowsDown*). Se l’opzione di stampa è uguale a 1 viene riproposto a video la matrice degli ordini di vendita aggiornata.

Nel caso in cui non sia possibile accoppiare gli ordini perché il miglior prezzo di vendita è inferiore al prezzo minimo di acquisto, l’ordine si inserisce nella matrice allo stesso modo con cui si immettono quelli in fase di pre-apertura.

Le stesse cose capitano nel caso di ricezione di un ordine di acquisto, ovviamente con i controlli e le modifiche sulle matrici e i contatori opportuni.

Alla fine del *file book.m* sono vi sono i metodi che forniscono dei dati, in modo che questi possano essere utilizzati da altri metodi del programma:

```

- (float) getPrice
{
    return executedPrice;
}

```

Restituisce l'ultimo prezzo formato nel mercato, richiesto dagli agenti prima di operare.

```
- (float) getMeanPrice
{
    return meanPrice;
}
```

Fornisce il prezzo medio.

```
- (float) getLaggedMeanPrice: (int) lag
{
    return [meanPriceHistory R: lag-1 C: 0];
}
```

Restituisce il prezzo medio di “*lag*” giorni prima (stabilito dall'utente del programma inserendo nell'interfaccia del *Model* il valore di *meanPriceHistoryLength*).

```
- (float) getMeanPriceIndex
{
    return meanPrice/[meanPriceHistory R: nAheadForecasting C:
0];
}
```

Fornisce l'incremento o il decremento del prezzo medio rispetto a “*nAheadForecasting*” giorni prima (stabilito dall'utente tramite l'interfaccia del *Model*).

```
- (int) getLocalHistory
{
    return getLocal(localHistory);
}
```

Applicando la terza funzione definita all'inizio del *file*, confronta il numero dei prezzi proposti in acquisto con quelli in vendita, allo scopo descritto nel momento in cui abbiamo trattato la funzione *getLocal*.

```
- (float) getSellOrderNumber
{
    return (float) sellOrderNumber;
}

- (float) getBuyOrderNumber
{
    return (float) buyOrderNumber;
}
```

Restituiscono il valore del numero degli ordini in acquisto e in vendita.

```
- (float) getPreviousClosingPrice
{
    return previousClosingPrice;
}
```

Fornisce il prezzo di chiusura del giorno precedente.

@end

Conclude l'implementazione del *book*.

5.3 LA COSTRUZIONE DEL BOOK DI JAVASUM.

Prima di iniziare a costruire il *book*, sono state necessarie alcune riflessioni in modo tale da poter utilizzare al meglio ciò che un linguaggio di programmazione sconfinato come Java mette a disposizione.

Dalla lettura del paragrafo precedente si può dedurre che tutto il *book* è incentrato sulle due matrici che contengono gli ordini, sui loro meccanismi di inserimento dei dati e su come interagiscono per concludere i contratti. La libreria di *utility* di Java mette a disposizione una serie completa di modi di conservazione degli oggetti, ognuno dei quali ha caratteristiche distintive ed è corredata di vari metodi che potrebbero rendere più semplice e più efficiente le righe di codice del programma.

Per poter effettuare una scelta appropriata al caso, è importante passare in rassegna i vari modi (Eckel, 2002):

- L'*Array* è una sequenza lineare di primitive o di riferimenti ad oggetti, tutti dello stesso tipo, che ha come caratteristica principale la velocità di accesso agli elementi. Utilizzando questo oggetto si deve tener presente che, una volta inizializzato l'oggetto, la sua dimensione, risulta essere fissa e se si cerca di inserire nuovi elementi passando i limiti si ottiene una *RuntimeException*. Fa parte di questo oggetto solo il campo *length*, di sola lettura, che fornisce il numero di elementi possono essere inseriti (la lunghezza dell'*array*). I metodi di supporto agli *array* contenuti nella classe *Arrays* sono:
 1. *equals()* che verifica l'uguaglianza di 2 *array*;
 2. *fill()* per inserire un valore in un *array*;
 3. *sort()* per riordinare l'*array*;
 4. *binarySearch()* per cercare un elemento all'interno di un *array* ordinato;
 5. *asList()* che trasforma un *array* in un contenitore *List*

Pur avendo il grande vantaggio dell'accesso rapido ad ogni suo elemento, l'utilizzo di *array* nella costruzione delle due matrici che contengono gli

ordini non è adeguato in quanto non conosciamo la lunghezza che devono avere (il numero di ordini non è stabilito a priori ma dipende dal comportamento degli agenti). Si potrebbe pensare di sovradimensionare l'*array*, ma questa soluzione non è un modo efficiente per gestire la situazione in quanto, con programmi corposi come può risultare una simulazione di un mercato di borsa e anche se l'evoluzione tecnologica ci mette a disposizione computer potenti, potrebbe essere necessario ottimizzare l'utilizzo della memoria.

Le altre modalità di conservazione degli oggetti messe a disposizione da Java, presentano tutti il medesimo limite, che consiste nel non poter utilizzare direttamente le primitive ed è quindi necessario fare uso delle classi *wrapper* (Integer, Double, ecc.) con un incremento di difficoltà di programmazione non indifferente. I tipi wrapper servono per considerare come primitivi oggetti che non lo sono.

Nella libreria dei contenitori di Java vi sono due concetti distinti:

1. *Collection* che sono gruppi di singoli elementi. Ad questo appartengono i *List*, che contengono gli oggetti in una sequenza particolare (mantengono l'ordine di immissione), e i *Set*, il quale non può avere elementi uguali e possiede un proprio metodo di riordino interno.
2. *Map* che contiene coppie di oggetti chiave-valore come un piccolo data-base. Possono essere creati facilmente contenitori in più dimensioni, come avviene per gli *array*. Ha un proprio metodo di riordino interno e non accetta più di un elemento di ciascun tipo (in base alla chiave).

Nel caso del *book* potrebbe essere più agevole utilizzare un tipo di contenitore che abbia un suo metodo interno di riordino, visto e considerato che le proposte sono in ordine crescente o decrescente. Ma da quanto appena esposto poco prima, si deduce che questo tipo di contenitori (*Set* e *Map*) hanno il limite, nel nostro caso, di non poter contenere elementi uguali (due ordini allo stesso prezzo possono essere immessi nel *book*). L'attenzione si sposta verso l'ultimo tipo di contenitori disponibili cioè i *List*.

Pensando al meccanismo di ricezione di ordini del *book*, è evidente come questi siano essenzialmente dei gruppi di primitive (nel caso di Sum prezzo e numero dell'agente); questo comporta un incremento della difficoltà in quanto i contenitori accettano oggetti e si dovrebbe quindi utilizzare le classi *wrapper*.

Per ovviare si potrebbe utilizzare un contenitore di *array*, in modo tale da sfruttare la facilità di accesso agli elementi di un *array*, rendendo semplice sia la creazione di un metodo di riordino del contenitore, sia l'estrazione dei dati contenuti per poter accoppiare gli ordini secondo il regolamento di Borsa Italiana s.p.a..

Sulla base delle precedenti considerazioni su come rendere più efficiente la programmazione del *book*, la soluzione che mi è sembra più opportuna è quella di creare degli *arrayList* che contengano degli *arrays*, in modo tale da

scavalcare la difficoltà di trattazione delle primitive da parte dei contenitori e con l'ulteriore vantaggio di sfruttare le proprietà degli *arrays*.

Si aprono così due opportunità:

1. come accadeva in Sum, far passare dall'agente, al metodo di ricezione dell'ordine del *book*, più variabili (prezzo, proprio numero identificativo, ecc.) e creare solo successivamente i vari *arrays*.
2. oppure, passaggio al metodo di un *array* all'interno del quale sono presenti le variabili che sintetizzano l'ordine.

Questa seconda soluzione a mio avviso rende più flessibile l'utilizzo del *book*, perché, così facendo, il metodo che viene attivato dall'agente ha la possibilità di ricevere anche ordini strutturalmente diversi, non essendo necessario definire la lunghezza dell'*array* che è ricevuto; la testa del metodo infatti è così strutturata:

```
public tipo_di_restituzione nome_metodo( double[])
```

senza l'indicazione tra parentesi quadre della dimensione dell'*array*.

Potranno così coesistere agenti che inviano ordini, come in Sum, tutti di quantità pari ad uno passando due sole variabili oppure agenti che passano ordini più strutturati con l'indicazione della quantità o del tipo di ordine immesso (inviando quindi più di due variabili).

Inizialmente, pur apportando questa modifica strutturale, mi sono rifatto fedelmente al codice di Sum creando un piccolo programmino in java, in modo tale da avere un modo per verificare l'esattezza del comportamento del programma, passando solo in un secondo momento alla simulazione vera e propria utilizzando il protocollo di *Swarm*.

5.3.1 IL FILE BOOK.JAVA.

La prima versione del *file* Book.java è così strutturata:

```
// Book.java utilizzabile con Generatore.java
import java.util.*;

public class Book {
    public int printing, orderNumber, count=0, agentNumber,
    maxOrderQuantity;
    public double executedPrice=1, meanPrice=1,
                previousClosingPrice=executedPrice,
    currentMeanPrice=0;
```

La prima riga riporta un commento, identificato dalle due *slash*, in cui è contenuto semplicemente il nome del *file* che si sta leggendo. Vi è poi la definizione di alcune variabili *public* che saranno utilizzate all'interno dei metodi della classe *Book*.

La prima di queste variabili, *printing*, è di tipo intero ed è utilizzata per attivare alcune righe di codice che altro non fanno se non alcune stampe a video in modo da poter verificare che non capitino cose bizzarre dovute ad errori di codificazione. Come accadeva in *Sum*, questa variabile sarà modificabile attraverso il *Model*.

La variabile *orderNumber* è semplicemente un contatore di ordini ricevuti dal *book*; allo stesso modo *count* è utilizzata per sapere il numero dei contratti conclusi.

agentNumber e *maxOrderQuantity* rispettivamente indicano il numero degli agenti che operano nella simulazione e la quantità massima di azioni che possono proporre per ogni ordine.

correntMeanPrice è una sommatoria dei prezzi dei contratti conclusi, in modo tale da avere, a fine giornata, un dato che ci permetta facilmente di calcolare il prezzo medio (assegnato poi alla variabile *meanPrice*). *executedPrice* è utilizzata per il prezzo corrente e *previousClosingPrice* indica il prezzo di chiusura del giorno precedente, rappresentato per ora dall'importo dell'ultimo contratto della giornata, non essendo ancora inserito il meccanismo di asta di chiusura.

```
/* the book works on the basis of two arrayList containing
sell order in
    increasing order or buy order in decreasing order. The
orders are
    arrays which have in the first position the price, in the
second
    the number of agent who places the order and in the third
the quantity.
    If an order obtains an immediate matching, it is not
filed */

public ArrayList buyOrderStorehouse=new ArrayList();
public ArrayList sellOrderStorehouse=new ArrayList();

// to extract data from the orders received from agents
public double extract( ArrayList nameList, int indexOrder,
int positionDatum ) {
    double a = ((double [])
nameList.get(indexOrder))[positionDatum];
    return(a);
}
```

Tutto ciò che è scritto tra i simboli */** e **/* è considerato commento anche se è posto su più di una riga. Subito dopo l'annotazione vi è la dichiarazione di due variabili di tipo *ArrayList* alle quali viene subito assegnato dello spazio in memoria utilizzando il costruttore. *buyOrderStorehouse* e *sellOrderStorehouse* sono i due contenitori, rispettivamente d'ordine d'acquisto e vendita.

Definisco poi il metodo *extract* per avere un modo agevole di ottenere i valori contenuti negli *array* che a loro volta si trova nell'*arrayList*. Quando si attiva questo metodo è necessario inserire tre dati *nameList* (nome del contenitore),

indexOrder (il numero di riga in cui l'ordine si trova, ricordando che le liste iniziano dalla riga 0) e *positionDatum* (indice di posizione del dato all'interno dell'*array*).

Prima del nome del metodo è necessario indicare il tipo di dato che viene restituito: in questo caso è necessario mettere *double* anche se alcuni dati, come ad esempio il numero identificativo dell'agente, sono semplicemente degli interi. Questo è dovuto al fatto che gli *array* non possono contenere oggetti non omogenei, per cui è necessario indicare cosa contengono, e una volta stabilito questo, non è possibile inserire dati di altro tipo (l'unico modo è effettuare un *casting*, quando consentito, facendo in modo che un genere di dato venga considerato come un altro). Essendo lo scopo principale di questa classe l'accoppiamento di proposte per la definizione dei prezzi di mercato ho preferito utilizzare il formato di quest'ultimi per la definizione degli *array*, utilizzando così i *double*.

Il corpo del metodo è costituito da due sole righe di codice, la prima delle quali è molto strutturata ed è più comprensibile se considerata una parte per volta. Con il metodo *get()* degli *arrayList* viene fornito l'oggetto in una certa posizione che può essere indicata tra parentesi. In questo caso è individuato l'elemento in posizione *indexOrder* nella lista *nameList*, ma, essendo un *array*, quello che si ottiene è un indirizzo di memoria. Una volta informato il compilatore che si punta ad un *array* con un *casting* (pre-ponendo "(double [])" alla riga di codice) è possibile utilizzare il modo di accesso rapido agli elementi, tipico di questi oggetti, indicando la posizione del dato tra parentesi quadre.

La seconda riga restituisce il dato cercato.

```
// to fill sellOrderStorehouse in increasing order
public int fillStorehouseIncreasingP(double newPrice){
    int index=0;
    for(int i=0;i<sellOrderStorehouse.size();i++){
        if(newPrice<extract(sellOrderStorehouse,i,0)){
            index=i;
            break;
        }
        else
            continue;
    }
    return(index);
}

//to fill buyOrderStorehouse in descreasing order
public int fillStorehouseDecreasingP (double newPrice){
    int index=0;
    for(int i=0;i<buyOrderStorehouse.size();i++){
        if(newPrice>extract(buyOrderStorehouse,i,0)){
            index=i;
            break;
        }
        else
```

```
        continue;
    }
    return(index);
}
```

L'esigenza dei due metodi sopra riportati, è nata per poter usufruire del metodo *add()* che consente di inserire gli oggetti all'interno degli *arrayList* in una determinata posizione aggiornando automaticamente gli indici degli oggetti già presenti; se al metodo *add()* non si indica la posizione, l'oggetto viene posizionato al fondo della lista. Sulla base di ciò, ho pensato che il riordino delle liste potesse essere di gran lunga semplificato, facendo in modo che gli ordini vengano inseriti al posto giusto non appena arrivano al *book* (ovviamente se non trovano immediata controparte).

Il prezzo proposto dall'agente è passato a uno dei due metodi in questione ed essi forniscono l'indice dove deve essere inserita la proposta; per ottenere questo risultato ho costruito un ciclo *for*, su tutta la lunghezza della lista presa in considerazione (il metodo *size()* indica la dimensione dell'*arrayList*), interrompendolo nel caso in cui non sia necessario arrivare in fondo riducendo così il tempo di esecuzione.

Il metodo *fillStorehouseIncreasingP* controlla di inserire la proposta subito prima dell'ordine con prezzo superiore, viceversa *fillStorehouseDecreasingP* di quello con prezzo inferiore, ottenendo così nel caso di prezzi uguali la precedenza agli ordini inseriti antecedentemente.

Una volta individuato l'indice è possibile inserire l'ordine con il metodo *add()*.

Andando avanti nel *file* si giunge ora al metodo attivato dagli agenti durante la fase di contrattazione continua.

```
// receiving an order when the market is open
public void setOrderFromAgent(double[] order) {
    if(printing==1)
        System.out.println("The book received an order from agent
" +
        order[1] + " at price " + order[0] + " for " + order[2]+
" shares");
    orderNumber++;
}
```

Come si nota *setOrderFromAgent* è di tipo *void* in quanto non restituisce nulla quando viene attivato. Come anticipato al metodo è passato un *array* che rappresenta l'ordine piazzato dall'agente che in quel momento ha richiamato il metodo. Le prime righe del corpo sono considerate solo se l'opzione di stampa *printing* è uguale a 1 e fanno scrivere a video dal programma il numero identificativo dell'agente che invia la proposta, il prezzo a cui acquista o vende e la quantità di azioni a cui si riferisce. La riga di codice successiva è l'incremento del contatore degli ordini ricevuti dal *book*, variabile puramente di controllo, considerato che per ora il programma non gira con agenti ma

semplicemente con un *file*¹⁶ che genera casualmente ordini: viene utilizzata per controllare che effettivamente tutti gli ordini vengano ricevuti.

A questo punto del *file* come in Sum vengono previsti 3 casi:

```
/* if price==0 no action required, but sending a message to
the agent
with price=0*/
if(order[0]==0);
```

Nel primo non capita ancora nulla, in quanto non avendo ancora interazione con gli agenti non è inviato alcun messaggio. È questo il caso in cui l'ordine ricevuto abbia prezzo uguale a 0: ovviamente viene considerato come se l'ordine non sia arrivato essendo impossibile nella realtà il verificarsi di questa situazione; una volta inserito il *file* nella simulazione vi sarà l'invio di un messaggio all'agente, con prezzo uguale a 0 per informarlo che è come se non avesse piazzato l'ordine (vedi codice di Sum).

```
// the agent is selling at minimum price if price<0
else if(order[0]<0) {
/* if there is a compatible order, this one is removed
and the order
received it's not inserted in the book. It's not
necessary to shift
rows because with remove() the arrayList updates the
index */
if(buyOrderStorehouse.size())>0 &&
extract(buyOrderStorehouse, 0, 0) >=
-order[0]) {
count++;
executedPrice=extract(buyOrderStorehouse, 0, 0);
currentMeanPrice+=executedPrice;
buyOrderStorehouse.remove(0);
if(printing==1) {
for(int i =0; i<buyOrderStorehouse.size(); i++){
for(int y=0; y<order.length; y++){
System.out.print(extract(buyOrderStorehouse, i,
y) +" ");
}
System.out.println();
};
System.out.println();
}
}

else {
order[0]= -order[0];
/* to insert the order in the book in increasing order. With
add( )the
arrayList updates the index */
```

¹⁶ In appendice A riporto il codice del programma in Java "Generatore".

```

if(sellOrderStorehouse.size()==0||order[0]>=extract(sellOrderStorehouse, (sellOrderStorehouse.size()-1),0))
    sellOrderStorehouse.add(order);
    else

sellOrderStorehouse.add(fillStorehouseIncreasingP(order[0]),order);

    if(printing==1) {
        for(int i =0; i<sellOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(sellOrderStorehouse, i,
y) +" ");
            }
            System.out.println();
        };
        System.out.println();
    }
}
}

```

Il codice in questo caso (prezzo<0 cioè ordine in vendita) è una traduzione da Objective C a Java senza quasi nessun tipo di cambiamento. Ovviamente, come detto prima, ho considerato che ora si utilizzano *arrays* e richiamo il mio nuovo metodo per inserire le proposte in modo crescente: ovviamente se la lista fosse vuota o se l'ordine dovesse essere messo in fondo (perché con prezzo superiore a tutti gli altri), il metodo *fillStorehouseIncreasingP* non è attivato in quanto è sufficiente che venga inserito col metodo *add()* senza indicare l'indice, rendendo così il processo più veloce.

```

// the agent is buying at the maximun price if price>0
    else {
/* if there is a compatible order, this one is removed and the
order received
    it's inserted in the book. It's not necessary shift rows
because with
    remove() the arrayList updates the index */
        if(sellOrderStorehouse.size()>0                                &&
extract(sellOrderStorehouse, 0, 0)
        <= order[0]) {
            count++;
            executedPrice=extract(sellOrderStorehouse, 0, 0);
            currentMeanPrice+=executedPrice;
            sellOrderStorehouse.remove(0);
            if(printing==1) {
                for(int i=0; i<sellOrderStorehouse.size(); i++){
                    for(int y=0; y<order.length; y++){
                        System.out.print(extract(sellOrderStorehouse, i,
y) +" ");
                    }
                    System.out.println();
                };
                System.out.println();
            }
        }
    }
}

```

```

        with price=0 */
        if(order[0]==0);

    // the agent is selling at minimum price if price<0
    else if(order[0]<0) {
        order[0]=-order[0];
        /* to insert the order in the book in increasing order.
With add( )the
        arrayList updates the index */
        if(sellOrderStorehouse.size()==0 ||
order[0]>=extract(sellOrderStorehouse,
(sellOrderStorehouse.size()-1), 0))
            sellOrderStorehouse.add(order);
        else

sellOrderStorehouse.add(fillStorehouseIncreasingP(order[0]),
order);
        if(printing==1) {
            for(int i =0; i<sellOrderStorehouse.size(); i++){
                for(int y=0; y<order.length; y++){
                    System.out.print(extract(sellOrderStorehouse, i,
y) + " ");
                }
                System.out.println();
            };
            System.out.println();
        }
    }

    // the agent is buying at the maximun price if price>0
    else {
        /* to insert the order in the book in decreasing
order.With add()the
        arrayList updates the index */
        if(buyOrderStorehouse.size()==0 ||
order[0]<=extract(buyOrderStorehouse,
(buyOrderStorehouse.size()-1) ,0))
            buyOrderStorehouse.add(order);
        else

buyOrderStorehouse.add(fillStorehouseDecreasingP(order[0]),orde
r);

        if(printing==1) {
            for(int i=0; i<buyOrderStorehouse.size(); i++){
                for(int y=0; y<order.length; y++){
                    System.out.print(extract(buyOrderStorehouse, i, y)
+ " ");
                }
                System.out.println();
            };
            System.out.println();
        }
    }
}
}

```

Il codice di questo metodo chiamato *setOrderBeforeOpeningFromAgent* non presenta alcuna differenza rispetto ad una parte di *setOrderFromAgent*, manca solo la parte in cui si controlla se vi è un ordine opposto compatibile (non dovendo in questa fase accoppiare le proposte).

Due metodi molto importanti per la simulazione sono quelli che sintetizzano le operazioni da effettuare all'inizio ed alla fine di ogni giornata:

```
// at the end of each day
public void setMeanPrice(){
    if(count>0)
        meanPrice=currentMeanPrice/count;
    // otherwise we keep previous value
}

// at the beginning of each day
public void setClean(){
    sellOrderStorehouse.clear();
    buyOrderStorehouse.clear();
    previousClosingPrice=executedPrice; // the last of
"yesterday"
    currentMeanPrice=0;
    count=0;
}
```

setMeanPrice si limita a calcolare il prezzo medio della giornata, utilizzando la sommatoria dei prezzi (*currentMeanPrice*) e il numero dei contratti conclusi durante la giornata (*count*). Il metodo *setClean*, invece, è attivato all'inizio di ogni giornata e serve per preparare il *book* ad un nuovo ciclo di contrattazione: si cancellano gli ordini che non hanno trovato contropartita nella giornata precedente e, non essendoci asta di chiusura, si assegna alla variabile del prezzo di chiusura (*previousClosingPrice*) il valore dell'ultimo contratto e si azzerano sommatoria dei prezzi e contatore dei contratti conclusi.

Gli ultimi metodi inseriti servono per lo più quando il *file* *Book.java* verrà modificato e inserito nel programma di simulazione che utilizza il protocollo di *Swarm*.

```
// to set the number of agents
public void setAgentNumber(int n){
    agentNumber=n;
}

// to set the number of agents
public void setMaxOrderQuantity(int m){
    maxOrderQuantity=m;
}

// to set the value of option printing
public void setPrinting(int p){
    printing=p;
}
```

```
}
```

Questi primi tre serviranno per assegnare i valori che l'utente del programma JavaSum inserirà attraverso l'interfaccia del *Model*.

```
// to get the last executed price
public double getPrice(){
    return(executedPrice);
}

// to get the mean price of yesterday
public double getMeanPrice(){
    return(meanPrice);
}

// to get the number of sell orders fill in the book
public int getSellOrderNumber(){
    return(sellOrderStorehouse.size());
}

// to get the number of buy orders fill in the book
public int getBuyOrderNumber(){
    return(buyOrderStorehouse.size());
}

// to get the closing price of yesterday
public double getPreviousClosingPrice(){
    return(previousClosingPrice);
}
```

Quest'ultimi, restituiscono il valore di determinate variabili utilizzate sia dagli agenti che dall'*Observer* nelle loro elaborazioni.

Come già ricordato il *file* finora osservato è semplicemente un programma in java attivato da un generatore di ordini casuali il cui codice è riportato in appendice A (Generatore.java). Il codice così prodotto, ha ora raggiunto un punto importante in quanto possiede tutte le componenti per poter interagire con gli agenti *random*, ovverosia agenti che operano in modo casuale nella simulazione (Agagliate, 2004). In estrema sintesi il punto raggiunto in questo momento è un *book* capace di gestire ordini composti solamente dal numero dell'agente e dal prezzo a cui vuole comprare o vendere le azioni (considerando quindi tutti gli ordini di quantità unitaria come accadeva in Sum). Prima di inserire nel codice le righe che consentono di lavorare con *Swarm*, faccio in modo che il *book*, una volta ricevuto l'ordine da un agente, lo trasformi in un *array* lungo quattro, in modo da renderlo standard indipendentemente dalla reale lunghezza inviata dall'agente.

```
// to standardize the order
public double[] standardizedOrder(double[] orderFromAgent){
    double [] order=new double [4];
    if(orderFromAgent.length==2){
```

```

        double[] orderStandard={orderFromAgent[0],
orderFromAgent[1], 1, 0};
        order=orderStandard;
    }
    if(orderFromAgent.length==3){
        double [] orderStandard={orderFromAgent[0],
orderFromAgent[1],
                                orderFromAgent[2], 0};
        order=orderStandard;
    }
    if(orderFromAgent.length==4){
        double [] orderStandard={orderFromAgent[0],
orderFromAgent[1],
                                orderFromAgent[2],
orderFromAgent[3]};
        order=orderStandard;
    }
    return(order);
}

```

Questa operazione consente di rendere il *book* molto flessibile, essendo in grado di ricevere ordini anche strutturalmente diversi tra di loro. Se i dati inviati sono solo due, come nel caso di agenti *random* (creati da Agagliate), la quantità è posta uguale ad 1 e l'ultimo parametro, che per ora non ha ancora un preciso scopo uguale a zero (potrebbe essere utilizzato per ordini al meglio o con limite di tempo di permanenza nel *book* oppure per ordini di tipo "esegui e cancella", "esegui comunque", ecc). Se invece la proposta ricevuta ha lunghezza pari a tre, sarà solo l'ultimo parametro ad essere preso in considerazione, mentre nel caso l'ordine sia lungo quattro non sarà necessario effettuare nessun tipo di operazione. Il metodo che effettua questa operazione è attivato non appena l'ordine arriva al *book*, in modo tale che si lavori sempre con un ordine che potremmo definire "*standard*", semplificando così il codice. In conclusione l'ordine elaborato è così composto:

Prezzo, Numero dell'agente che propone, Quantità, Parametro

L'ultimo passaggio, per completare la prima fase di realizzazione, consiste nel creare un metodo che consenta di informare gli agenti dell'esito degli ordini in modo tale che possano tenere una propria contabilità. Come accadeva in Sum-0.66, gli agenti sono sempre informati, anche nel caso in cui non sia stipulato il contratto: in questo caso riceveranno un messaggio con prezzo uguale a zero. Il metodo che consente di effettuare queste operazioni è così strutturato:

```

// to inform the agent of executed price
private void message(double n, double p){
    BasicSumAgent anAgent;
    indexAgentListIndex.setOffset((int)n-1);
    anAgent = (BasicSumAgent)indexAgentListIndex.get();
    anAgent.setConfirmationOfExecutedPrice(p);
}

```

Durante la simulazione la lista degli agenti è continuamente rimescolata per far sì che non vengano chiamati ad operare sempre nella stessa sequenza. Per poter mandare il messaggio all'agente destinatario è necessario conoscere il suo indirizzo di memoria; è stato quindi necessario creare l'indice della lista in modo tale da poter puntare all'elemento attraverso questo, acquisendo così l'indirizzo dell'agente desiderato.

Il metodo sopra riportato, utilizza l'iteratore della lista, cioè un metodo che consente di passare in modo sequenziale da un elemento all'altro della lista presa in considerazione: essendo possibile posizionarlo su di un elemento specifico, attraverso il metodo *setOffset* lo poniamo in corrispondenza del numero dell'agente che si vuole informare, potendo così, col metodo *get*, ottenere l'indirizzo di memoria. Il metodo *setConfirmationOfExecutedPrice* che viene attivato dal *book* è quello che consente all'agente di tenere la propria contabilità (numero azioni possedute, moneta a disposizione o situazione debitoria, ricchezza raggiunta, ecc).

A questo punto ho raggiunto un primo grande traguardo: disporre di un *file* facilmente integrabile nel codice della nostra simulazione di borsa capace di interagire con gli agenti che operano nel mercato.

L'ultimo passo ancora da effettuare è la modifica del *file* in modo da poter utilizzare *Swarm*.

È necessario per prima cosa importare le librerie:

```
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import swarm.collections.*;
```

Le prime due sono essenziali per creare gli oggetti della simulazione ed assegnare ad ognuno di una propria zona di memoria (*aZone*) (Staelin, 2000). L'ultima invece è necessaria per poter utilizzare le liste della libreria di *Swarm* (utile per poter inviare i messaggi e informare gli agenti, avendo utilizzato l'iteratore della lista e i suoi metodi per conoscere gli indirizzi di memoria).

Bisogna poi rendere la classe *Book* una sottoclasse di *SwarmObjectImpl* in modo tale da poter assegnargli fin dall'inizio una propria zona di memoria in cui vengono caricati tutti i metodi e le variabili che utilizzerà (altrimenti il *Model* e l'*Observer* non potrebbero gestire e osservare la simulazione).

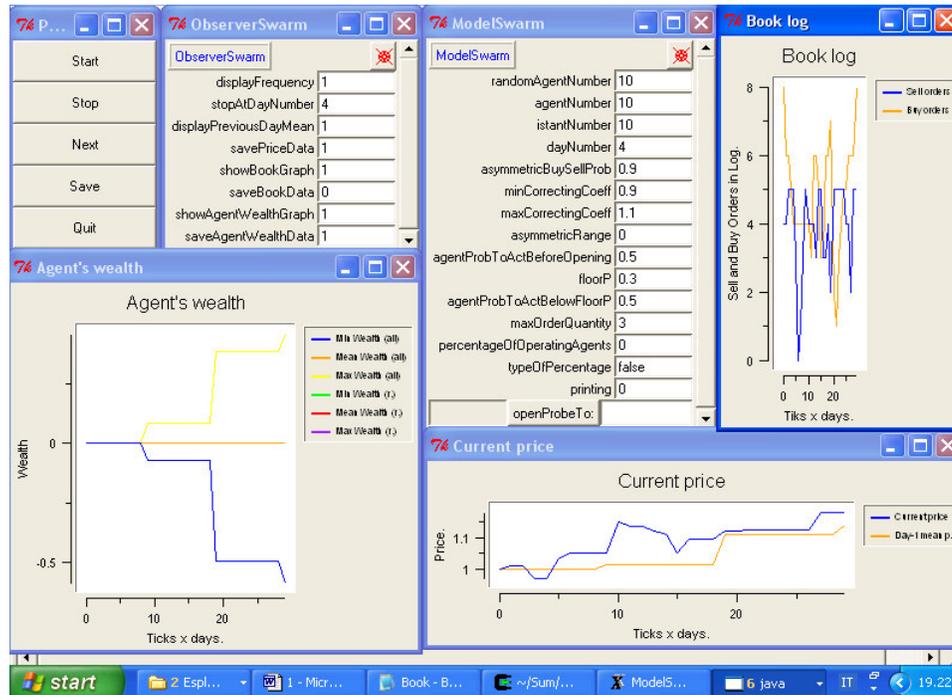
Da ultimo si chiama il costruttore essenziale per la creazione dell'oggetto *book*:

```
// Constructor
public Book(Zone aZone){
    super(aZone);
}
```

ad esso viene passata la zona di memoria da assegnare al *book* e nel suo corpo viene chiamato il costruttore della propria classe genitore (*SwarmObjectImpl*). Sostituendo il *file* del *book* così ottenuto si passa alla versione JavaSum-0.3 che presenta un ambiente simulato in cui operano solo agenti *random* e in cui non

si hanno grandi innovazioni rispetto alla versione Sum-0.66 scritta in Objective C.

Avviando la simulazione si ottiene¹⁷:



5.3.2 LA GESTIONE DELLE QUANTITA'.

Il *book* messo a disposizione del nostro ambiente simulato, in questo momento, gestisce solo ordini di quantità unitaria, consentendo una più facile gestione delle proposte parziali.

La decisione di utilizzare gli *arrayList* al posto di matrici, che in questo contesto avrebbero avuto il grosso limite di dover essere definite nelle dimensioni senza sapere a priori il numero di ordini che il *book* avrebbe ricevuto, e il concepimento di ordini sintetizzati da *arrays*, hanno aperto la strada verso un maggior realismo della simulazione.

La struttura dell'ordine inviato consente di inserire facilmente la gestione delle quantità e soprattutto, attraverso l'ultimo parametro, l'identificazione del "tipo" di ordine inserito dall'agente, consentendo così di utilizzare ordini diversi da quelli con limite di prezzo (unico tipo previsto da Sum-0.66).

¹⁷ Per una spiegazione completa di ciò che accade, e di quello che viene generato si veda il capitolo 4, in cui si parla in modo dettagliato di ciò che il modello mette a disposizione dell'utente.

Gli `arrayList` sono più lenti in lettura di circa la metà, rispetto all'utilizzo di matrici, non avendo un modo, a differenza degli `arrays`, di accesso rapido agli elementi. Potendo però gestire le quantità con semplicità, si possono eliminare un gran numero di interazioni che il programma effettua, quando riceve le proposte o invia i messaggi agli agenti, recuperando alcuni secondi che potrebbero essere preziosi quando si utilizza la simulazione.

Sulla base di queste considerazioni, le modifiche da apportare non incontrano grosse difficoltà, considerando che la struttura è stata fin dall'inizio impostata per questo passaggio.

L'ordine *standard* che il *book* riceve, contiene al suo interno, in terza posizione (la seconda, considerando che la numerazione inizia dallo 0), l'elemento che indica la quantità per la quale l'ordine è piazzato. Sapendo questo è possibile inserire nel codice del metodo utilizzato in contrattazione continua (*setOrderFromAgent*), dei controlli su questo parametro, in modo da gestire la quantità. Le situazioni che si possono verificare sono:

1. Se l'ordine che arriva non ha controparte immediata non capita nulla rispetto a prima e l'ordine viene inserito nel lato del *book* a cui è destinato (acquisti o vendite).
2. Se ha controparte, si inserisce nel codice l'istruzione *while* che consente di effettuare il gruppo di istruzioni che lo seguono finché non si verifica una condizione prestabilita, indicata tra parentesi tonde, secondo questa struttura:

```
while(condizione) {  
    istruzione;  
    istruzione;  
    .....}
```

La condizione che deve essere soddisfatta è che l'ordine appena ricevuto abbia quantità superiore a 0, che ci sia un ordine compatibile con il prezzo proposto e che la lista in cui si cerca la controparte non sia vuota. Cos' facendo si gestisce l'ordine in modo adeguato, uscendo dal ciclo *while* nel caso in cui sia finita la gestione della quantità.

Le istruzioni da eseguire sono differenti a seconda che la quantità sia uguale, inferiore o superiore del primo ordine in lista. Nel primo caso la quantità è esaurita subito quindi è sufficiente informare gli agenti interessati, aggiornare le variabili, azzerare la quantità per uscire dall'istruzione *while* e rimuovere l'ordine in contropartita dalla lista.

Se la quantità dell'ordine appena ricevuto è minore di quella proposta dalla controparte, le istruzioni sono le medesime, l'unica differenza è relativa a quella che elimina l'ordine dalla lista in quanto, in questo caso, deve correggere la proposta per la quantità di azioni oggetto del contratto appena concluso.

Nell'ultimo caso, cioè quantità superiore di quella della proposta in attesa, si corregge l'ordine arrivato e si eliminano man mano gli ordini compatibili

finché non si verifica una delle situazioni della condizione dell'istruzione *while*¹⁸.

Con questa modifica sono necessarie alcune variazioni nel codice dell'*Observer*, in quanto, a differenza di quanto accadeva prima, le variabili *sellOrderNumber* e *buyOrderNumber* non indicano più il quantitativo di azioni in attesa di controproposta, e quindi il grafico che utilizzava queste perde significato (per rappresentare tale quantitativo in attesa, non è più sufficiente osservare la lunghezza della lista, perché ora gli ordini inseriti possono riferirsi a più di una azione). Ho così inserito altre due variabili (*sharesInSellSide* e *sharesInBuySide*), nel codice del *book*, che contano il numero effettivo di azioni in attesa ed ho poi modificato il grafico *book log* affinché si riferisca ad esse e non più al numero di ordini.

Le modifiche nell'*Observer* devono essere apportate sia per la costruzione del grafico sia per il salvataggio dei dati su file¹⁹.

Il passo successivo della programmazione, prevede l'inserimento di alcuni metodi che permettano lo studio della dinamica di formazione dei prezzi, con lo scopo principale di studiare il formarsi delle bolle e dei crash di mercato (Chinaglia, 2004)²⁰.

Prima di effettuare queste innovazioni, come anticipato precedentemente, ho provato ad inserire gli *arrayList* anche nella contabilità degli agenti, in modo tale da non dover utilizzare matrici sovradimensionate. Ho poi confrontato entrambe i modi di comunicare con gli agenti (inviare n messaggi di quantità unitaria o 1 messaggio che si riferisce a n azioni), per verificare che l'utilizzo delle liste, da parte di ogni operatore, non rallentasse di molto la simulazione sulla base delle considerazioni effettuate in precedenza. Dai risultati, ho potuto constatare che effettivamente si ha un rallentamento, che per ora risulta essere accettabile (dell'ordine di una decina di secondi utilizzando 300 agenti, simulando 100 giorni, tenendo presente che agli operatori è consentito emettere più ordini nella stessa giornata).

Al fine di inviare un unico messaggio, per ogni contratto concluso, ho modificato il metodo *act2*, nel file *BasicSumAgent.java*, riguardante il modo con cui gli agenti effettuano la contabilità, sia per l'inserimento delle liste, sia per la gestione delle quantità (si veda l'appendice B). Piccole modifiche sono state apportate anche a *Model* e *RandomAgent*, dovute al fatto che, non essendo più necessario dimensionare le matrici, alcune variabili possono essere eliminate (*istantNumber* e *maxOrderQuantity*).

¹⁸ In appendice B riporto il codice della classe *Book* definitivo prima dell'inserimento delle aste.

¹⁹ Nel paragrafo successivo si parla in modo approfondito dei dati che l'*observer* fornisce e salva su file.

²⁰ Vedi nota precedente

5.4 INSERIMENTO DI METODI E GRAFICI PER LO STUDIO DELLA DINAMICA DI FORMAZIONE DEI PREZZI.

Come già affermato nei capitoli precedenti, le simulazioni ad agenti sono utilizzate per lo studio di fenomeni complessi. Ovviamente, per poter osservare ciò che i modelli producono in termini di risultati, è necessario introdurre nel programma un meccanismo di registrazione dei dati prodotti dalla simulazione. Riprendendo ciò che Sum mette a disposizione (Chinaglia, 2004) ho introdotto nel codice del *book* alcuni metodi, attivati dall'*Observer*, che forniscono dati importanti per lo studio della dinamica dei prezzi.

Le modifiche apportate al codice dell'*Observer*²¹, oltre ad essere concentrate su questi obiettivi da raggiungere, è stato strutturato in modo che sia possibile all'inizio della simulazione decidere cosa disegnare e cosa salvare su file attraverso le *probe* dell'*Observer*. La flessibilità è molto importante in quanto a volte potrebbe non essere necessario utilizzare tutte le potenzialità del modello, creando così, a seconda dei casi, un'ottimizzazione dei tempi di esecuzione della simulazione. Vediamo ora in dettaglio le modifiche apportate al *book*.

5.4.1 QUANTITA' DI AZIONI NEL *BOOK* AL TERMINE DELLA FASE DI APERTURA.

Nel modello Sum gli agenti inviano *n* ordini di quantità unitaria per acquistare o vendere *n* azioni. Quindi per conoscere il numero delle azioni presenti nel *book* dopo la conclusione della fase di pre-apertura (in cui non vi è ancora asta e viene solo utilizzata per non iniziare giornate con *book* vuoto) è sufficiente contare gli ordini che sono proposti sia nel lato degli acquisti che nel lato delle vendite.

Le variabili interessate sono *sellOrderNumberBeforeOpening* e *buyOrderNumberBeforeOpening*, entrambe costantemente aggiornate non appena il *book* riceve un nuovo ordine. Attraverso i rispettivi metodi *getSellOrderNumberBeforeOpening* e *getBuyOrderNumberBeforeOpening* attivati dall'*Observer* è poi possibile salvare su *file* o rappresentare graficamente i dati alla fine della fase di pre-apertura.

Per ricreare lo stessa informazione all'interno del nuovo modello JavaSum è stato prima di tutto necessario inserire due nuove variabili:

```
/** The number of shares in buy side of book in opening or
closing Auctions. */
public int sharesInBuySideInOpeningOrClosingAuctions;
```

²¹ In appendice C è riportato parte del codice del modello (classi principali senza agenti).

```
/** The number of shares in sell side of book in opening or
closing Auctions. */
public int sharesInSellSideInOpeningOrClosingAuctions;
```

Nell'appendice B si può notare che queste due variabili sono aggiornate non appena un ordine viene inserito da un'agente; così facendo alla fine del primo *tick* di simulazione, periodo nel quale si svolge la pre-apertura (nel modello più evoluto si ha l'asta d'apertura), si può sapere esattamente il quantitativo di azioni presenti nei due lati di acquisto e di vendita.

Ovviamente l'*Observer* attiva i due metodi

```
/** To get the number of shares in sell side in opening or
closing Auctions. */
public int getSharesInSellSideInOpeningOrClosingAuctions(){
    return(sharesInSellSideInOpeningOrClosingAuctions);
}

/** To get the number of shares in buy side in opening or
closing Auctions. */
public int getSharesInBuySideInOpeningOrClosingAuctions(){
    return(sharesInBuySideInOpeningOrClosingAuctions);
}
```

che, come si può notare, restituiscono il valore delle due variabili in questione. L'informazione che si ottiene è molto importante in quanto la composizione del *book* prima dell'apertura, e la sua successiva dinamica, influenza sicuramente l'andamento dei prezzi.

5.4.2 DIFFERENZA TRA LE QUANTITÀ DI AZIONI IN ACQUISTO E IN VENDITA.

Il secondo dato molto importante al fine dello studio della dinamica di formazione dei prezzi è senza dubbio lo sbilancio delle quantità nei due lati del *book*. Secondo le leggi di mercato l'eccesso di domanda crea un aumento dei prezzi così come a sua volta un eccesso di offerta porta ad una diminuzione degli stessi. I meccanismi di formazione dei prezzi e dell'inserimento delle proposte nei due lati del *book*, nel mercato reale e di conseguenza nel nostro modello, portano ovviamente a questo tipo di situazioni. Si pensi per esempio ad un periodo in cui si concentrino proposte in acquisto: maggiore è il numero degli ordini in acquisto, più è probabile che quelli in attesa nel lato delle vendite vengano eseguiti. Data la priorità dei prezzi minori i successivi saranno a prezzo superiore e chi vuole essere sicuro di acquistare dovrà proporre prezzi superiori alimentando la tendenza al rialzo.

Il modello per conoscere in ogni istante l'eventuale squilibrio tra le quantità presenti nel *book* utilizza due variabili simili a quelle descritte precedentemente, con l'unica differenza di essere aggiornate non solo nel metodo *setOrderInOpeningOrClosingAuctionsFromAgent* ma anche in *setOrderFromAgent*. Le variabili in questione sono *sharesInBuySide* e

sharesInSellSide. Il metodo attivato dall'*Observer* per salvare e rappresentare i dati nel grafico è così strutturato:

```
/** To get the spread between quantities in sell and buy
side. */
public int getBuySellQuantitySpread(){
    return(sharesInBuySide-sharesInSellSide);
}
```

il quale restituisce solamente la differenza tra le quantità presenti nei due lati del *book*.

5.4.3 DIFFERENZA TRA IL PREZZO MIGLIORE ED IL PEGGIORE DELLE LISTE DI ORDINI.

Il terzo dato importante che verrà registrato all'interno di un *file*, e rappresentato graficamente, è la differenza tra il miglior prezzo ed il peggiore delle due liste in attesa di controproposta.

I risultati che si ottengono danno un'indicazione di quanto gli ordini permarranno del *book*. Ovviamente più la distanza è elevata e maggiore sarà la probabilità che alcuni ordini resteranno a lungo nelle liste o addirittura saranno ineseguiti a fine giornata.

Al fine di ottenere l'informazione appena descritta sono stati inseriti due metodi all'interno del codice che consentiranno all'*Observer* di ottenere i dati da salvare e da rappresentare:

```
/** To get the spread between the first and the last price in
the sell side. */
public double getSellFirstLastSpread(){
    double spread=0;
    if(sellOrderStorehouse.size()==0)
        spread=0;
    else
        spread=extract(sellOrderStorehouse, 0, 0)-
            extract(sellOrderStorehouse,
sellOrderStorehouse.size()-1, 0);
    return(spread);
}
```

```
/** To get the spread between the first and the last price in
the buy side. */
public double getBuyFirstLastSpread(){
    double spread=0;
    if(buyOrderStorehouse.size()==0)
        spread=0;
    else
        spread=extract(buyOrderStorehouse, 0, 0)-
            extract(buyOrderStorehouse, buyOrderStorehouse.size()-
1, 0);
    return(spread);
}
```

La struttura è la stessa per entrambe con l'unica ovvia differenza degli *arrayList* a cui fanno riferimento.

Come è possibile osservare dal codice, nel caso in cui non vi siano ordini all'interno delle liste, il valore è posto uguale a zero, mentre nel caso contrario viene effettuata la differenza tra il prezzo in prima posizione e quello in ultima, servendosi del metodo *extract* precedentemente descritto.

5.4.4 DIFFERENZA TRA MIGLIOR PREZZO LETTERA E MIGLIOR DENATO.

```
/** To get the spread between first bid and first ask price.
 */
public double getSpread(){
    double spread=0;
    if(sellOrderStorehouse.size()==0 &&
buyOrderStorehouse.size(>0)
        spread=0-extract(buyOrderStorehouse, 0, 0);
    else if(sellOrderStorehouse.size()==0 &&
buyOrderStorehouse.size()==0)
        spread=0;
    else if(sellOrderStorehouse.size(>0 &&
buyOrderStorehouse.size()==0)
        spread=extract(sellOrderStorehouse, 0, 0)-0;
    else
        spread=extract(sellOrderStorehouse, 0, 0)-
extract(buyOrderStorehouse, 0, 0);
    return(spread);
}
```

Il metodo *getSpread* fornisce la differenza tra i due migliori prezzi presenti nei due lati del *book*.

Lavorando con *arrayList* è necessario prendere in considerazione tutti i possibili casi estremi che si possono verificare, in modo tale da non far interrompere il programma di simulazione con un errore: questo accadrebbe nel caso in cui si interrogasse una posizione vuota o addirittura inesistente di un *arrayList*. Come per i precedenti metodi, il nome del metodo inizia per “*get*” in quanto il metodo sarà attivato dall’*Observer* per salvare i dati e creare il grafico ad essi relativo.

5.5 INSERIMENTO DELLE ASTE

Gran parte del lavoro di programmazione è stato dedicato all’inserimento delle aste. Ogni giornata di trattazione, nel modello Sum-0.66, è costituita da due fasi:

- La fase prima dell’apertura, in cui gli ordini sono solo inseriti allo scopo di non iniziare una nuova giornata con *book* vuoto, durante la

quale non si accoppiano le proposte compatibili. Alla fine di questa fase si hanno quindi due liste di ordini che si incrociano, ovvero con prezzi che potrebbero essere formati (si ha la presenza, ad esempio, di chi vuole vendere minimo a 2 e contemporaneamente di chi vuole acquistare al massimo a tre). Nella realtà questo non accade mai, la fase di contrattazione continua inizia sempre con il miglior prezzo in vendita maggiore del miglior prezzo in acquisto. Il motivo per cui non accade è il meccanismo dell'asta che Borsa Italiana s.p.a. ha delineato nel suo regolamento.

- La fase di negoziazione continua in cui i contratti sono conclusi secondo le regole precedentemente indicate.

Un passo molto importante, verso un livello di realismo sempre maggiore del modello, è la ricostruzione delle regole delle fasi di asta.

La versione più evoluta di JavaSum ha la giornata di contrattazione così composta:

- Il primo *tick* è dedicato interamente all'asta di apertura scissa come nella realtà nei suoi tre momenti principali:
 1. pre-asta in cui è determinato il prezzo teorico d'apertura
 2. validazione in cui è validato il prezzo teorico
 3. apertura in cui sono conclusi i contratti.
- La fase di negoziazione continua che, a discrezione dell'utilizzatore del modello, può essere costituita da un numero variabile per ogni simulazione di *tick*.
- L'ultimo *tick* è dedicato all'asta di chiusura che non presenta grosse differenze dall'asta iniziale.

5.5.1 L'ASTA DI APERTURA.

Nel mercato reale questa fase si configura come un mercato a chiamata elettronico. Ad un primo periodo, durante il quale le proposte si negoziano si accumulano, segue la determinazione di un unico prezzo a cui sono chiuse tutte le proposte compatibili.

Lo scopo di questa fase è quello di determinare un prezzo che sia espressione della domanda e dell'offerta di un quantitativo iniziale di titolo ritenuto significativo e che assicuri la continuità del mercato rispetto alla seduta precedente.

Nel modello è ricreato lo stesso identico meccanismo: per poter ottenere questo risultato sono state inserite alcune azioni che durante la simulazione sono svolte in sequenza all'interno del codice del Model:

```
// We create the list of simulation actions
```

```
//Open auction
modelActions1 = new ActionGroupImpl(getZone());

sel = SwarmUtils.getSelector("Book", "setClean");
modelActions1.createActionTo$message(theBook, sel);

sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
modelActions1.createActionTo$message(listShuffler, sel,
agentList);

sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
modelActions1.createActionForEach$message(agentList, sel);

sel = SwarmUtils.getSelector("Book", "setTheoreticalPrice");
modelActions1.createActionTo$message(theBook, sel);

sel = SwarmUtils.getSelector("Book", "setOpeningPrice");
modelActions1.createActionTo$message(theBook, sel);

sel = SwarmUtils.getSelector("Book", "opening");
modelActions1.createActionTo$message(theBook, sel);
```

Il primo metodo attivato è *setClean* il quale riporta allo stato iniziale tutte le variabili e gli *arrayList* permettendo in questo modo di iniziare un nuovo giorno di contrattazione senza trascinare dati dai giorni precedenti di contrattazione.

Il metodo *shuffleWholeList* è utilizzato semplicemente per rimescolare la lista in modo tale che gli agenti non siano interpellati sempre nello stesso ordine. *act0* è il metodo utilizzato dagli agenti durante le fasi d'asta.

I successivi tre sono quelli che permettono al modello di ricreare l'asta di apertura:

setTheoreticalPrice calcola il prezzo teorico d'apertura sulla base delle 4 regole stabilite da Borsa Italiana s.p.a. inseguito descritte;

setOpeningPrice valida il prezzo teorico se esso non ha uno scostamento superiore al $\pm 10\%$ dal prezzo di riferimento del giorno precedente;

Opening consente di concludere i contratti e di informare gli agenti del buon esito delle loro proposte.

Ma come è effettivamente strutturata l'asta di apertura all'interno dei file del modello? *setOrderBeforeOpeningFromAgent* viene sostituito in questa nuova versione da *setOrderInOpeningOrClosingAuctionsFromAgent* all'interno del quale ho inserito l'attivazione del metodo *setTableForTheoreticalPrice*. Lo scopo di questo inserimento è la creazione di una tabella in base della quale sia possibile stabilire in ogni momento il valore del prezzo teorico di apertura.

Borsa Italiana s.p.a. ha indicato le seguenti regole di calcolo del prezzo teorico (Damilano, 2002):

1. il prezzo teorico d'apertura è quello che consente la negoziazione della maggiore quantità di titoli;

2. qualora a prezzi diversi siano negoziabili pari quantità di titoli, il prezzo teorico d'apertura risulta quello al quale è possibile negoziare la quantità che produce minor differenza tra il volume degli acquisti e il volume delle vendite;
3. il prezzo teorico d'apertura risulta corrispondente al prezzo più vicino al prezzo di riferimento dell'ultima seduta, qualora vi siano uguali minori quantità non negoziabili a prezzi diversi;
4. se i prezzi sono equidistanti dal prezzo di riferimento, il prezzo teorico d'apertura risulta il maggiore tra i due.

Prendendo spunto da un esempio riportato sul testo sopra indicato, ho immaginato di ricreare internamente al programma una tabella così strutturata:

Prezzo teorico	Totale acquisti	Totale vendite	Quantità negoziabile	Differenza
4.93	3.000	58.000	3.000	55.000
4.92	33.000	58.000	33.000	25.000
4.91	39.000	33.000	33.000	6.000
4.90	41.000	11.000	11.000	30.000

Il risultato che volevo ottenere era quello di avere a disposizione per ogni prezzo il quantitativo che si sarebbe potuto trattare, nel caso in cui a fine asta quel determinato prezzo fosse stato validato.

Non essendo possibile conoscere a priori il numero dei prezzi diversi che sono proposti in ogni asta, ho creato un *arrayList* denominato *setTableForTheoreticalPrice*.

Aggiornando opportunamente la tabella ad ogni inserimento di proposta, si può applicare con molta semplicità le prime due regole precedentemente elencate. Come si può vedere dal codice in appendice i metodi sono molto strutturati e complessi in quanto fin da subito la tabella è stata adattata per poter trattare altri tipi di ordini che saranno in futuro inseriti (ad esempio ordini al meglio trattati nel prossimo paragrafo).

Viste le innumerevoli operazioni di controllo che il programma effettua, ogni volta che arriva un nuovo ordine, per aggiornare le righe della tabella, l'*arrayList*, attraverso il metodo *toArray()*, è trattato come se fosse una matrice consentendo una maggiore rapidità di accesso agli elementi.

I passaggi effettuati internamente dal programma sono:

l'ordine che arriva al *book* è subito passato al metodo *setTableForTheoreticalPrice* il quale a sua volta lo trasferisce al metodo *setRows*; l'aggiornamento della tabella inizia con la creazione di una riga che contenga i dati utili all'individuazione del prezzo teorico, e che sia strutturata secondo la nomenclatura della tabella, in modo da semplificarne l'utilizzo.

Inizialmente tale riga riporta il prezzo, la quantità, il minimo tra i dati in seconda e terza posizione dell'ordine standard inserito (una delle quali pari a

zero riferendosi o ad acquisto o a vendita e non a entrambe) e il valore assoluto della precedente differenza.

Nel caso in cui l'ordine che arriva è al meglio, quindi con prezzo zero ma con quantità non negativa, e la tabella è ancora vuota, l'operazione effettuata è l'inserimento della riga appena creata. Le quantità degli ordini a prezzo d'apertura sono tenute in memoria le quantità in modo che siano addizionate alla quota di azioni relativi a tutti i prezzi successivamente proposti, in quanto questi ordini saranno chiusi per qualsiasi prezzo d'apertura. Viceversa se vi sono già righe nella tabella e l'ordine è al meglio, si somma direttamente la quantità e si aggiorna sia la quantità minima negoziabile che il valore assoluto della differenza.

Se invece il prezzo è maggiore di 0, ad indicare un ordine non a prezzo d'asta, si deve controllare se il prezzo è già presente in modo tale da non avere una riga doppia. Al contrario se non è ancora presente, si inserisce la nuova riga in modo che i prezzi siano in ordine decrescente e che le quantità siano pari a quella che si avrebbe avuta a quel prezzo senza tenere conto dell'ultimo ordine. Così facendo il metodo nella sua parte finale ha la possibilità di aggiornare le quantità di tutte le righe semplicemente addizionando la nuova quantità giunta al *book*.

Una volta finita la fase di inserimento degli ordini si passa al calcolo del prezzo teorico. Il metodo utilizzato è *setTheoreticalPrice* il quale sulla base dei dati della tabella applica le quattro regole stabilite da Borsa Italiana s.p.a.. Dovendo anch'esso effettuare molte interazione sulle righe e sui dati della tabella si tratta l'*arrayList* come una matrice. Il primo ciclo *for* individua la quantità massima negoziabile e indica se vi sono più prezzi che consentono di produrla.

Il prezzo teorico viene posto uguale a 0 se la tabella è vuota o la quantità massima negoziabile è pari a 0, uguale all'unico prezzo se vi è una sola quantità massima altrimenti si passa alla seconda regola. Per rendere più veloce il flusso di operazioni effettuate dal metodo si tiene in memoria sia l'indice della prima quantità massima negoziabile sia il numero di esse in modo tale da non dover più rileggere tutta la matrice ma solo la parte che interessa. Il ciclo successivo rispecchia esattamente le operazioni appena descritte con la variante di prendere in considerazione la differenza tra i volumi negoziabili. Nel caso in cui vi siano uguali minori differenze si passa alla terza regola: si procede creando un *array* delle differenze tra i prezzi interessati e il prezzo di riferimento. Tale prezzo è calcolato alla fine di ogni giornata e consente di prendere come prezzo teorico quello più vicino ad esso dando una certa continuità alle trattazioni. In fine se vi sono due prezzi equidistanti si passa alla quarta regola prendendo come prezzo il maggiore tra i due.

Una volta individuato il prezzo teorico, la fase di validazione nel modello è ricreata con il metodo *setOpeningPrice*, il quale seguendo il regolamento di borsa stabilisce che il prezzo teorico diventa d'apertura se non discosta dal prezzo di riferimento di $\pm 10\%$. Se così non fosse il prezzo d'asta è posto uguale a 0.

Il metodo *setOpening* è quello che, nel caso in cui sia stato validato il prezzo d'apertura, prepara gli ordini inseriti in fase d'asta alla conclusione dei contratti; viceversa se il prezzo risulta indeterminato, riorganizza il *book* in modo da poter operare correttamente durante la fase di asta continua.

La prima parte del metodo è stata scritta in prospettiva dell'inserimento degli ordini al meglio, per cui assegna ad essi il prezzo d'apertura (se determinato) e modifica il parametro che indicava il tipo di ordine ponendolo uguale a 0. La seconda parte, invece prende in considerazione i casi di indeterminazione che sono:

1. Il *book* è vuoto;
2. Un lato è vuoto e l'altro contiene solo ordini con limite: quest'ultimi passano al *book* di continua con il loro prezzo e la loro priorità;
3. Un lato vuoto e l'altro con tutti ordini con prezzo d'apertura: tali proposte passano al *book* di continua con prezzo pari a quello di controllo che in questa fase risulta quello di riferimento;
4. Un lato vuoto e l'altro con ordini sia a prezzo d'apertura che con prezzo limitato: le proposte a prezzo d'apertura prendono come prezzo il miglior prezzo limite del proprio lato mentre le altre mantengono il loro prezzo e la loro priorità.
5. entrambe i lati con solo prezzi limitati e il miglior in vendita risulta minore del migliore in acquisto: tutti gli ordini passano al *book* di continua con il loro prezzo e la loro priorità;
6. entrambe i lati con solo ordini a prezzo d'asta: tutti prendono prezzo di controllo (prezzo di riferimento) e mantengono le loro priorità.

In questo metodo si ha l'unica semplificazione rispetto alla realtà che consiste nel non effettuare altre aste nel caso in cui il prezzo teorico non venga validato. In questo caso se vi sono ordini al prezzo d'apertura ho assegnato ad essi il miglior prezzo del loro lato e se non ci sono prezzi con limite, il prezzo di riferimento, operando una semplificazione che si avvicina a regole utilizzate in altri casi nella realtà.

Ovviamente tutti gli ordini con prezzo di apertura sono inseriti in appositi *arrayList* secondo il loro ordine di inserimento, non potendo sapere a priori la posizione in cui saranno inseriti. Una volta prezzati il loro inserimento all'interno del *book* è effettuato sia ponendoli in ordine di prezzo, ma anche controllando l'ultimo parametro dell'ordine standard, in cui è indicato il numero progressivo di inserimento, mantenendo per ogni ordine la propria priorità temporale.

Si tenga presente che l'ordine nella versione più evoluta risulta così strutturato:

Prezzo, Numero dell'agente che propone, Quantità, Parametro, Numero progressivo di inserimento dell'ordine.

L'ultima parte del metodo si occupa della conclusione dei contratti. Se vi è un prezzo d'apertura validato, per cui diverso da zero, il metodo conclude i contratti fin quando, contemporaneamente, vi sono ordini nei due lati e i migliori prezzi che via via si trovano in prima posizione risultano, nel lato degli acquisti, maggiore o uguale e nel lato delle vendite minore e uguale al prezzo d'asta.

Il prezzo di riferimento più volte citato precedentemente è calcolato dal metodo *setReferencePrice*, seguendo quanto stabilito da Borsa Italiana s.p.a.: risulta uguale al prezzo d'asta di chiusura del giorno precedente e qualora sia risultato indeterminato, si utilizza il prezzo medio ponderato dell'ultimo 10% della quantità negoziate.

A tal fine ho dovuto creare una specie di registro di tutti i contratti conclusi chiamata *closingContractLog*. Nel caso in cui il prezzo di chiusura sia uguale a 0 si calcola a quanto equivale il 10% della quantità totale trattata e sulla base di questa si procede al calcolo della media ponderata.

5.5.2 L'ASTA DI CHIUSURA.

```
//Close auction
modelActions3 = new ActionGroupImpl(getZone());

sel = SwarmUtils.getSelector("Book", "recreateTableForTheoreticalPrice");
modelActions3.createActionTo$message(theBook, sel);

sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
modelActions3.createActionTo$message(listShuffler, sel, agentList);

sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
modelActions3.createActionForEach$message(agentList, sel);

sel = SwarmUtils.getSelector("Book", "setTheoreticalPrice");
modelActions3.createActionTo$message(theBook, sel);

sel = SwarmUtils.getSelector("Book", "setOpeningPrice");
modelActions3.createActionTo$message(theBook, sel);

sel = SwarmUtils.getSelector("Book", "opening");
modelActions3.createActionTo$message(theBook, sel);
```

La parte di codice sopra riportato si riferisce alle azioni che il *Model* compie per poter effettuare l'asta di chiusura. Nel mercato reale ciò che avviene in questa fase è speculare rispetto all'asta di apertura per cui è sufficiente richiamare i metodi precedentemente descritti. Per poter distinguere se si è in una fase o nell'altra ho fatto in modo che l'oggetto *book* sia a conoscenza del

tick in cui si sta operando: il programma scrive a video, se l'utente pone la variabile *printing* uguale a 1 nella finestra del Model, in che fase ha ricevuto ogni ordine a seconda che si sia nel primo o nell'ultimo istante del giorno fittizio di simulazione.

Ovviamente per poter operare correttamente ho dovuto ricreare la tabella precedentemente descritta con gli ordini che non sono ancora eseguiti all'inizio dell'asta di chiusura. Il metodo che permette di considerare anche queste proposte è *recreateTableForTheoreticalPrice*, il quale semplicemente passa al metodo *setTableForTheoreticalPrice* tutti gli ordini ancora presenti nel *book*. Successivamente gli agenti inseriscono i loro ordini procedendo con la fase di pre-asta, d'asta e di validazione.

5.6 INSERIMENTO DEGLI ORDINI AL MEGLIO.

Di seguito riporto integralmente l'articolo del regolamento di borsa in cui vengono indicati tutti i tipi di proposte che possono essere immessi nelle varie fasi e la loro modalità di esecuzione:

Articolo 4.1.5 del REGOLAMENTO DEI MERCATI ORGANIZZATI E GESTITI DA BORSA ITALIANA S.P.A. deliberato dall'Assemblea di Borsa Italiana S.p.A. del 15 dicembre 2003 e approvato dalla Consob con delibera n. 14439 del 24 febbraio 2004 entrato in vigore il giorno 8/3/2004

(Proposte di negoziazione)

1. Nelle fasi di pre-asta, le proposte di negoziazione:
 - a) possono essere immesse con limite di prezzo o senza limite di prezzo (proposte "al prezzo d'asta");
 - b) possono essere specificate con le seguenti modalità di esecuzione:
 - 1) "valida fino alla cancellazione": l'eventuale quantità ineseguita della proposta permane nel mercato fino al termine della seduta, quando viene automaticamente cancellata;
 - 2) "valida fino alla data specificata": la proposta permane nel mercato mantenendo la priorità temporale originaria per la quantità ineseguita
 - 3) "esegui e cancella": la proposta viene eseguita, anche parzialmente, per la quantità disponibile in asta; l'eventuale saldo residuo viene cancellato automaticamente al termine della fase di asta;
 - c) se immesse al prezzo di asta, assumono dinamicamente il prezzo al quale avrebbero le maggiori possibilità di essere eseguite.
2. Durante la negoziazione continua, le proposte:
 - a) possono essere immesse con limite di prezzo o senza limite di prezzo ("proposte al prezzo di mercato" o "applicazioni");
 - b) possono essere specificate con le seguenti modalità di esecuzione:
 - 1) "valida fino alla cancellazione";

2) "esegui e cancella": la proposta viene eseguita, anche parzialmente, per le quantità disponibili e al prezzo indicato oppure al miglior prezzo del lato opposto del mercato se la proposta è senza limite di prezzo; l'eventuale saldo residuo viene cancellato automaticamente;

3) "esegui quantità minima specificata": la proposta viene eseguita anche parzialmente almeno per il quantitativo minimo e alle condizioni di prezzo indicate o migliori; se detto quantitativo non è disponibile nel mercato la proposta viene cancellata automaticamente;

4) "tutto o niente": la proposta viene eseguita unicamente per l'intero quantitativo indicato al momento dell'inserimento e alle condizioni di prezzo indicate; se ciò non è possibile la proposta viene cancellata automaticamente;

5) "valida fino alla data specificata";

c) se immesse senza limite di prezzo, possono essere altresì specificate con la modalità di esecuzione "esegui comunque"; in tal caso, la conclusione dei contratti avviene automaticamente ai prezzi delle proposte di segno contrario più convenienti, ordinate secondo i criteri di priorità di cui all'articolo 4.1.4, comma 3;

d) se immesse con limite di prezzo, possono essere altresì specificate con la modalità di esecuzione "esponi al raggiungimento del prezzo specificato": la proposta viene accettata, ma è esposta sul mercato solo al raggiungimento del prezzo indicato;

e) se immesse con limite di prezzo, gli operatori possono limitare la visualizzazione nel mercato a una quantità parziale a cui deve corrispondere un controvalore compreso tra il valore minimo stabilito da Borsa Italiana nelle Istruzioni e il controvalore totale della proposta; tali proposte sono automaticamente cancellate all'inizio di una fase di pre-asta, anche nel caso in cui quest'ultima sia attivata ai sensi dell'articolo 4.9.2, comma 2, lettera b).

3. Durante la negoziazione continua e la pre-asta di chiusura, possono essere immesse con o senza limite di prezzo proposte "valide solo in asta di chiusura". Durante la fase di negoziazione continua le proposte non sono esposte sul mercato: esse sono conservate nel sistema per la loro partecipazione alle negoziazioni nella fase di pre-asta e sono registrate secondo la priorità temporale determinata dall'orario di immissione; all'inizio della fase di pre-asta sono esposte sul mercato con priorità temporale inferiore a quella delle proposte già presenti sul mercato e priorità temporale superiore a quella delle proposte immesse nella fase di pre-asta stessa. Le proposte vengono automaticamente cancellate al termine della seduta per l'eventuale quantità ineseguita.

4. Non è consentita l'immissione di proposte con limite di prezzo aventi prezzi superiori o inferiori ai limiti percentuali di variazione massima dei prezzi stabiliti da Borsa Italiana nelle Istruzioni.

Il modello Sum-0.66 prendeva in considerazione solo ordini con limite di prezzo; l'ultima versione di JavaSum come precedentemente anticipato prevede l'inserimento di ordini senza limite con alcune caratteristiche che di seguito verranno presentate.

Nella fase di pre-asta, il tipo di proposta che può essere immesso è l'ordine a prezzo d'asta, il quale inizialmente non ha limite ma non appena finisce la prima fase prende prezzo secondo le modalità presentate nel paragrafo precedente.

Nell'inserimento degli ordine senza limite di prezzo, ho preferito rendere rigida la determinazione del tipo di proposta. Quelli senza limite per poter essere trattati come tali devono presentare due caratteristiche:

- prezzo uguale a 0;
- parametro uguale a 1 o 2 a seconda che siano in acquisto o in vendita.

Viceversa per essere ordini con limite di prezzo devono avere:

- prezzo maggiore o minore di 0 a seconda che siano in acquisto o in vendita;
- parametro uguale a 0.

Tutte le altre varianti, per ora, non sono assolutamente prese in considerazione a differenza dei casi in cui sia il prezzo che il parametro o la sola quantità siano uguali a zero. In questi casi si ipotizza che l'agente interpellato non è interessato a piazzare un ordine.

Nella fase d'asta gli ordini senza limite sono lavorati dal programma come descritto nei precedenti paragrafi, mentre in fase di contrattazione continua si è scelto di considerarli come se fossero ordini del tipo "esegui comunque". La scelta è stata effettuata per poter avere un tipo di proposta che rendesse più liquido il mercato creando la possibilità per gli agenti interessati di vedere sicuramente tutto il quantitativo proposto soddisfatto.

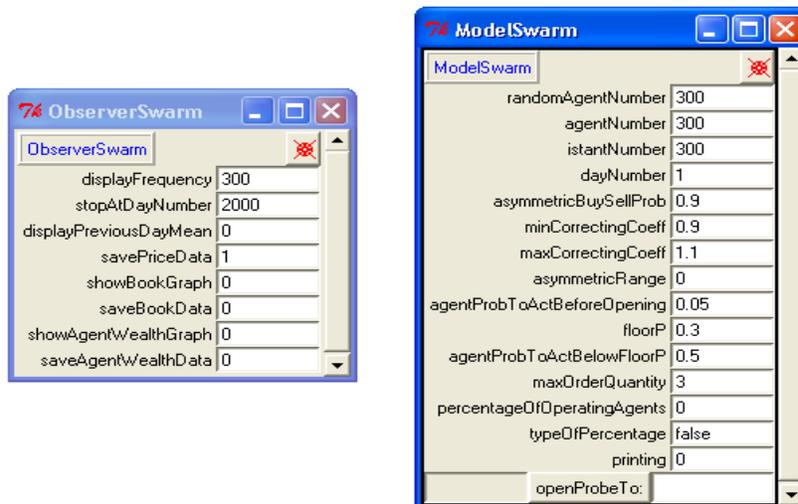
CAPITOLO 6

ALCUNE SIMULAZIONI E STUDIO DEI RISULTATI OTTENUTI

In questo capitolo saranno presi in considerazione i risultati ottenuti da simulazioni che differiscono tra di loro per il variare di un solo parametro, in modo da verificare e osservare l'influenza che questi hanno sul modello. Per maggiore completezza, passerò da una prima comparazione tra ciò che a suo tempo era stato rilevato da Chinaglia nella propria tesi (2004) e quello che si può osservare da simulazioni effettuate con il nuovo modello JavaSum. Alla fine del capitolo sarà preso in considerazione ciò che comporta un'approssimazione del prezzo a tre decimali, rendendo il modello ancora più vicino alla realtà.

6.1 ESPERIMENTI SENZA LE ASTE.

Volendo avere dati comparabili, inizialmente ho la necessità di utilizzare una versione antecedente a quella con le aste. JavaSum 0.4 permette di utilizzare solo agenti *random* e non prevede aste. Esattamente come in Sum-0.66 vi è una fase di pre-apertura in cui sono raccolte le proposte senza concludere contratti, in modo tale da non avere un *book* vuoto all'inizio di ogni giornata simulata. L'unica differenza è la gestione delle quantità, che però non altera assolutamente il meccanismo di formazione dei prezzi. I dati utilizzati sono esattamente gli stessi che Chinaglia aveva proposto nell'esperimento base in Sum²²:



²² Capitolo 6.3 della sua tesi.

Considerando le innumerevoli variabili che interagiscono nel modello di simulazione, l'obiettivo dei dati di seguito riportati non è certamente la comprensione puntuale delle dinamiche dei prezzi al variare di determinati parametri, ma ha come traguardo l'individuazione di possibili cause che generano determinati fenomeni.

L'obiettivo del primo esperimento è individuare come le serie storiche dei prezzi si modificano, nella simulazione, aumentando la sola probabilità di operare durante la fase di pre-apertura.

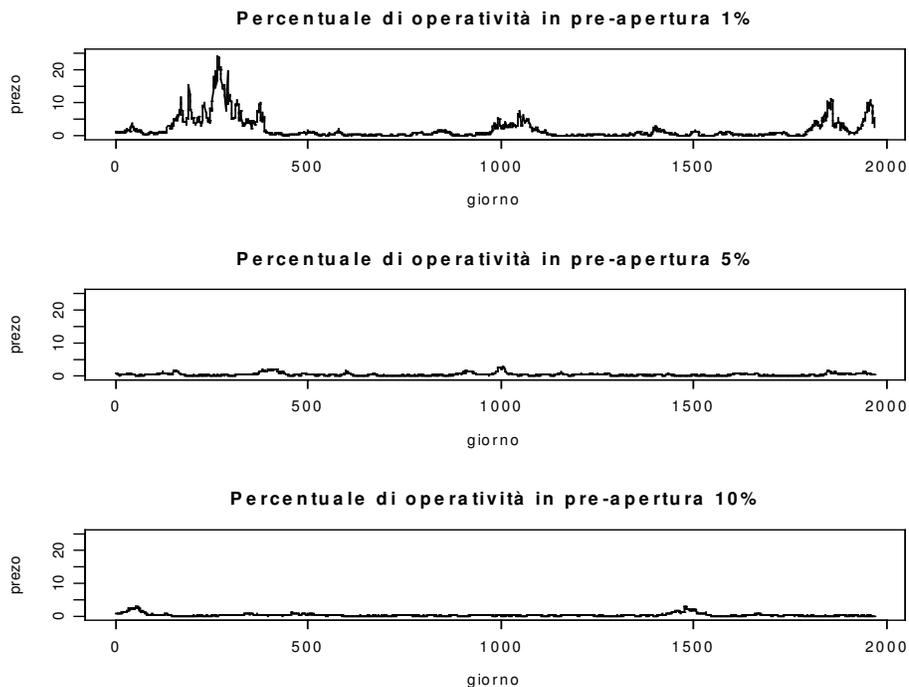
Il modello è utilizzato, inizialmente, con una probabilità molto bassa dello 0.01 (1%), per poi passare al 5% ed infine al 10%.

Ciò che si era notato in Sum, è confermato con JavaSum, considerando che a questo livello non vi sono grosse differenze.

I grafici sottostanti sono prodotti con un programma *open source* chiamato R, il quale mi permette anche agevolmente di calcolare alcuni valori statistici che spiegano in modo più rigoroso, quello che su di un grafico con migliaia di dati può sfuggire.

La cosa che balza subito all'occhio dai primi grafici è che la variabilità del prezzo tende a diminuire al crescere della probabilità di operare nella fase di pre-apertura. Gli andamenti di forte crescita e decrescita tendono a diminuire sia in numero che in intensità.

Quello che già Chinaglia aveva affermato può ora essere confermato anche con il nuovo modello: un *book* con pochi ordini al suo interno e molto probabilmente poco omogenei, favoriscono l'insorgere di bolle e crash.



Nella tabella che segue sono indicati i valori statistici che le tre serie storiche dei prezzi presentano in tutto il loro intervallo:

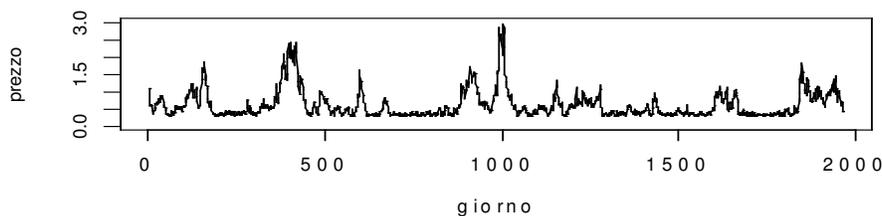
VALORI	Probabilità di operare in pre-apertura		
	all'1%	al 5%	al 10%
Min	0.2926	0.2975	0.2909
1st Qu	0.4342	0.3530	0.3590
Median	0.8490	0.4590	0.4559
Mean	2.0870	0.6222	0.5963
3rd Qu	2.3480	0.7460	0.6420
Max	23.9900	2.9730	3.2470
Variance	9.41269	0.06801724	0.06983127
Standard deviation	3.068011	0.4103938	0.4275645

Come si può notare la differenza più marcata si ha tra la prima prova e le altre. A tal proposito si tenga presente che l'1% significa avere mediamente solo tre ordini all'inizio della contrattazione continua: la probabilità che il book si svuoti è più elevata, per cui la prima proposta che arriva in questo caso risulta la migliore ed essa può differire dall'ultimo prezzo formatosi sul mercato del 10% ($minCorrectingCoeff = 0.9$ e $maxCorrectingCoeff = 1.1$).

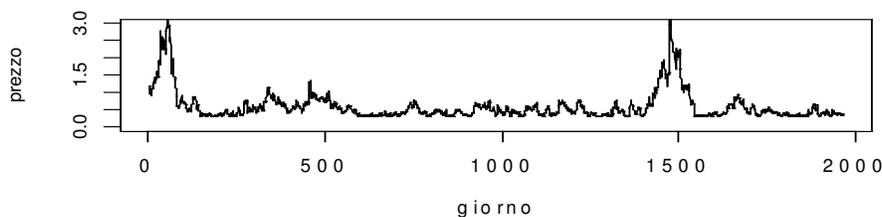
La situazione è meno probabile negli ultimi due casi che, rispettivamente, preparano il book con 15 e 30 ordini in attesa.

Ingrandendo gli ultimi due grafici è possibile osservare che, pur non avendo differenze significative nei valori delle rispettive varianze, l'ultimo presenta minori formazioni di crescite e discese prolungate del prezzo.

Percentuale di operatività in pre-apertura 5 %



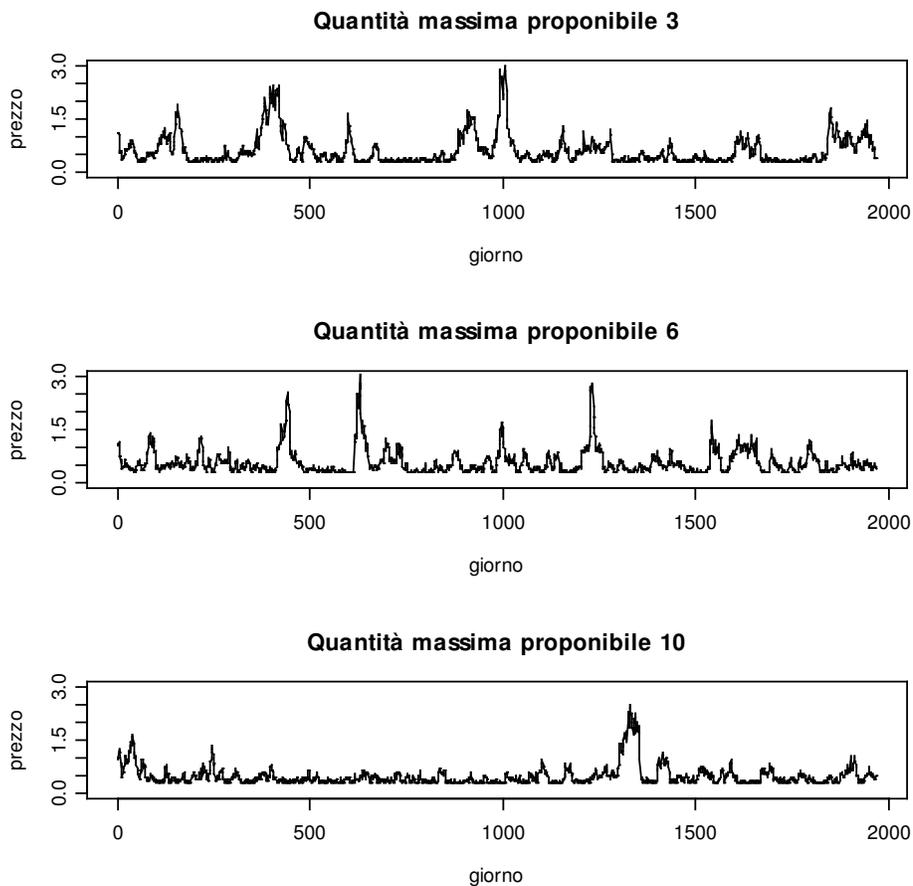
Percentuale di operatività in pre-apertura 10 %



La maggiore linearità è evidente.

Vediamo ora se anche i risultati che si ottengono variando la quantità massima proponibile da parte di ogni agente possono essere confermati.

Nel modello Sum si era giunti alla conclusione che all'aumentare della quantità massima proponibile, aumentava la variabilità del prezzo. I grafici che seguono ripropongono la serie storica del prezzo che si è formato facendo operare 300 agenti *random* per 2000 giorni, con la probabilità di operare in fase di pre-apertura pari al 5%. Rispettivamente *maxOrderQuantity* è stata impostata pari a 3 (uno dei casi precedenti), 6, 10 per continuare a confrontarli con i risultati proposti da Chinaglia.



A differenza di quanto avveniva con il modello Sum, i risultati che si ottengono sono leggermente diversi. Nei primi due grafici non si notano grosse differenze dal punto di vista della variabilità: entrambe presentano il formarsi di un discreto numero di fenomeni di crescita e decrescita rapida del prezzo. La differenza da ciò che a suo tempo aveva osservato Chinaglia si trova

nell'ultimo grafico. Quello che mi aspettavo era un aumento della variabilità dei prezzi all'aumentare della quantità negoziata sul mercato virtuale. Con Sum era stata riscontrata una forte volatilità nei primi 500 giorni di simulazione con maxOrderQuantity=10, per poi, nei restanti 1500 giorni, riavere un andamento molto più lineare del tutto simile a quello appena proposto nei grafici precedenti.

Nella tabella seguente riporto i dati che ho ottenuto con JavaSum:

VALORI	Quantità massima proponibile		
	3	6	10
Min	0.2975	0.2956	0.2952
1st Qu	0.3530	0.3629	0.3399
Median	0.4590	0.4780	0.4107
Mean	0.6222	0.5878	0.4985
3rd Qu	0.7460	0.6787	0.5443
Max	2.9730	3.0490	2.4700
Variance	0.1684231	0.1190886	0.07809204
Standard deviation	0.4103938	0.3450922	0.2794495

Tenendo presente che gli agenti operano in modo del tutto casuale, sia per quanto riguarda la quantità che per il prezzo, faccio alcune considerazioni al fine di capire quale dei due risultati rispecchia di più la realtà.

Fermo restando il numero degli agenti e sapendo che tutti operano una sola volta per *tick*, l'aumento della quantità massima proponibile negli ordini sicuramente aumenta l'imprevedibilità di ciò che avviene sul mercato. Gli scenari che si aprono sono molto ampi e diversi fra loro: potendo essere più ampia la quantità proposta per ogni ordine, la variabilità dipende molto dal concentrarsi o meno di ordini opposti con quantità simili (meno probabile perché il *range* di variazione è più ampio in questo caso). Un esempio può rendere meglio i concetti appena esposti: sapendo che il numero degli agenti è fisso, e in un certo senso anche quello degli ordini proposti, una proposta con una quantità elevata ha la possibilità (ovviamente non la certezza) di rimanere più a lungo come migliore proposta riducendo la variabilità del prezzo (se gli ordini di segno opposto successivamente immessi hanno quantità minori), viceversa una proposta per una piccola quantità ha molta più probabilità di essere subito eseguita lasciando spazio alle proposte con priorità minore.

Nel caso di possibilità di scelta della quantità tra 0 e 3, un ordine con quantità elevata (pari a 3) ha la probabilità di essere soddisfatto, nel caso di compatibilità di prezzo pari al 33% circa, dando così spazio agli ordini che hanno minore priorità. Nel caso di variazione della quantità tra 0 e 10, un ordine con elevata quantità, in questo caso 10, ha probabilità di essere interamente soddisfatto (sempre se vi è compatibilità di prezzo della proposta contraria) pari al 10%. Le cose si ribaltano per ordini con quantità piccole.

Per cui, se le migliori proposte nei due lati del *book* si riferiscono a quantità elevate è più probabile che la volatilità diminuisca.

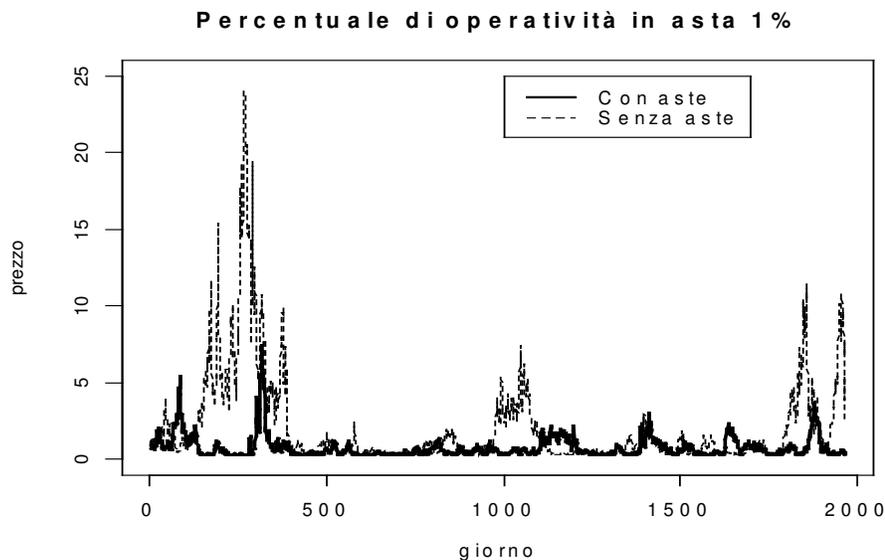
Tutto il ragionamento è fatto considerando che la quantità proposta è calcolata per mezzo di una variabile aleatoria distribuita uniformemente. Al crescere della quantità l'ordine, per una quantità elevata, ha sempre meno probabilità di essere subito soddisfatto rimanendo così all'interno del book più a lungo limitando la variabilità del prezzo. Viceversa ordini per quantità piccole hanno molta più probabilità di essere subito eseguiti aumentando la variabilità.

Sul mercato reale i titoli che sono scambiati per piccole quantità durante la giornata di borsa sono quelli più volatili (titoli sottili), per cui il risultato ottenuto con JavaSum è più realistico di quello ottenuto con Sum, tenendo però presente che in entrambe i casi la dinamica è puramente casuale avendo usato solo agenti *random*.

È ora interessante osservare cosa cambia con l'inserimento delle aste.

6.2 ESPERIMENTI CON LE ASTE.

In questo paragrafo, sempre in un'ottica di produrre dati che possano essere paragonati con quelli precedentemente descritti, rifaccio le simulazioni, prima viste, utilizzando la versione 0.7 di JavaSum. La prima parte di esperimenti è volta a osservare cosa comporta l'inserimento del meccanismo di asta nella serie dei prezzi generata dal programma. Come nel paragrafo precedente, lascio operare nella fase d'asta prima l'1%, poi il 5% e per ultimo il 10% degli agenti. La durata della simulazione è sempre 2000 giorni e gli agenti sono 300 *random*. Per un confronto immediato, rispetto a quanto detto prima, produco grafici in cui siano compresi anche le rispettive simulazioni senza aste con JavaSum 0.4.

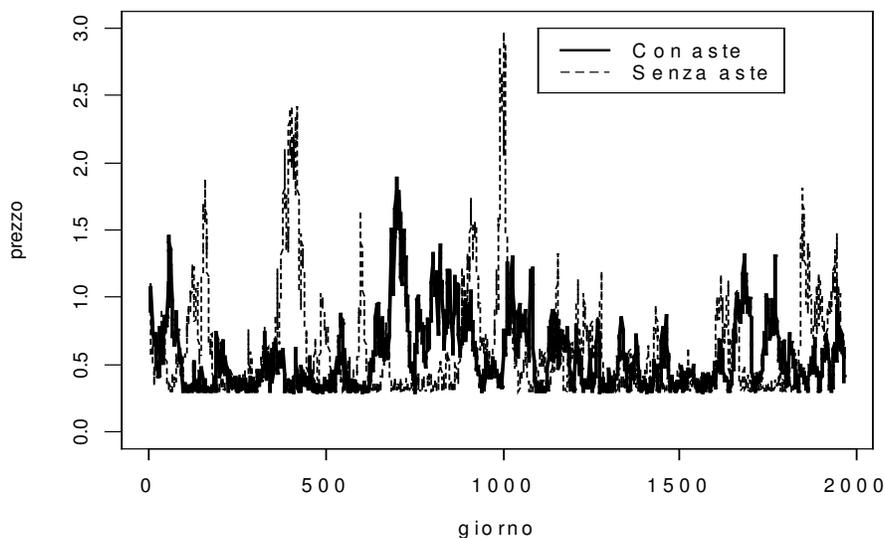


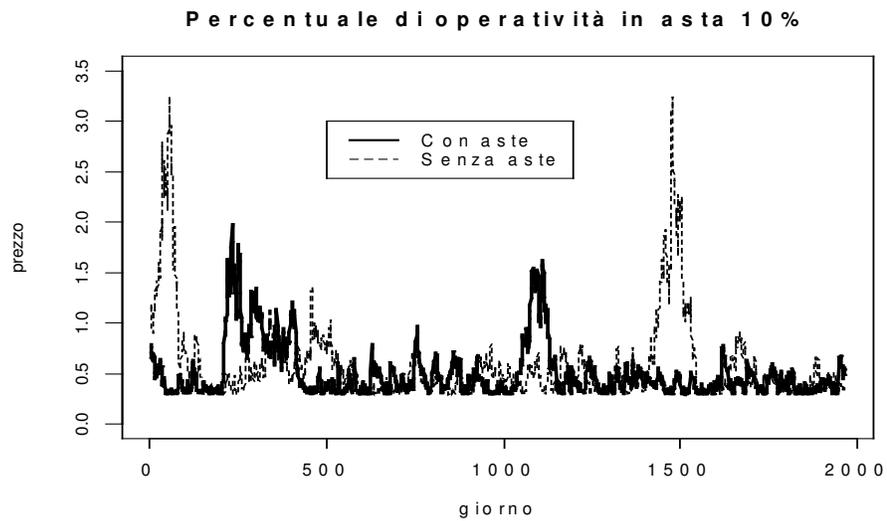
VALORI	Probabilità di operare in pre-apertura	
	all'1% senza aste	al 1% con aste
<i>Min</i>	0.2926	0.2961
<i>1st Qu</i>	0.4342	0.3585
<i>Median</i>	0.8490	0.5266
<i>Mean</i>	2.0870	0.7671
<i>3rd Qu</i>	2.3480	0.8844
<i>Max</i>	23.9900	7.4980
<i>Variance</i>	9.41269	0.4956388
<i>Standard deviation</i>	3.068011	0.7040162

Come si può dedurre da questo primo grafico di comparazione, la variabilità del prezzo diminuisce con l'inserimento delle aste. Uno dei motivi per cui capita questo è sicuramente dovuto al fatto che il book dopo le fasi d'asta presenta i due lati più corti rispetto a quanto accadeva prima. Da uno studio approfondito del periodo di preparazione ad una bolla, nella tesi di Chinaglia è stato individuato, come fenomeno ricorrente, l'allungamento della differenza tra il primo e l'ultimo prezzo nei due lati del *book*. Le aste accorciano questa differenza avendo come principale regola, per la determinazione del prezzo teorico di apertura, la conclusione dei contratti che consentono di accoppiare la maggiore quantità di azioni.

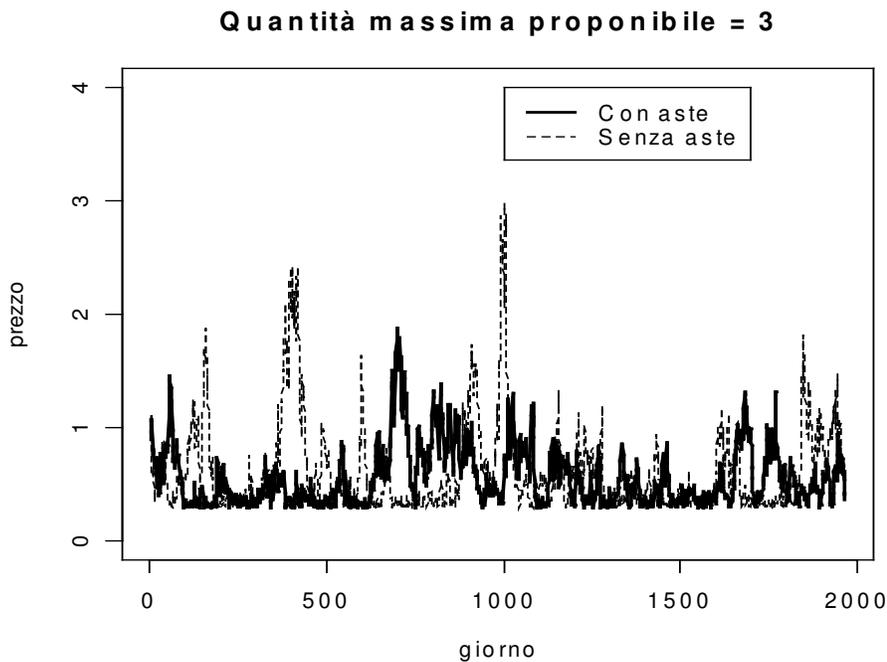
Senza aste il *book* inizia la contrattazione continua con ordini che potrebbero essere accoppiati: questa mancata conclusione crea liste più lunghe di ordini su entrambe i lati, lasciando maggiore probabilità di salite o discese di prezzo marcate.

Percentuale di operatività in asta 5 %

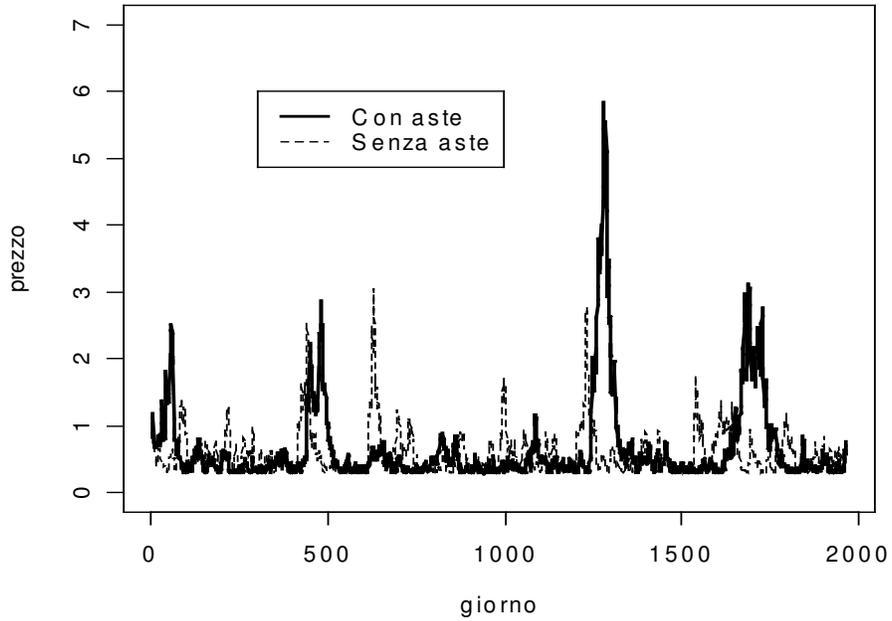




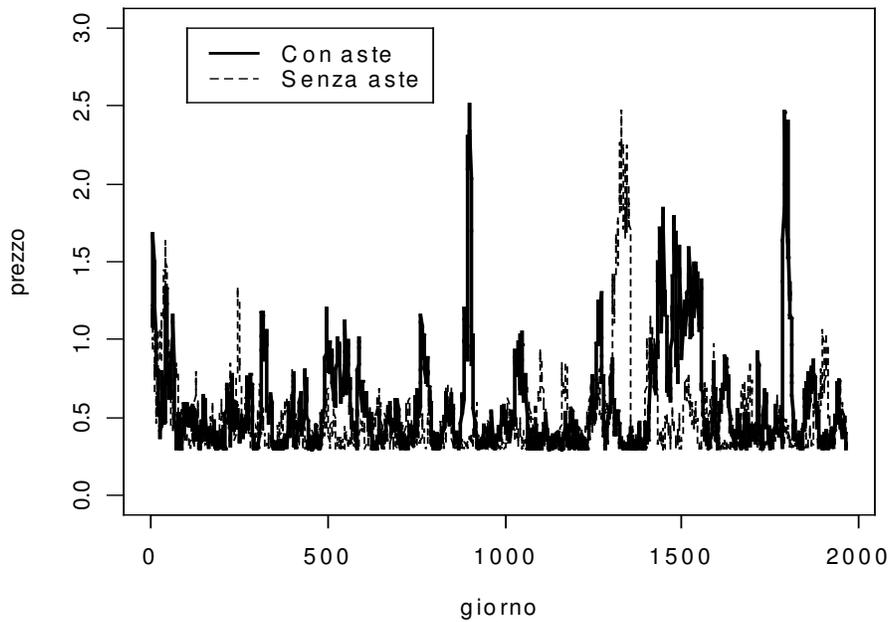
Aumentando la probabilità di operare durante le fasi d'asta si ha una sempre minore insorgenza di bolle e, come già osservato nel primo grafico di questo paragrafo, l'ampiezza della bolla tende a ridursi con l'utilizzo delle aste. I prossimi grafici si riferiscono a tre simulazioni in cui, a parità di altri parametri, tra cui la probabilità di operare in asta al 5%, è variata la quantità massima proponibile da ogni agente.



Quantità massima proponibile = 6



Quantità massima proponibile = 10



Come si può notare dai grafici, in presenza di aste, man mano che la quantità massima aumenta si ha una crescita della variabilità rispetto alle simulazioni con sola fase di pre-apertura.

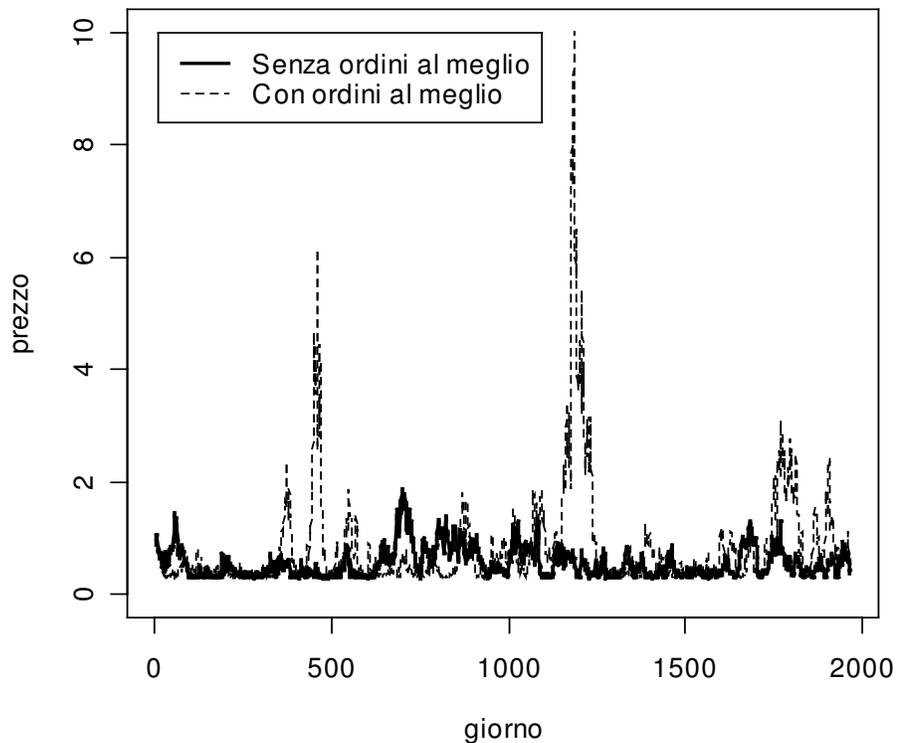
6.3 ESPERIMENTI CON ORDINI AL MEGLIO.

La versione 0.7 di JavaSum consente agli agenti di inserire nel *book* ordini con prezzo d'asta, nella rispettiva fase, e ordini al meglio, con la caratteristica operativa ECO ("esegui comunque), durante la contrattazione continua.

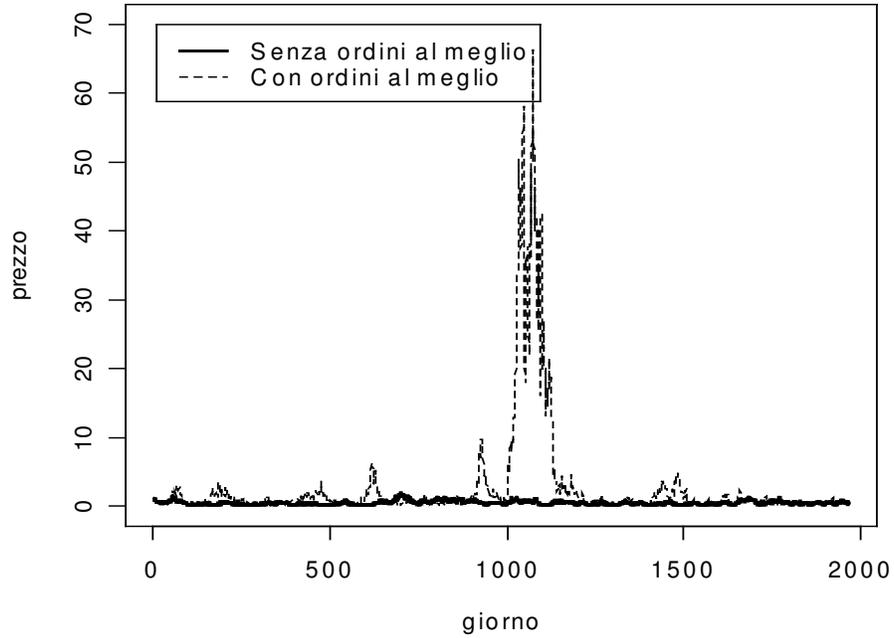
L'inserimento di un ordine al meglio permette all'agente di concludere il contratto a qualsiasi prezzo, per cui al crescere della probabilità di inserimento di tali ordini, gli aumenti e le diminuzioni rapide dei prezzi dovrebbero essere maggiori sia in numero che in ampiezza.

Vediamo graficamente ciò che ho ottenuto facendo interagire 300 agenti *random* per 2000 giorni, con una probabilità del 5% di operare in asta e crescenti probabilità di inserimento ordini al meglio, ponendo la variabile *agentProbToActWithMarketPrice* uguale a 5%, 10% e 50%.

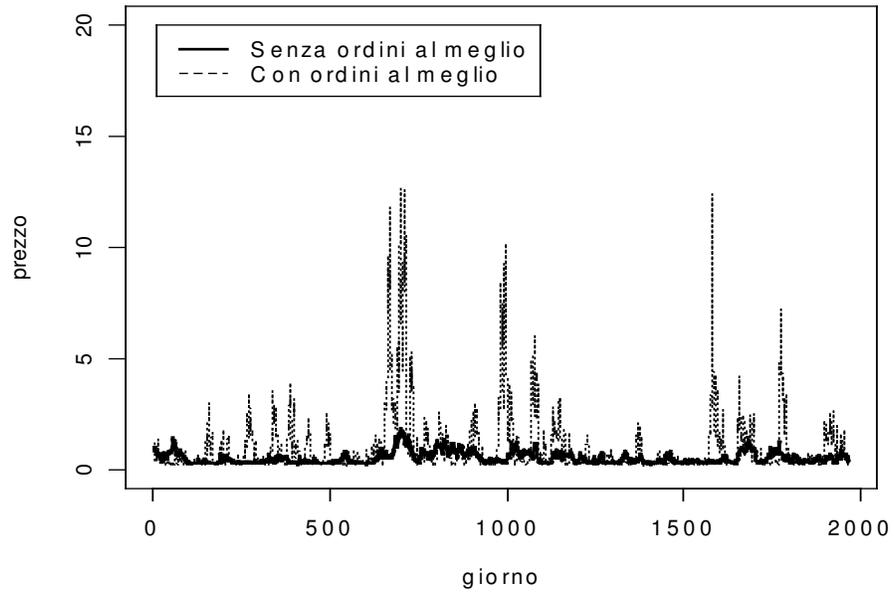
Probabilità di inserimento ordini al meglio = 5%



Probabilità di inserimento ordini al meglio = 10%



Probabilità di inserimento ordini al meglio = 50%



Nell'osservare i grafici precedenti è utile tenere presente che il regolamento dettato da Borsa Italiana s.p.a., dà precise indicazioni sugli ordini al meglio. L'articolo 4.1.10 del regolamento, nella sua parte finale, impone che l'inserimento di proposte senza limite di prezzo, nella fase di contrattazione continua, sia possibile solo in presenza di almeno una proposta di negoziazione di segno contrario con limite di prezzo. Quindi nel caso in cui gli ordini al meglio siano inseriti in un momento in cui il lato opposto è vuoto è come se non siano stati proposti visto e considerato che nella realtà non è possibile, in questo caso, inserirli. Alla luce di ciò è più facile comprendere perché aumentando molto la probabilità d'inserimento di tali ordini, pur avendo un maggiore numero di bolle, si ha una diminuzione nel *range* massimo di oscillazione. Finita la fase d'asta nel caso in cui venga validato il prezzo teorico, essendoci molti ordini al meglio il book si presenta con meno proposte. Nella fase successiva il grande numero di proposte al meglio consentono con maggiore frequenza lo svuotamento dei lati non dando più spazio ai successivi ordini al prezzo di mercato. Si può quindi concludere tanti ordini al meglio producono più bolle che possono però essere troncate in intensità dal fatto che un lato del *book* si svuoti.

Nei tre grafici questo fenomeno è evidente: passando dal 5 al 10% di probabilità si nota una crescita del *range* di oscillazione del prezzo che giunge fino ad un prezzo di circa 70; aumentando ulteriormente la probabilità d'inserimento, si nota una crescita della variabilità ma un troncamento del range di oscillazione dei prezzi dovuto quasi certamente allo svuotamento di uno dei lati del *book*.

Di seguito riporto la tabella che con i valori statistici delle serie storiche che si sono generate nelle varie simulazioni.

VALORI	Prob. Di inserimento ordini al meglio			
	0	5%	10%	50%
Min	0.2905	0.2927	0.2913	0.1555
1st Qu	0.3536	0.3673	0.4484	0.3040
Median	0.4629	0.5133	0.7587	0.4946
Mean	0.5559	0.8396	2.8440	1.0580
3rd Qu	0.6882	0.8826	1.5970	1.1930
Max	1.8840	10.0300	66.2600	12.6200
Variance	0.06801724	0.8851202	58.5787	2.166116
Standard deviation	0.2608012	0.9408082	7.653673	1.471773

La tabella conferma quanto detto precedentemente sulla base della sola osservazione dei grafici.

6.4 APPROSSIMAZIONE DEL PREZZO PROPOSTO DAGLI AGENTI.

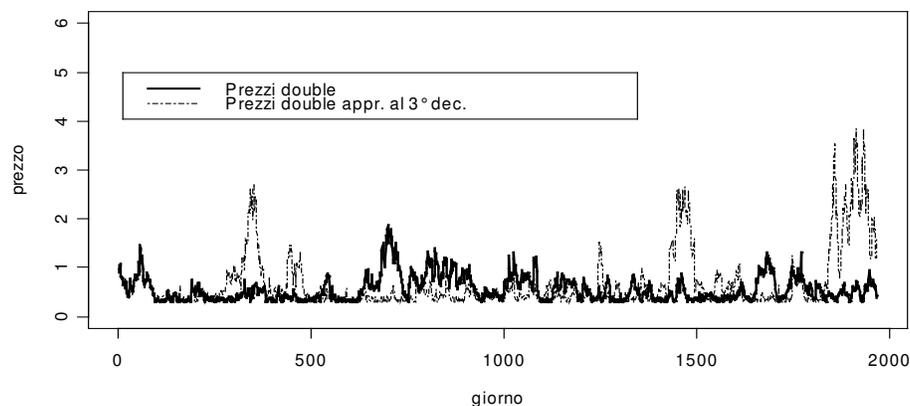
Nelle Istruzioni di Borsa Italiana entrate in vigore 1//3/2005 (ISTRBIT n° 79) l'articolo n° IA.4.1.13 pone alcuni limiti ai prezzi che possono essere proposti dagli operatori. Il comma 1 informa che i prezzi delle proposte di negoziazione possono essere multipli di valori ("tick") stabiliti per ogni strumento finanziario e per ogni seduta di Borsa in relazione ai prezzi di riferimento dei singoli strumenti come segue:

- per le azioni, warrant, quote di OICR e diritti di opzione:

<i>Prezzo di riferimento (Euro)</i>	<i>Moltiplicatore (tick)</i>
Inferiore o uguale a 0,2500	0,0001
0,2501-1,0000	0,0005
1,0001-2,000	0,0010
2,0001-5,000	0,0025
5,0001-10,0000	0,0050
Superiore a 10,0000	0,0100

Nel modello JavaSum gli agenti propongono prezzi del tipo *double* con un numero elevato di decimali. Riguardando i risultati ottenuti precedentemente mi sono domandato se potesse modificare la variabilità un'approssimazione dei prezzi ad un numero minore di decimali. La composizione del *book* in un dato istante influisce molto sull'andamento dei prezzi per cui il troncamento dei prezzi permette di avere una maggiore concentrazione degli ordini su di uno stesso prezzo. L'idea quindi che lo stesso numero di ordini siano distribuiti su di un numero minore di prezzi mi fa pensare che ci sia una maggiore probabilità di conclusione dei contratti.

Prezzi approssimati al terzo decimale



Alla luce di queste considerazioni, ho fatto in modo che ogni ordine non appena arrivi al book sia troncato nel prezzo alla terza cifra decimale, avvicinandoci maggiormente al mercato reale.

Il grafico sopra riportato mette in relazione la stessa simulazione nei due casi di prezzi *double* standard e approssimati.

Con gli stessi identici ordini proposti la serie presenta bolle più marcate, nel caso della simulazione con approssimazione, a confermare di ciò che si era supposto in precedenza.

APPENDICI

APPENDICE A

CODICE DI GENERATORE.JAVA

```
//creazione casuale degli ordini per utilizzare i metodi del book
import java.util.*;

public class Generatore {
    static Random rand = new Random();
    public static void main(String[] args) {
        int beforeOpeningNumberOrders = 50;
        int numberOrders = 0;
        Book aBook = new Book();
        aBook.setPrinting(0);
        //array
        for(int i = 0; i < beforeOpeningNumberOrders; i++) {
            /* Is variabile che segue serve per
                generare valori positivi o negativi casuali all'interno dell'array*/
            int a = rand.nextInt() % 10;
            int b = rand.nextInt() % 10;
            if(b<=3){
                double[] order = {(int)(a*rand.nextDouble()%10), Math.abs(rand.nextInt() % 10),
                Math.abs(rand.nextInt() % 10) , Math.abs(rand.nextInt() % 2)};
                System.out.println("Array lungo 4");
                aBook.setOrderBeforeOpeningFromAgent(order);
            }
            else if(b>3 && b <=6){
                double[] order = {(int)(a*rand.nextDouble()%10), Math.abs(rand.nextInt() % 10),
                Math.abs(rand.nextInt() % 10) , 0};
                System.out.println("Array lungo 3");
                aBook.setOrderBeforeOpeningFromAgent(order);
            }
            else if(b>6){
                double[] order = {(int)(a*rand.nextDouble()%10), Math.abs(rand.nextInt() % 10),
                1 , 0};
                System.out.println("Array lungo 2");
                aBook.setOrderBeforeOpeningFromAgent(order);
            }
        }
        System.out.println();
        System.out.println("Il numero di ordini prima dell'apertura è: " + aBook.orderNumber);
        System.out.println("Tabella per calcolare il prezzo teorico:");
        System.out.println();
        for(int i=0; i<aBook.tableForTheoreticalPrice.size(); i++){
            for(int y=0; y<5; y++){
                System.out.print(((double []) aBook.tableForTheoreticalPrice.get(i))[y] + " ");
            }
            System.out.println();
        }
        System.out.println();

        for(int i = 0; i < numberOrders; i++) {
            /* la variabile che segue serve per
                generare valori positivi o negativi casuali all'interno dell'array*/
            int a = rand.nextInt() % 10;
            int b = rand.nextInt() % 10;
            if(b<=3){
                double[] order = {a*rand.nextDouble(), Math.abs(rand.nextInt() % 10),
                Math.abs(rand.nextInt() % 10) , Math.abs(rand.nextInt() % 3)};
                System.out.println("Array lungo 4");
                aBook.setOrderFromAgent(order);
            }
        }
    }
}
```

Appendice A

```
else if(b>3 && b <=6){
double[] order = {a*rand.nextDouble(), Math.abs(rand.nextInt() % 10),
    Math.abs(rand.nextInt() % 10) , 0};
System.out.println("Array lungo 3");
aBook.setOrderFromAgent(order);
}
else if(b>6){
double[] order = {a*rand.nextDouble(), Math.abs(rand.nextInt() % 10),
    1 , 0};
System.out.println("Array lungo 2");
aBook.setOrderFromAgent(order);
}
}

System.out.println();
System.out.println("La sommatoria è: " + aBook.currentMeanPrice);
aBook.setMeanPrice();
System.out.println("Il prezzo medio è: " + aBook.getMeanPrice());

// un po' di stampe di controllo

System.out.println("Il numero di ordini è: " + aBook.orderNumber);
System.out.println("Numero d'ordini di vendita è: "+aBook.getSellOrderNumber());
System.out.println("Numero d'ordini di acquisto è: "+aBook.getBuyOrderNumber());
System.out.println("Il prezzo dell'ultimo contratto è: " +aBook.executedPrice);
System.out.println("Sono stati conclusi " + aBook.count + " contratti");
System.out.println();
aBook.setClean();
System.out.println();
System.out.println("Numero d'ordini di vendita è: "+aBook.getSellOrderNumber());
System.out.println("Numero d'ordini di acquisto è: "+aBook.getBuyOrderNumber());
System.out.println("Il prezzo di chiusura precedente è: "+aBook.getPreviousClosingPrice());
System.out.println("Il numero di contratti è: "+aBook.count);
}
}
```

APPENDICE B

JAVASUM: CODICE DEL *BOOK* PRIMA DELL'INSERIMENTO DELLE ASTE

```
// Book.java

import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import java.util.*;

import swarm.collections.*;

/**
 * This is the book.
 * The book works on the basis of two arrayList containing sell order in
 * increasing order or buy order in decreasing order. The orders are arrays which
 * have in the first position the price, in the second the number of agent who places
 * the order and in the third the quantity.
 * If an order obtains an immediate matching, it is not filed
 *
 * @author Antonio de Ruvo
 * @author antderu@libero.it
 */

public class Book extends SwarmObjectImpl{
    /** The index used for send message to agents. */
    public ListIndex indexAgentListIndex;
    /** If 1 many objects print data on the terminal window. */
    public int printing;
    /** The number of orders received from agents. */
    public int orderNumber;
    /** The number of share negotiated */
    public int count=0;
    /** The number of shares in buy side of book. */
    public int sharesInBuySide;
    /** The number of shares in sell side of book. */
    public int sharesInSellSide;
    /** The price of the last contract closing. */
    public double executedPrice=1;
    /** The mean price of yesterday. */
    public double meanPrice=1;
    /** The closing price of yesterday. */
    public double previousClosingPrice=executedPrice;
    /** The addition of prices*quantity to calculate the mean price. */
    public double currentMeanPrice=0;
    /** The arrayList which contains the buy orders. */
    public ArrayList buyOrderStorehouse=new ArrayList();
    /** The arrayList which contains the sell orders. */
    public ArrayList sellOrderStorehouse=new ArrayList();

    /** Constructor for a new book. */
    public Book(Zone aZone){
        super(aZone);
    }

    /** To inform the agent of executed price.
     * @see BasicSumAgent
     */
    private void message(double n, double p){
        BasicSumAgent anAgent;
        indexAgentListIndex.setOffset((int)n-1);
        anAgent = (BasicSumAgent)indexAgentListIndex.get();
        anAgent.setConfirmationOfExecutedPrice(p);
    }
}
```

Appendice B

```
/** To standardize the order. */
public double[] standardizedOrder(double[] orderFromAgent){
    double [] order=new double [4];
    if(orderFromAgent.length==2){
        double[] orderStandard={orderFromAgent[0], orderFromAgent[1], 1, 0};
        order=orderStandard;
    }
    if(orderFromAgent.length==3){
        double [] orderStandard={orderFromAgent[0], orderFromAgent[1],
            orderFromAgent[2], 0};
        order=orderStandard;
    }
    if(orderFromAgent.length==4){
        double [] orderStandard={ orderFromAgent[0], orderFromAgent[1],
            orderFromAgent[2], orderFromAgent[3]};
        order=orderStandard;
    }
    return(order);
}

/** To extract data from the orders received from agents. */
public double extract( ArrayList nameList, int indexOrder, int positionDatum ){
    Object[] b=nameList.toArray();
    double a=((double []) b[indexOrder])[positionDatum];
    return(a);
}

/** To fill sellOrderStorehouse in increasing order */
public int fillStorehouseIncreasingP(double newPrice){
    int index=0;
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        if(newPrice<extract(sellOrderStorehouse, i, 0)){
            index=i;
            break;
        }
    }
    return(index);
}

/** To fill buyOrderStorehouse in decreasing order. */
public int fillStorehouseDecreasingP( double newPrice){
    int index=0;
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        if(newPrice>extract(buyOrderStorehouse, i, 0)){
            index=i;
            break;
        }
    }
    return(index);
}

/** To know the adress of memory of the agent. */
public void setAgentListIndex(ListIndex i){
    indexAgentListIndex=i;
}

/** To set the value of option printing. */
public void setPrinting(int p){
    printing=p;
}

/** At the end of each day to calculate the mean price. */
public void setMeanPrice(){
    if(count>0)
        meanPrice=currentMeanPrice/count;
    // otherwise we keep previous value
}
```

```

/** At the beginning of each day to clean the book. */
public void setClean(){
    sellOrderStorehouse.clear();
    buyOrderStorehouse.clear();
    previousClosingPrice=executedPrice; // the last of "yesterday"
    currentMeanPrice=0;
    count=0;
    sharesInSellSide=0;
    sharesInBuySide=0;
    if(printing==1)
        System.out.println("Book: clean");
}

/** Receiving an order before opening from an agent. */
public void setOrderBeforeOpeningFromAgent(double[] orderFromAgent) {
    double [] order = standardizedOrder(orderFromAgent);
    if(printing==1)
        System.out.println("The book received a before opening order from agent " + order[1] +
            " at price " + order[0] + " for " + order[2]+ " share(s), with option = " + order[3]);
    orderNumber++;

    /* if price==0 or quantity==0 no action required, but sending a message to the agent
    * with price=0 */
    if(order[0]==0 || order[2]==0)
        message(order[1], 0.0);

    // the agent is selling at minimum price if price<0
    else if(order[0]<0) {
        message(order[1], 0.0);
        order[0]=-order[0];
    }
    /* to insert the order in the book in increasing order.
    * With add() the arrayList updates the index */
    if(sellOrderStorehouse.size()==0 ||
        order[0]>=extract(sellOrderStorehouse, (sellOrderStorehouse.size()-1), 0))
        sellOrderStorehouse.add(order);
    else
        sellOrderStorehouse.add(fillStorehouseIncreasingP(order[0]), order);
    sharesInSellSide+=order[2];
    if(printing==1) {
        for(int i=0; i<sellOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(sellOrderStorehouse, i, y) + " ");
            }
            System.out.println();
        };
    }
    System.out.println();
}

// the agent is buying at the maximum price if price>0
else {
    message(order[1], 0.0);
    /* to insert the order in the book in decreasing order.
    * With add() the arrayList updates the index */
    if(buyOrderStorehouse.size()==0 ||
        order[0]<=extract(buyOrderStorehouse, (buyOrderStorehouse.size()-1), 0))
        buyOrderStorehouse.add(order);
    else
        buyOrderStorehouse.add(fillStorehouseDecreasingP(order[0]), order);
    sharesInBuySide+=order[2];
    if(printing==1) {
        for(int i=0; i<buyOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(buyOrderStorehouse, i, y) + " ");
            }
            System.out.println();
        };
    }
    System.out.println();
}

```

Appendice B

```
    }
  }
}

/** Receiving an order when the market is open. */
public void setOrderFromAgent(double[] orderFromAgent) {
    double [] order = standardizedOrder(orderFromAgent);
    if(printing==1)
        System.out.println("The book received an order from agent " + order[1] +
            " at price " + order[0] + " for " + order[2]+ " share(s), with option = " + order[3]);
    orderNumber++;

    /* if price==0 or quantity==0 no action required, but sending a message to the agent
    * with price=0 */
    if(order[0]==0 || order[2]==0)
        message(order[1], 0.0);

    // the agent is selling at minimum price if price<0
    else if(order[0]<0) {
        /* if there is a compatible order, this one is removed and the order received
        * it's not inserted in the book. It's not necessary to shift rows because
        * with remove() the arrayList updates the index */
        while(buyOrderStorehouse.size()>0 && extract(buyOrderStorehouse, 0, 0)>=-order[0]
            && order[2]!=0) {
            if(order[2]==extract(buyOrderStorehouse, 0, 2)){
                count+=order[2];
                executedPrice=extract(buyOrderStorehouse, 0, 0);
                for(int i=0; i<order[2]; i++){
                    message(order[1], -executedPrice);
                    message(extract(buyOrderStorehouse, 0, 1), executedPrice);
                }
                currentMeanPrice+=executedPrice*order[2];
            }
            buyOrderStorehouse.remove(0);
            sharesInBuySide-=order[2];
            order[2]=0;
        }
        if(printing==1) {
            for(int i=0; i<buyOrderStorehouse.size(); i++){
                for(int y=0; y<order.length; y++){
                    System.out.print(extract(buyOrderStorehouse, i, y) + " ");
                }
                System.out.println();
            }
        }
        System.out.println();
    }
    else if(order[2]<extract(buyOrderStorehouse, 0, 2)){
        count+=order[2];
        executedPrice=extract(buyOrderStorehouse, 0, 0);
        for(int i=0; i<order[2]; i++){
            message(order[1], -executedPrice);
            message(extract(buyOrderStorehouse, 0, 1), executedPrice);
        }
        currentMeanPrice+=executedPrice*order[2];
        ((double[]) buyOrderStorehouse.get(0))[2]=extract(buyOrderStorehouse, 0, 2)
            -order[2];
        sharesInBuySide-=order[2];
        order[2]=0;
    }
    if(printing==1) {
        for(int i=0; i<buyOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(buyOrderStorehouse, i, y) + " ");
            }
            System.out.println();
        }
    }
    System.out.println();
}
}
}
else if(order[2]>extract(buyOrderStorehouse, 0, 2)){
```

```

count+=extract(buyOrderStorehouse, 0, 2);
executedPrice=extract(buyOrderStorehouse, 0, 0);
for(int i=0; i<extract(buyOrderStorehouse, 0, 2); i++){
    message(order[1], -executedPrice);
    message(extract(buyOrderStorehouse, 0, 1), executedPrice);
}
currentMeanPrice+=executedPrice*extract(buyOrderStorehouse, 0, 2);
order[2]=order[2]-extract(buyOrderStorehouse, 0, 2);
sharesInBuySide-=extract(buyOrderStorehouse, 0, 2);
buyOrderStorehouse.remove(0);
if(printing==1) {
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(buyOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
}
if(order[2]!=0) {
    message(order[1], 0.0);
    order[0]=-order[0];
/* to insert the order in the book in increasing order.
* With add() the arrayList updates the index */
if(sellOrderStorehouse.size()==0 ||
order[0]>=extract(sellOrderStorehouse, (sellOrderStorehouse.size()-1), 0))
sellOrderStorehouse.add(order);
else
    sellOrderStorehouse.add(fillStorehouseIncreasingP(order[0]), order);
sharesInSellSide+=order[2];
if(printing==1) {
    for(int i =0; i<sellOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(sellOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
}
}
// the agent is buying at the maximum price if price>0
else {
    /* if there is a compatible order, this one is removed and the order received
    * it's inserted in the book. It's not necessary shift rows because
    * with remove() the arrayList updates the index */
while(sellOrderStorehouse.size()>0 && extract(sellOrderStorehouse, 0, 0)<=order[0]
&& order[2]!=0) {
    if(order[2]==extract(sellOrderStorehouse, 0, 2)){
count+=order[2];
executedPrice=extract(sellOrderStorehouse, 0, 0);
for(int i=0; i<order[2]; i++){
    message(order[1], executedPrice);
    message(extract(sellOrderStorehouse, 0, 1), -executedPrice);
}
currentMeanPrice+=executedPrice*order[2];
sellOrderStorehouse.remove(0);
sharesInSellSide-=order[2];
order[2]=0;
if(printing==1) {
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(sellOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
}
}
}
}
}

```

Appendice B

```
};
System.out.println();
}
}
else if(order[2]<extract(sellOrderStorehouse, 0, 2)){
count+=order[2];
executedPrice=extract(sellOrderStorehouse, 0, 0);
for(int i=0; i<order[2]; i++){
message(order[1], executedPrice);
message(extract(sellOrderStorehouse, 0, 1), -executedPrice);
}
currentMeanPrice+=executedPrice*order[2];
((double[]) sellOrderStorehouse.get(0))[2]=extract(sellOrderStorehouse, 0, 2)
-order[2];
sharesInSellSide-=order[2];
order[2]=0;
if(printing==1) {
for(int i=0; i<sellOrderStorehouse.size(); i++){
for(int y=0; y<order.length; y++){
System.out.print(extract(sellOrderStorehouse, i, y) + " ");
}
System.out.println();
};
System.out.println();
}
}
else if(order[2]>extract(sellOrderStorehouse, 0, 2)){
count+=extract(sellOrderStorehouse, 0, 2);
executedPrice=extract(sellOrderStorehouse, 0, 0);
for(int i=0; i<extract(sellOrderStorehouse, 0, 2); i++){
message(order[1], executedPrice);
message(extract(sellOrderStorehouse, 0, 1), -executedPrice);
}
currentMeanPrice+=executedPrice*extract(sellOrderStorehouse, 0, 2);
order[2]=order[2]-extract(sellOrderStorehouse, 0, 2);
sharesInSellSide-=extract(sellOrderStorehouse, 0, 2);
sellOrderStorehouse.remove(0);
if(printing==1) {
for(int i=0; i<sellOrderStorehouse.size(); i++){
for(int y=0; y<order.length; y++){
System.out.print(extract(sellOrderStorehouse, i, y) + " ");
}
System.out.println();
};
System.out.println();
}
}
}
if(order[2]!=0) {
message(order[1], 0.0);
/* to insert the order in the book in decreasing order.
* With add()the arrayList updates the index */
if(buyOrderStorehouse.size()==0 ||
order[0]<=extract(buyOrderStorehouse, (buyOrderStorehouse.size()-1) ,0))
buyOrderStorehouse.add(order);
else
buyOrderStorehouse.add(fillStorehouseDecreasingP(order[0]),order);
sharesInBuySide+=order[2];
if(printing==1) {
for(int i=0; i<buyOrderStorehouse.size(); i++){
for(int y=0; y<order.length; y++){
System.out.print(extract(buyOrderStorehouse, i, y) + " ");
}
System.out.println();
};
System.out.println();
}
}
}
```

```
    }  
  }  
  
  /** To get the last executed price. */  
  public double getPrice(){  
    return(executedPrice);  
  }  
  
  /** To get the mean price of yesterday. */  
  public double getMeanPrice(){  
    return(meanPrice);  
  }  
  
  /** To get the number of sell orders filled in the book. */  
  public int getSellOrderNumber(){  
    return(sellOrderStorehouse.size());  
  }  
  
  /** To get the number of shares filled in the sell side of book. */  
  public int getSharesInSellSide(){  
    return(sharesInSellSide);  
  }  
  
  /** To get the number of buy orders filled in the book. */  
  public int getBuyOrderNumber(){  
    return(buyOrderStorehouse.size());  
  }  
  
  /** To get the number of shares filled in the buy side of book. */  
  public int getSharesInBuySide(){  
    return(sharesInBuySide);  
  }  
  
  /** To get the closing price of yesterday. */  
  public double getPreviousClosingPrice(){  
    return(previousClosingPrice);  
  }  
}
```


APPENDICE C

JAVASUM: CODICE DEL MODELLO PIÙ AGGIORNATO (VERSIONE 0.8)

StartJavaSum.java

```
// StartJavaSum.java

import swarm.Globals;
import swarm.defobj.Zone;

/**
 * The StartJavaSum class contains main().
 * We follow here the typical Swarm structure with main() (in Start...
 * as a convention) generating the Observer and the Observer
 * generating the Model.
 *
 * @author Marco Agagliate
 */
public class StartJavaSum
{
    /**
     * The main() function is the top-level place where everything starts.
     */
    public static void main (String[] args)
    {
        /** The observer in our application. */
        ObserverSwarm observerSwarm;

        // Swarm initialization: all Swarm apps must call this first.
        Globals.env.initSwarm ("JavaSum", "0.8",
            "pietro.terna@unito.it", args);

        // Create a top-level Swarm object, observerSwarm, and
        // build its internal objects and activities.

        observerSwarm =
            (ObserverSwarm)Globals.env.lispAppArchiver.getWithZone$key(
                Globals.env.globalZone, "observerSwarm");

        // to save control panel position
        Globals.env.setWindowGeometryRecordName
            (observerSwarm, "observerSwarm");

        // build objets into the observev
        observerSwarm.buildObjects();

        // build actions into the observev
        observerSwarm.buildActions();

        // activate
        observerSwarm.activateIn(null);

        // Now start the displaySwarm and the control panel it
        // provides.
        observerSwarm.go();

        // The user has pressed Quit. Drop everything and return.
        observerSwarm.drop();
    }
}
```

ObserverSwarm.java

```
// ObserverSwarm.java

import swarm.Globals;
```

Appendice C

```
import swarm.Selector;

import swarm.defobj.Zone;
import swarm.defobj.ZoneImpl;
import swarm.defobj.Symbol;

import swarm.simtoolsgui.GUISwarm;
import swarm.simtoolsgui.GUISwarmImpl;

import swarm.activity.ActionGroup;
import swarm.activity.ActionGroupImpl;
import swarm.activity.Schedule;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;
import swarm.objectbase.EmptyProbeMap;
import swarm.objectbase.EmptyProbeMapImpl;

import swarm.collections.ListImpl;

import swarm.analysis.EZGraph;
import swarm.analysis.EZGraphImpl;

/**
 * ObserverSwarm.java The observer swarm is collection of objects that
 * are used to run and observe the ModelSwarm that actually comprises
 * the simulation.
 *
 *
 * @author Marco Agagliate, Antonio de Ruvo
 */
public class ObserverSwarm extends GUISwarmImpl
{
    // Declare the display parameters and their default values.
    /** Update frequency. */
    public int displayFrequency      = 1;
    /** To stop simulation at the end of a day. */
    public int stopAtDayNumber      = 90;
    /** To save on a file the price data. */
    public int savePriceData        = 0;
    /** To create file and represent previous day mean price. */
    public int displayPreviousDayMean = 0;
    /** To show book graph. */
    public int showBookLogGraph     = 0;
    /** To save book data. */
    public int saveBookLogData      = 0;
    /** To show agent's wealth graph. */
    public int showAgentWealthGraph = 0;
    /** To save agent's wealth data. */
    public int saveAgentWealthData  = 0;
    /** To show spread graph between first bid and first ask price. */
    public int showBidAskSpreadGraph = 0;
    /** To save spread data between first bid and first ask price. */
    public int saveBidAskSpreadData  = 0;
    /** To show spread graph between quantities in sell and buy side. */
    public int showBuySellQuantitySpreadGraph = 0;
    /** To save spread data between quantities in sell and buy side. */
    public int saveBuySellQuantitySpreadData = 0;
    /** To show spread graph between the first and the last price in the two side. */
    public int showBuySellFirstLastSpreadGraph = 0;
    /** To save spread data between the first and the last price in the two side. */
    public int saveBuySellFirstLastSpreadData = 0;
    /** To show the number of shares in the two side before opening or closing. */
    public int showBuySellSharesNumberInOpeningOrClosingAuctions = 0;

    // Declare other variables local to ObserverSwarm.
```

```

/** the ModelSwarm we are observing */
public ModelSwarm modelSwarm;
/** the single Schedule instance */
public ScheduleImpl displaySchedule;
/** The book of the simulation. */
public Book theBook;
/** two ActionGroup for sequence of GUI events */
public ActionGroupImpl displayActions;
/** our graphics or EZGraph output to files*/
public EZGraphImpl priceGraph, priceFile, bookLogGraph,
    agentWealth, agentWealthGraph,
    sharesInSellSideFile, sharesInBuySideFile,
    minWealthAllFile, meanWealthAllFile,maxWealthAllFile,
    minWealthRandomFile,meanWealthRandomFile, maxWealthRandomFile,
    minWealthMarketImitatingFile,meanWealthMarketImitatingFile,maxWealthMarketImitatingFile,
    minWealthLocallyImitatingFile,meanWealthLocallyImitatingFile,maxWealthLocallyImitatingFile,
    minWealthStopLossFile,meanWealthStopLossFile,maxWealthStopLossFile,
    meanPriceFile, bidAskSpreadGraph,
    currentBidAskSpreadFile, sharesInBuySideInOpeningOrClosingAuctionsFile,
    sharesInSellSideInOpeningOrClosingAuctionsFile, quantitySpreadGraph,
    currentQuantitySpreadFile, firstLastSpreadGraph,
    currentBuyFirstLastSpreadFile, currentSellFirstLastSpreadFile;

/** Constructor for a new ObserverSwarm. */
public ObserverSwarm(Zone aZone)
{
    // Use the parent class to create an observer swarm.
    super(aZone);

    // Build a custom probe map. Without a probe map, the default
    // is to show all variables and messages. Here we choose to
    // customize the appearance of the probe, giving a nicer
    // interface.

    // Create the probe map and give it the ObserverSwarm class.
    EmptyProbeMapImpl probeMap = new EmptyProbeMapImpl(aZone, getClass());

    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("displayFrequency",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("stopAtDayNumber",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("displayPreviousDayMean",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("savePriceData",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("showBookLogGraph",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("saveBookLogData",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("showAgentWealthGraph",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("saveAgentWealthData",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("showBidAskSpreadGraph",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("saveBidAskSpreadData",
            ObserverSwarm.this.getClass());
    probeMap.addProbe(Globals.env.probeLibrary

```

Appendice C

```
.getProbeForVariable$inClass("showBuySellQuantitySpreadGraph",
    ObserverSwarm.this.getClass());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("saveBuySellQuantitySpreadData",
        ObserverSwarm.this.getClass()));
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("showBuySellFirstLastSpreadGraph",
        ObserverSwarm.this.getClass()));
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("saveBuySellFirstLastSpreadData",
        ObserverSwarm.this.getClass()));
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("showBuySellSharesNumberInOpeningOrClosingAuctions",
        ObserverSwarm.this.getClass()));

// And finally install our probe map into the probeLibrary.
// Note that this library object was automatically created by
// initSwarm.
Globals.env.probeLibrary.setProbeMap$For(probeMap, getClass());
}

/** Create the objects used to display the model. */
public Object buildObjects(){
    // Zone modelZone;
    Selector sel;

    // Use the parent class to initialize the process.
    super.buildObjects();

    // We create the model that we're actually observing, by
    // creating an instance of the ModelSwarm class, modelSwarm.
    modelSwarm =
        (ModelSwarm)Globals.env.lispAppArchiver.getWithZone$key(
            getZone(), "modelSwarm");

    // Now create probe objects on the model and on ourselves.

    Globals.env.createArchivedProbeDisplay(modelSwarm, "modelSwarm");
    Globals.env.createArchivedProbeDisplay(this, "observerSwarm");

    // Instruct the control panel to wait for a button event: we
    // halt here until someone hits a control panel button.
    // Eventually this will allow the user a chance to fill in
    // parameters before the simulation runs.
    getControlPanel().setStateStopped();

    // Allow the model swarm to build its objects.
    modelSwarm.buildObjects();

    // checking the consistence of the displayFrequency with the istantNumber:
    if(displayFrequency !=1 && displayFrequency %
        modelSwarm.istantNumber != 0)
    {
        System.out.println("displayFrequency must be 1 or multiple of istantNumber.\n");
        System.exit(1);
    }

    theBook = modelSwarm.getBook();

    priceGraph = new EZGraphImpl(getZone (), "Current price",
        "Ticks x days.", "Price.", "priceGraph");
    Globals.env.setWindowGeometryRecordName (priceGraph, "priceGraph");

    // the data for the priceGraph
    priceGraph.createSequence$withFeedFrom$andSelector
        ("Current price", theBook,
        SwarmUtils.getSelector(theBook,"getPrice"));
}
```

```

if (displayPreviousDayMean==1){
    priceGraph.createSequence$withFeedFrom$andSelector
        ("Day-1 mean p.", theBook,
        SwarmUtils.getSelector(theBook, "getMeanPrice"));
}

    if (savePriceData==1){
// file
priceFile = new EZGraphImpl(getZone(), "price");
priceFile.createSequence$withFeedFrom$andSelector
    ("txt", theBook, SwarmUtils.getSelector(theBook, "getPrice"));

// file
meanPriceFile = new EZGraphImpl(getZone(), "meanPrice");
meanPriceFile.createSequence$withFeedFrom$andSelector
    ("txt", theBook, SwarmUtils.getSelector(theBook, "getMeanPrice"));
    }

if (showBookLogGraph==1 || showBuySellSharesNumberInOpeningOrClosingAuctions==1){
    bookLogGraph = new EZGraphImpl(getZone(), "Book log", "Tiks x days.",
        "Number of shares logged in buy or sell side.",
        "bookLogGraph");
    Globals.env.setWindowGeometryRecordName (bookLogGraph, "bookLogGraph");

// the data for the bookLogGraph
    if (showBookLogGraph==1){
bookLogGraph.createSequence$withFeedFrom$andSelector
    ("Shares in sell side", theBook,
    SwarmUtils.getSelector(theBook, "getSharesInSellSide"));
bookLogGraph.createSequence$withFeedFrom$andSelector
    ("Shares in buy side", theBook,
    SwarmUtils.getSelector(theBook, "getSharesInBuySide"));
    }
    if (showBuySellSharesNumberInOpeningOrClosingAuctions==1){
bookLogGraph.createSequence$withFeedFrom$andSelector
    ("Shares in sell side before opening or closing", theBook,
    SwarmUtils.getSelector(theBook, "getSharesInSellSideInOpeningOrClosingAuctions"));
bookLogGraph.createSequence$withFeedFrom$andSelector
    ("Shares in buy side before opening or closing", theBook,
    SwarmUtils.getSelector(theBook, "getSharesInBuySideInOpeningOrClosingAuctions"));
    }
    }

    if (saveBookLogData==1){
// file
sharesInSellSideFile = new EZGraphImpl(getZone(), "sharesInSellSide");
sharesInSellSideFile.createSequence$withFeedFrom$andSelector
    ("txt", theBook, SwarmUtils.getSelector(theBook,
    "getSharesInSellSide"));
sharesInBuySideFile = new EZGraphImpl(getZone(), "sharesInBuySide");
sharesInBuySideFile.createSequence$withFeedFrom$andSelector
    ("txt", theBook, SwarmUtils.getSelector(theBook,
    "getSharesInBuySide"));
sharesInSellSideInOpeningOrClosingAuctionsFile = new EZGraphImpl(getZone(),
    "sharesInSellSideInOpeningOrClosingAuctions");
sharesInSellSideInOpeningOrClosingAuctionsFile.createSequence$withFeedFrom$andSelector
    ("txt", theBook, SwarmUtils.getSelector(theBook,
    "getSharesInSellSideInOpeningOrClosingAuctions"));
sharesInBuySideInOpeningOrClosingAuctionsFile = new EZGraphImpl(getZone(),
    "sharesInBuySideInOpeningOrClosingAuctions");
sharesInBuySideInOpeningOrClosingAuctionsFile.createSequence$withFeedFrom$andSelector
    ("txt", theBook, SwarmUtils.getSelector(theBook,
    "getSharesInBuySideInOpeningOrClosingAuctions"));
    }

    if (showBidAskSpreadGraph==1){
        bidAskSpreadGraph = new EZGraphImpl(getZone(), "Current bid-ask spread",
            "Tiks x days.", "Spread", "bidAskSpreadGraph");
    }

```

Appendice C

```
Globals.env.setWindowGeometryRecordName (bidAskSpreadGraph,
    "bidAskSpreadGraph");

    // the data for bidAskSpreadGraph
bidAskSpreadGraph.createSequence$withFeedFrom$andSelector
("Bid-ask spread", theBook,
SwarmUtils.getSelector(theBook, "getSpread"));
}

if (saveBidAskSpreadData==1){
//file
currentBidAskSpreadFile = new EZGraphImpl(getZone(), "bid-askSpread");
currentBidAskSpreadFile.createSequence$withFeedFrom$andSelector("txt",
theBook, SwarmUtils.getSelector(theBook, "getSpread"));
}

if (showBuySellQuantitySpreadGraph==1){
quantitySpreadGraph = new EZGraphImpl(getZone(),
"Current buy and sell quantity spread",
"Ticks x days.", "Spread", "quantitySpreadGraph");
Globals.env.setWindowGeometryRecordName (quantitySpreadGraph,
"quantitySpreadGraph");

// the data for quantitySpreadGraph
quantitySpreadGraph.createSequence$withFeedFrom$andSelector
("Quantity spread", theBook,
SwarmUtils.getSelector(theBook, "getBuySellQuantitySpread"));
}

if (saveBuySellQuantitySpreadData==1){
//file
currentQuantitySpreadFile = new EZGraphImpl(getZone(), "quantitySpread");
currentQuantitySpreadFile.createSequence$withFeedFrom$andSelector("txt",
theBook, SwarmUtils.getSelector(theBook, "getBuySellQuantitySpread"));
}

if (showBuySellFirstLastSpreadGraph==1){
firstLastSpreadGraph = new EZGraphImpl(getZone(),
"Current first-last spread", "Ticks x days.",
"Spread", "firstLastSpreadGraph");
Globals.env.setWindowGeometryRecordName (firstLastSpreadGraph,
"firstLastSpreadGraph");

// the data for firstLastSpreadGraph
firstLastSpreadGraph.createSequence$withFeedFrom$andSelector
("Buy first-last spread", theBook,
SwarmUtils.getSelector(theBook, "getBuyFirstLastSpread"));
firstLastSpreadGraph.createSequence$withFeedFrom$andSelector
("Sell first-last spread", theBook,
SwarmUtils.getSelector(theBook, "getSellFirstLastSpread"));
}

if (saveBuySellFirstLastSpreadData==1){
//file
currentBuyFirstLastSpreadFile = new EZGraphImpl(getZone(),
"buyFirstLastSpread");
currentBuyFirstLastSpreadFile.createSequence$withFeedFrom$andSelector("txt",
theBook, SwarmUtils.getSelector(theBook, "getBuyFirstLastSpread"));
currentSellFirstLastSpreadFile = new EZGraphImpl(getZone(),
"sellFirstLastSpread");
currentSellFirstLastSpreadFile.createSequence$withFeedFrom$andSelector("txt",
theBook, SwarmUtils.getSelector(theBook, "getSellFirstLastSpread"));
}

if (showAgentWealthGraph==1){
agentWealthGraph = new EZGraphImpl(getZone(), "Agent's wealth",
"Ticks x days.", "Wealth", "agentWealthGraph");
Globals.env.setWindowGeometryRecordName (agentWealthGraph,
```

```

"agentWealthGraph");

// the data for the agentGraph
agentWealthGraph.createMinSequence$withFeedFrom$andSelector
  ("Min Wealth (all)", modelSwarm.getAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
  ("Mean Wealth (all)", modelSwarm.getAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
  ("Max Wealth (all)", modelSwarm.getAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMinSequence$withFeedFrom$andSelector
  ("Min Wealth (r.)", modelSwarm.getRandomAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
  ("Mean Wealth (r.)", modelSwarm.getRandomAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
  ("Max Wealth (r.)", modelSwarm.getRandomAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMinSequence$withFeedFrom$andSelector
  ("Min Wealth (m.i)", modelSwarm.getMarketImitatingAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
  ("Mean Wealth (m.i)", modelSwarm.getMarketImitatingAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
  ("Max Wealth (m.i)", modelSwarm.getMarketImitatingAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMinSequence$withFeedFrom$andSelector
  ("Min Wealth (l.i)", modelSwarm.getLocallyImitatingAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
  ("Mean Wealth (l.i)", modelSwarm.getLocallyImitatingAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
  ("Max Wealth (l.i)", modelSwarm.getLocallyImitatingAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMinSequence$withFeedFrom$andSelector
  ("Min Wealth (s.l)", modelSwarm.getStopLossAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createAverageSequence$withFeedFrom$andSelector
  ("Mean Wealth (s.l)", modelSwarm.getStopLossAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
agentWealthGraph.createMaxSequence$withFeedFrom$andSelector
  ("Max Wealth (s.l)", modelSwarm.getStopLossAgentList(),
  SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
  }

if (saveAgentWealthData==1){
  // file
  minWealthAllFile = new EZGraphImpl(getZone(), "minWealthAll");
  minWealthAllFile.createMinSequence$withFeedFrom$andSelector
    ("txt", modelSwarm.getAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
  meanWealthAllFile = new EZGraphImpl(getZone(), "meanWealthAll");
  meanWealthAllFile.createAverageSequence$withFeedFrom$andSelector
    ("txt", modelSwarm.getAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
  maxWealthAllFile = new EZGraphImpl(getZone(), "maxWealthAll");
  maxWealthAllFile.createMaxSequence$withFeedFrom$andSelector
    ("txt", modelSwarm.getAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
  minWealthRandomFile = new EZGraphImpl(getZone(), "minWealthRandom");
  minWealthRandomFile.createMinSequence$withFeedFrom$andSelector
    ("txt", modelSwarm.getRandomAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
}

```

```

meanWealthRandomFile = new EZGraphImpl(getZone(), "meanWealthRandom");
meanWealthRandomFile.createAverageSequence$withFeedFrom$andSelector
("txt", modelSwarm.getRandomAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
maxWealthRandomFile = new EZGraphImpl(getZone(), "maxWealthRandom");
maxWealthRandomFile.createMaxSequence$withFeedFrom$andSelector
("txt", modelSwarm.getRandomAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
minWealthMarketImitatingFile = new EZGraphImpl(getZone(), "minWealthMarketImitating");
minWealthMarketImitatingFile.createMinSequence$withFeedFrom$andSelector
("txt", modelSwarm.getMarketImitatingAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
meanWealthMarketImitatingFile = new EZGraphImpl(getZone(), "meanWealthMarketImitating");
meanWealthMarketImitatingFile.createAverageSequence$withFeedFrom$andSelector
("txt", modelSwarm.getMarketImitatingAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
maxWealthMarketImitatingFile = new EZGraphImpl(getZone(), "maxWealthMarketImitating");
maxWealthMarketImitatingFile.createMaxSequence$withFeedFrom$andSelector
("txt", modelSwarm.getMarketImitatingAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
minWealthLocallyImitatingFile = new EZGraphImpl(getZone(), "minWealthLocallyImitating");
minWealthLocallyImitatingFile.createMinSequence$withFeedFrom$andSelector
("txt", modelSwarm.getLocallyImitatingAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
meanWealthLocallyImitatingFile = new EZGraphImpl(getZone(), "meanWealthLocallyImitating");
meanWealthLocallyImitatingFile.createAverageSequence$withFeedFrom$andSelector
("txt", modelSwarm.getLocallyImitatingAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
maxWealthLocallyImitatingFile = new EZGraphImpl(getZone(), "maxWealthLocallyImitating");
maxWealthLocallyImitatingFile.createMaxSequence$withFeedFrom$andSelector
("txt", modelSwarm.getLocallyImitatingAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
minWealthStopLossFile = new EZGraphImpl(getZone(), "minWealthStopLoss");
minWealthStopLossFile.createMinSequence$withFeedFrom$andSelector
("txt", modelSwarm.getStopLossAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
meanWealthStopLossFile = new EZGraphImpl(getZone(), "meanWealthStopLoss");
meanWealthStopLossFile.createAverageSequence$withFeedFrom$andSelector
("txt", modelSwarm.getStopLossAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
maxWealthStopLossFile = new EZGraphImpl(getZone(), "maxWealthStopLoss");
maxWealthStopLossFile.createMaxSequence$withFeedFrom$andSelector
("txt", modelSwarm.getStopLossAgentList(),
    SwarmUtils.getSelector("BasicSumAgent", "getWealthAtMeanDailyPrice"));
    }
return this;
}

/**
 * Create the actions necessary for the simulation. This is where
 * the schedule is built (but not run!) Here we create a display
 * schedule - this is used to display the state of the world and
 * check for user input.
 */
public Object buildActions()
{
    Selector sel;

    // Use the parent class to begin the process.
    super.buildActions();

    // Call on the model swarm to build and schedule its actions.
    modelSwarm.buildActions();

    // Create an ActionGroup for display. This is a list of
    // display actions that we want to occur at each step in
    // simulation time.
    // "doTkEvents", is required at the end to make everything

```

```

// happen. We then check to see if modelSwarm has
// told us to stop the simulation.
displayActions = new ActionGroupImpl(getZone());

// to update probes
sel = SwarmUtils.getSelector(Globals.env.probeDisplayManager, "update");
displayActions.createActionTo$message(Globals.env.probeDisplayManager, sel);
sel = SwarmUtils.getSelector(getActionCache(), "doTkEvents");
displayActions.createActionTo$message(getActionCache(), sel);

sel = SwarmUtils.getSelector(priceGraph, "step");
displayActions.createActionTo$message(priceGraph, sel);

if (savePriceData==1)
{
    sel = SwarmUtils.getSelector(priceFile, "step");
    displayActions.createActionTo$message(priceFile, sel);

    sel = SwarmUtils.getSelector(meanPriceFile, "step");
    displayActions.createActionTo$message(meanPriceFile, sel);
}

if (showBookLogGraph==1 || showBuySellSharesNumberInOpeningOrClosingAuctions==1 )
{
    sel = SwarmUtils.getSelector(bookLogGraph, "step");
    displayActions.createActionTo$message(bookLogGraph, sel);
}

if (saveBookLogData==1)
{
    sel = SwarmUtils.getSelector(sharesInSellSideFile, "step");
    displayActions.createActionTo$message(sharesInSellSideFile, sel);
    sel = SwarmUtils.getSelector(sharesInBuySideFile, "step");
    displayActions.createActionTo$message(sharesInBuySideFile, sel);
    sel = SwarmUtils.getSelector(sharesInSellSideInOpeningOrClosingAuctionsFile,
        "step");
    displayActions.createActionTo$message(
        sharesInSellSideInOpeningOrClosingAuctionsFile, sel);
    sel = SwarmUtils.getSelector(sharesInBuySideInOpeningOrClosingAuctionsFile,
        "step");
    displayActions.createActionTo$message(
        sharesInBuySideInOpeningOrClosingAuctionsFile, sel);
}

if (showBidAskSpreadGraph==1)
{
    sel = SwarmUtils.getSelector(bidAskSpreadGraph, "step");
    displayActions.createActionTo$message(bidAskSpreadGraph, sel);
}

if (saveBidAskSpreadData==1)
{
    sel = SwarmUtils.getSelector(currentBidAskSpreadFile, "step");
    displayActions.createActionTo$message(currentBidAskSpreadFile, sel);
}

if (showBuySellQuantitySpreadGraph==1)
{
    sel = SwarmUtils.getSelector(quantitySpreadGraph, "step");
    displayActions.createActionTo$message(quantitySpreadGraph, sel);
}

if (saveBuySellQuantitySpreadData==1)
{
    sel = SwarmUtils.getSelector(currentQuantitySpreadFile, "step");
    displayActions.createActionTo$message(currentQuantitySpreadFile, sel);
}

```

Appendice C

```
        if (showBuySellFirstLastSpreadGraph==1)
        {
            sel = SwarmUtils.getSelector(firstLastSpreadGraph, "step");
            displayActions.createActionTo$message(firstLastSpreadGraph, sel);
        }

        if (saveBuySellFirstLastSpreadData==1)
        {
            sel = SwarmUtils.getSelector(currentBuyFirstLastSpreadFile, "step");
            displayActions.createActionTo$message(currentBuyFirstLastSpreadFile, sel);
            sel = SwarmUtils.getSelector(currentSellFirstLastSpreadFile, "step");
            displayActions.createActionTo$message(currentSellFirstLastSpreadFile, sel);
        }

    if (showAgentWealthGraph==1)
    {
        sel = SwarmUtils.getSelector(agentWealthGraph, "step");
        displayActions.createActionTo$message(agentWealthGraph, sel);
    }

    if (saveAgentWealthData==1)
    {
        sel = SwarmUtils.getSelector(minWealthAllFile, "step");
        displayActions.createActionTo$message(minWealthAllFile, sel);
        sel = SwarmUtils.getSelector(meanWealthAllFile, "step");
        displayActions.createActionTo$message(meanWealthAllFile, sel);
        sel = SwarmUtils.getSelector(maxWealthAllFile, "step");
        displayActions.createActionTo$message(maxWealthAllFile, sel);
        sel = SwarmUtils.getSelector(minWealthRandomFile, "step");
        displayActions.createActionTo$message(minWealthRandomFile, sel);
        sel = SwarmUtils.getSelector(meanWealthRandomFile, "step");
        displayActions.createActionTo$message(meanWealthRandomFile, sel);
        sel = SwarmUtils.getSelector(maxWealthRandomFile, "step");
        displayActions.createActionTo$message(maxWealthRandomFile, sel);
        sel = SwarmUtils.getSelector(minWealthMarketImitatingFile, "step");
        displayActions.createActionTo$message(minWealthMarketImitatingFile, sel);
        sel = SwarmUtils.getSelector(meanWealthMarketImitatingFile, "step");
        displayActions.createActionTo$message(meanWealthMarketImitatingFile, sel);
        sel = SwarmUtils.getSelector(maxWealthMarketImitatingFile, "step");
        displayActions.createActionTo$message(maxWealthMarketImitatingFile, sel);
        sel = SwarmUtils.getSelector(minWealthLocallyImitatingFile, "step");
        displayActions.createActionTo$message(minWealthLocallyImitatingFile, sel);
        sel = SwarmUtils.getSelector(meanWealthLocallyImitatingFile, "step");
        displayActions.createActionTo$message(meanWealthLocallyImitatingFile, sel);
        sel = SwarmUtils.getSelector(maxWealthLocallyImitatingFile, "step");
        displayActions.createActionTo$message(maxWealthLocallyImitatingFile, sel);
        sel = SwarmUtils.getSelector(minWealthStopLossFile, "step");
        displayActions.createActionTo$message(minWealthStopLossFile, sel);
        sel = SwarmUtils.getSelector(meanWealthStopLossFile, "step");
        displayActions.createActionTo$message(meanWealthStopLossFile, sel);
        sel = SwarmUtils.getSelector(maxWealthStopLossFile, "step");
        displayActions.createActionTo$message(maxWealthStopLossFile, sel);
    }

    sel = SwarmUtils.getSelector(this, "checkToStop");
    displayActions.createActionTo$message(this, sel);

    // Finally, put the ActionGroup into a display schedule.
    // Note that the repeat interval is set by our
    // own Swarm data structure. The display is frequently the slowest part of a
    // simulation, when it is redraw less frequently things are faster
    // We can use here a display frequency lower (e.g.1) than the number of steps
    // in the model (step number = agentNumber) to display the prices of each
    // step-tick
    displaySchedule = new ScheduleImpl(getZone(), displayFrequency);
    displaySchedule.at$createAction(displayFrequency-1, displayActions);
```

```

        return this;
    }

    /**
     * Activate the schedules so that they are ready to run. The
     * swarmContext argument is the zone in which the ObserverSwarm is
     * activated. Typically the ObserverSwarm is the top-level swarm,
     * so it is activated in "null". The other (sub)swarms and
     * schedules will be activated inside of the ObserverSwarm
     * context.
     */
    public Activity activateIn(Swarm swarmContext)
    {
        // Use the parent class to activate ourselves in the context
        // passed to us.
        super.activateIn(swarmContext);

        // Now activate the model swarm in the ObserverSwarm context.
        modelSwarm.activateIn(this);

        // Then activate the ObserverSwarm schedule in the
        // ObserverSwarm context.
        displaySchedule.activateIn(this);

        // Finally, return the activity we have built - the thing that
        // is ready to run.
        return getActivity();
    }

    /** To check for the stopping conditions. */
    public void checkToStop()
    {
        // if stopAtDayNumber is left to 0, the program will never stop
        // this stop occurs after the first tick of the time in modelSwarm,
        // but this fact does not affect the results in the observerSwarm
        // being the stop performed before the graph update
        if (stopAtDayNumber != 0 && stopAtDayNumber <= modelSwarm.getCurrentDay())
        {
            System.out.println("Stopping at day number" + modelSwarm.getCurrentDay());
            // we can restart by pressing "Start" or "Next", but the simulation
            // will run for displayFrequency ticks and then it will stop again
            getControlPanel().setStateStopped();
        }
    }
}

```

ModelSwarm.java

```

// ModelSwarm.java

import swarm.Globals;
import swarm.Selector;
import swarm.defobj.Zone;

import swarm.objectbase.Swarm;
import swarm.objectbase.SwarmImpl;
import swarm.objectbase.EmptyProbeMap;
import swarm.objectbase.EmptyProbeMapImpl;

import swarm.activity.ActionGroupImpl;
import swarm.activity.ScheduleImpl;
import swarm.activity.Activity;

import swarm.collections.ListImpl;
import swarm.collections.ListIndex;
import swarm.collections.ListShufflerImpl;
import swarm.collections.ArrayImpl;

```

Appendice C

```
/**
 * The Model of JavaSum. The Model contains the Units
 * and all the related tools.
 *
 * @author Marco Agagliate
 *
 */
public class ModelSwarm extends SwarmImpl
{
    // Declare the model parameters and their default values.

    public int randomAgentNumber      = 300;
    public int marketImitatingAgentNumber = 0;
    public int locallyImitatingAgentNumber = 0;
    public int stopLossAgentNumber     = 0;
    /** The total number of agents. */
    public int agentNumber             =
randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber+stopLossAgentNumber;
    /** The total number of istants per day. */
    public int istantNumber             = 300;
    /** The number of current day. */
    public int dayNumber               = 0;
    /** The asimmetry of buy and sell probability. */
    public double asymmetricBuySellProb = 0.9;
    /** The coefficient that RandomAgents multiplies
 * by the last price for the order.
 */
    public double minCorrectingCoeff    = 0.95;
    /** The coefficient that RandomAgents multiplies
 * by the last price for the order.
 */
    public double maxCorrectingCoeff    = 1.05;
    /**The asimmetry of max/minCorrectingCoeff. */
    public double asymmetricRange      = 0.0;
    /** The probability of placing an order with market price. */
    public double agentProbToActWithMarketPrice = 0.0;
    /** The probability of placing an order in the opening or closing phase.
 * So a day starts and finish with an action, with a realistic effect.
 */
    public double agentProbToActInOpeningOrClosingAuctions = 0.5;
    /** The minimum price at which the agents acts. */
    public double floorP               = 0.3;
    /** The probability to act below floor price. */
    public double agentProbToActBelowFloorP = 0.5;
    /** The maximum number of order per agent. */
    public int maxOrderQuantity        = 3;

    /** The max rate of loss that stop loss agents effort. */
    public double maxLossRate          = 0.1;
    /** The interval that stop loss agents consider for their strategy. */
    public int stopLossInterval        = 2;
    /** To check the position(sharequantity) of the agent. */
    public int checkingIfShortOrLong   = 1;

    /** The percentage quota of agents which act in the market.*/
    public int percentageOfOperatingAgents = 0;
    /** The maximum number of operating agents. */
    public int maxNumberOfOperatingAgents = 1;
    /** The type of percentage of agents. If it is true percentage
 * is a maximum value; if it is false percentage is a
 * fixed value.
 */
    public boolean typeOfPercentage    = false;

    /** If 1 book prints data on the terminal window,
 * 2 for Agent. */
    public int printing                = 0;
```

```

// Declare some other needed variables.
/** The list of all the agents. */
public ListImpl agentList;

/** The list of the RandomAgents. */
public ListImpl randomAgentList;
/** The list of the MarketImitatingAgents.*/
public ListImpl marketImitatingAgentList;
/** The list of the LocallyImitatingAgents.*/
public ListImpl locallyImitatingAgentList;
/** The list of the StopLossAgents.*/
public ListImpl stopLossAgentList;

/** The list of the agents in create order. */
public ListImpl indexAgentList;

/** Its iterator. */
public ListIndex indexAgentListIndex;

/** ActionGroup for holding an ordered sequence of action. */
public ActionGroupImpl modelActions1,
                        modelActionsLS,
                        modelActions2,
                        modelActions3,
                        modelActions4;

/** The Schedule operating in the Model. */
public ScheduleImpl modelSchedule;

/** The book of the simulation. */
public Book theBook;
/** The agents of the simulation. */
public RandomAgent    anAgent1;
public MarketImitatingAgent anAgent2;
public LocallyImitatingAgent anAgent3;
public StopLossAgent    anAgent4;

/** The randomRuleMaster manages RandomAgents. */
public RandomRuleMaster randomRuleMaster;
/** The stopLossRuleMaster manages StopLossAgents.*/
public StopLossRuleMaster stopLossRuleMaster;
/** This is a "ghost agent".
 * See the comments of CurrentAgent class.
 */
public CurrentAgent theCurrentAgent;
/** The object for the calls of the agents. */
public CurrentIstant theCurrentIstant;

/** Constructor for a new ModelSwarm. */
public ModelSwarm(Zone aZone)
{
    // Use the parent class to create a top-level swarm.
    super(aZone);

    // Build a customized probe map. Without a custom probe map
    // the default is to show all variables and messages. Here we
    // choose to customize the appearance of the probe, giving a
    // nicer interface.

    // Create the probe map and give it the ModelSwarm class.
    EmptyProbeMapImpl probeMap = new EmptyProbeMapImpl(aZone, getClass());

    // Now add probes for the variables we wish to probe.
    probeMap.addProbe(Globals.env.probeLibrary
        .getProbeForVariable$inClass("randomAgentNumber",
            ModelSwarm.this.getClass ());

```

```

probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("marketImitatingAgentNumber",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("locallyImitatingAgentNumber",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("stopLossAgentNumber",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("agentNumber",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("istantNumber",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("dayNumber",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("asymmetricBuySellProb",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("minCorrectingCoeff",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("maxCorrectingCoeff",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("asymmetricRange",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("agentProbToActWithMarketPrice",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("agentProbToActInOpeningOrClosingAuctions",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("floorP",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("agentProbToActBelowFloorP",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("maxOrderQuantity",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("maxLossRate",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("stopLossInterval",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("checkingIfShortOrLong",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("percentageOfOperatingAgents",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("maxNumberOfOperatingAgents",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("typeOfPercentage",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForVariable$inClass("printing",
        ModelSwarm.this.getClass ());
probeMap.addProbe(Globals.env.probeLibrary
    .getProbeForMessage$inClass("openProbeTo:",

```

```

        ModelSwarm.this.getClass ());

    // And finally install our probe map into the probeLibrary.
    // Note that this library was created by initSwarm().
    Globals.env.probeLibrary.setProbeMap$For(probeMap, getClass());
    }

/** Build the model objects. */
public Object buildObjects()
{
    int i;

    // use the parent class buildObject() method to initialize the
    // process
    super.buildObjects();

    // Now create the model's objects.
    // randomRuleMaster
    randomRuleMaster = new RandomRuleMaster(getZone());
    randomRuleMaster.setAgentProbToActInOpeningOrClosingAuctions
        ( agentProbToActInOpeningOrClosingAuctions );
    randomRuleMaster.setMinCorrectingCoeff( minCorrectingCoeff );
    randomRuleMaster.setMaxCorrectingCoeff( maxCorrectingCoeff );
    randomRuleMaster.setAsymmetricRange( asymmetricRange );
    randomRuleMaster.setFloorP$andAgentProbToActBelowFloorP
        ( floorP, agentProbToActBelowFloorP );

    // stopLossRuleMaster
    stopLossRuleMaster = new StopLossRuleMaster(getZone());
    stopLossRuleMaster.setAgentProbToActInOpeningOrClosingAuctions
        ( agentProbToActInOpeningOrClosingAuctions );
    stopLossRuleMaster.setMinCorrectingCoeff( minCorrectingCoeff );
    stopLossRuleMaster.setMaxCorrectingCoeff( maxCorrectingCoeff );
    stopLossRuleMaster.setAsymmetricRange( asymmetricRange );
    stopLossRuleMaster.setFloorP$andAgentProbToActBelowFloorP
        ( floorP, agentProbToActBelowFloorP );

    // Now create a List object to manage all the agent we are
    // about to create.
    agentList = new ListImpl(getZone());
    randomAgentList = new ListImpl(getZone());
    marketImitatingAgentList = new ListImpl(getZone());
    locallyImitatingAgentList = new ListImpl(getZone());
    stopLossAgentList = new ListImpl(getZone());
    indexAgentList = new ListImpl(getZone());
    indexAgentListIndex = indexAgentList.listBegin(getZone());

    if (agentNumber==0)
    {
        System.out.println("Nonsense: agentNumber cannot be 0");
        System.exit(1);
    }

    // the book
    theBook = new Book(getZone());
    theBook.setAgentListIndex( indexAgentListIndex );
    theBook.setPrinting( printing );
    theBook.setIstantNumber( istantNumber );
    if(printing==1){
        System.out.println("Length of meanPriceHistory: " +theBook.meanPriceHistory.length);
    }

    // a few checks
    if (asymmetricBuySellProb<0.5)
    {
        System.out.println

```

Appendice C

```
        ("The asymmetricBuySellProb cannot be < 0.5 (internally set to 0.5).\n");
        asymmetricBuySellProb=0.5;
    }

// randomAgent
for (i=1;i<=randomAgentNumber;i++)
    {
        anAgent1 = new RandomAgent(getZone(), maxOrderQuantity);
        anAgent1.setNumber( i );
        anAgent1.setMaxOrderQuantity( maxOrderQuantity );
        anAgent1.setAgentProbToActWithMarketPrice( agentProbToActWithMarketPrice );
        anAgent1.setBook( theBook );
        anAgent1.setRuleMaster( randomRuleMaster );
        anAgent1.setPrinting( printing );
        agentList.addLast(anAgent1);
        randomAgentList.addLast(anAgent1);
        indexAgentList.addLast(anAgent1);
    }

// marketImitatingAgent
for (i=randomAgentNumber+1;i<=randomAgentNumber+marketImitatingAgentNumber;i++)
    {
        anAgent2 = new MarketImitatingAgent(getZone(), maxOrderQuantity);
        anAgent2.setNumber( i );
        anAgent2.setMaxOrderQuantity( maxOrderQuantity );
        anAgent2.setAgentProbToActWithMarketPrice( agentProbToActWithMarketPrice );
        anAgent2.setAsymmetricBuySellProb( asymmetricBuySellProb );
        anAgent2.setBook( theBook );
        anAgent2.setRuleMaster( randomRuleMaster );
        anAgent2.setPrinting( printing );
        agentList.addLast(anAgent2);
        marketImitatingAgentList.addLast(anAgent2);
        indexAgentList.addLast(anAgent2);
    }

// locallyImitatingAgent
for
(i=randomAgentNumber+marketImitatingAgentNumber+1;i<=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber;i++)
    {
        anAgent3 = new LocallyImitatingAgent(getZone(), maxOrderQuantity);
        anAgent3.setNumber( i );
        anAgent3.setMaxOrderQuantity( maxOrderQuantity );
        anAgent3.setAgentProbToActWithMarketPrice( agentProbToActWithMarketPrice );
        anAgent3.setAsymmetricBuySellProb( asymmetricBuySellProb );
        anAgent3.setBook( theBook );
        anAgent3.setRuleMaster( randomRuleMaster );
        anAgent3.setPrinting( printing );
        agentList.addLast(anAgent3);
        locallyImitatingAgentList.addLast(anAgent3);
        indexAgentList.addLast(anAgent3);
    }

// stopLossAgent
for
(i=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber+1;i<=randomAgentNumber+marketImitatingAgentNumber+locallyImitatingAgentNumber+stopLossAgentNumber;i++)
    {
        anAgent4 = new StopLossAgent(getZone(), maxOrderQuantity);
        anAgent4.setNumber( i );
        anAgent4.setMaxOrderQuantity( maxOrderQuantity );
        anAgent4.setAgentProbToActWithMarketPrice( agentProbToActWithMarketPrice );
        anAgent4.setStopLossInterval( stopLossInterval );
        anAgent4.setMaxLossRate$andCheckingIfShortOrLong( maxLossRate,checkingIfShortOrLong );
        anAgent4.setBook( theBook );
        anAgent4.setRuleMaster( stopLossRuleMaster );
        anAgent4.setPrinting( printing );
        agentList.addLast(anAgent4);
    }
```

```

        stopLossAgentList.addLast(anAgent4);
        indexAgentList.addLast(anAgent4);
    }

    // current agent
    theCurrentAgent = new CurrentAgent(getZone());
    theCurrentAgent.setAgentList( agentList );
    theCurrentAgent.setPrinting( printing );

    // number of operating agents

    maxNumberOfOperatingAgents
    = percentageOfOperatingAgents*agentNumber/100;
    if (percentageOfOperatingAgents == 0)
        maxNumberOfOperatingAgents = 1;

    // current istant
    theCurrentIstant = new CurrentIstant(getZone());
    theCurrentIstant.setCurrentAgent( theCurrentAgent );
    theCurrentIstant.setMaxNumberOfOperatingAgents
    ( maxNumberOfOperatingAgents );
    theCurrentIstant.setPercentageOfOperatingAgents
    ( percentageOfOperatingAgents );
    theCurrentIstant.setTypeOfPercentage( typeOfPercentage );
    theCurrentIstant.setPrinting( printing );
    return this;
}

/**
 * Here is where the model schedule is built, the data structures
 * that define the simulation of time in the model. The core is an
 * actionGroup that has a list of actions. Then that's put in a Schedule.
 */
public Object buildActions()
{
    Selector sel;

    int i, j;

    ListShufflerImpl listShuffler = new ListShufflerImpl(getZone());

    // First, use the parent class to initialize the process.
    super.buildActions();

    // We create the list of simulation actions

    //Open auction
    modelActions1 = new ActionGroupImpl(getZone());
    sel = SwarmUtils.getSelector(this, "setAgentNumber");
    modelActions1.createActionTo$message(this, sel);
    sel = SwarmUtils.getSelector("Book", "setClean");
    modelActions1.createActionTo$message(theBook, sel);
    sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
    modelActions1.createActionTo$message(listShuffler, sel, agentList);
    sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
    modelActions1.createActionForEach$message(agentList, sel);
    sel = SwarmUtils.getSelector("Book", "setTheoreticalPrice");
    modelActions1.createActionTo$message(theBook, sel);
    sel = SwarmUtils.getSelector("Book", "setOpeningPrice");
    modelActions1.createActionTo$message(theBook, sel);
    sel = SwarmUtils.getSelector("Book", "opening");
    modelActions1.createActionTo$message(theBook, sel);

    //Shuffle list
    modelActionsLS = new ActionGroupImpl(getZone());
    sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
    modelActionsLS.createActionTo$message(listShuffler, sel, agentList);

```

```

//Acting in the market
modelActions2 = new ActionGroupImpl(getZone());
sel = SwarmUtils.getSelector("CurrentIstant", "callAgents");
modelActions2.createActionTo$message(theCurrentIstant, sel);
sel = SwarmUtils.getSelector("Book", "setNumberOfIstant");
modelActions2.createActionTo$message(theBook, sel);

//Close auction
modelActions3 = new ActionGroupImpl(getZone());
sel = SwarmUtils.getSelector("Book", "recreateTableForTheoreticalPrice");
modelActions3.createActionTo$message(theBook, sel);
sel = SwarmUtils.getSelector(listShuffler, "shuffleWholeList");
modelActions3.createActionTo$message(listShuffler, sel, agentList);
sel = SwarmUtils.getSelector("BasicSumAgent", "act0");
modelActions3.createActionForEach$message(agentList, sel);
sel = SwarmUtils.getSelector("Book", "setTheoreticalPrice");
modelActions3.createActionTo$message(theBook, sel);
sel = SwarmUtils.getSelector("Book", "setOpeningPrice");
modelActions3.createActionTo$message(theBook, sel);
sel = SwarmUtils.getSelector("Book", "opening");
modelActions3.createActionTo$message(theBook, sel);

//Accounting
modelActions4 = new ActionGroupImpl(getZone());
sel = SwarmUtils.getSelector("Book", "setMeanPrice");
modelActions4.createActionTo$message(theBook, sel);
sel = SwarmUtils.getSelector("BasicSumAgent", "act2");
modelActions4.createActionForEach$message(agentList, sel);
sel = SwarmUtils.getSelector(this, "increaseCurrentDayNumber");
modelActions4.createActionTo$message(this, sel);

// Then we create a schedule that executes the modelActions.
// We use here agentNumber steps in each cycle, while
// the observer uses a low display frequency (e.g. 1)
// to show the price of each step-tick we can also use a high d.f.
// (e.g. 1000 with 100 agents) to run a faster simulation.

modelSchedule = new ScheduleImpl(getZone(), istantNumber);
modelSchedule.at$createAction(0, modelActions1);

for (i=0;i<istantNumber;i++)
{
    modelSchedule.at$createAction(i, modelActionsLS);
    modelSchedule.at$createAction(i, modelActions2);
}
modelSchedule.at$createAction(istantNumber-1, modelActions3);
modelSchedule.at$createAction(istantNumber-1, modelActions4);

return this;
}

/**
 * Now set up the model's activation. swarmContext indicates where
 * we're being started in - typically, this model is run as a
 * subswarm of an observer swarm. */
public Activity activateIn(Swarm swarmContext)
{
    // Use the parent class to activate ourselves in the context
    // passed to us.
    super.activateIn(swarmContext);

    // Then activate the schedule in ourselves.
    modelSchedule.activateIn(this);

    // Finally, return the activity we have built.
    return getActivity();
}

```

```

/** The method increases the number of the day. */
public void increaseCurrentDayNumber()
{
    dayNumber++;
    if(printing==1)
        System.out.println("Day number " + dayNumber);
}

/** The method set the number of the agents. */
public void setAgentNumber()
{
    agentNumber= randomAgentNumber+marketImitatingAgentNumber+
        locallyImitatingAgentNumber+stopLossAgentNumber;
}

/** The method returns the number of the day. */
public int getCurrentDay()
{
    return dayNumber;
}

/** The method returns the list of the agents. */
public ListImpl getAgentList()
{
    return agentList;
}

/** The method returns the list of the RandomAgents. */
public ListImpl getRandomAgentList()
{
    return randomAgentList;
}

/** The method returns the list of the MarketImitatingAgents. */
public ListImpl getMarketImitatingAgentList()
{
    return marketImitatingAgentList;
}

/** The method returns the list of the LocallyImitatingAgents. */
public ListImpl getLocallyImitatingAgentList()
{
    return locallyImitatingAgentList;
}

/** The method returns the list of the StopLossAgents. */
public ListImpl getStopLossAgentList()
{
    return stopLossAgentList;
}

/** The method returns the index of indexAgentList. */
public ListIndex getAgentListIndex()
{
    return indexAgentListIndex;
}

/** The method returns the book. */
public Book getBook()
{
    return theBook;
}

/** The method opens the probe on an agent. */
public void openProbeTo(int n)

```

Appendice C

```
    {
        BasicSumAgent anAgent;

        if (n>=1 && n<=agentNumber)
        {
            indexAgentListIndex.setOffset(n-1);
            anAgent = (BasicSumAgent)indexAgentListIndex.get();
            Globals.env.createArchivedProbeDisplay(anAgent, "anAgent");
        }
    }
}
```

Book.java

```
// Book.java

import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;
import java.util.*;
import java.math.BigDecimal;

import swarm.collections.*;

/**
 * This is the book.
 * The book works on the basis of two arrayList containing sell order in
 * increasing order or buy order in decreasing order. The orders are arrays which
 * have in the first position the price, in the second the number of agent who places
 * the order and in the third the quantity.
 * If an order obtains an immediate matching, it is not filed
 *
 * @author Antonio de Ruvo e-mail:antderu@libero.it
 * @author Integrations of Bruno Mencarelli
 */

public class Book extends SwarmObjectImpl{
    /** The index used for send message to agents. */
    public ListIndex indexAgentListIndex;
    /** If 1 many objects print data on the terminal window. */
    public int printing;
    /** The total number of istant. */
    public int istantNumber;
    /** The number of actual istant. */
    public int numberOfIstant;
    /** The number of orders received from agents. */
    public int orderNumber;
    /** The number of share negotiated */
    public int count=0;
    /** The number of shares in buy side of book. */
    public int sharesInBuySide;
    /** The number of shares in sell side of book. */
    public int sharesInSellSide;
    /** The number of shares in buy side of book in opening or closing Auctions. */
    public int sharesInBuySideInOpeningOrClosingAuctions;
    /** The number of shares in sell side of book in opening or closing Auctions. */
    public int sharesInSellSideInOpeningOrClosingAuctions;
    /** The price of the last contract closing. */
    public double executedPrice=1;
    /** The mean price of yesterday. */
    public double meanPrice=1;
    /** The closing price of yesterday. */
    public double previousClosingPrice=executedPrice;
    /** The addition of prices*quantity to calculate the mean price. */
    public double currentMeanPrice=0;
    /** The arrayList which contains the buy orders. */
    public ArrayList buyOrderStorehouse=new ArrayList();
    /** The arrayList which contains the sell orders. */
    public ArrayList sellOrderStorehouse=new ArrayList();
}
```

```

/** The arrayList to log the closing contracts. */
public ArrayList closingContractLog=new ArrayList();
/** The arrayList as a table to evaluate the theoretic price. */
public ArrayList tableForTheoreticalPrice=new ArrayList();
/** The arrayList which contains the buy orders with auction price. */
public ArrayList buyOrderWithAuctionPriceStorehouse=new ArrayList();
/** The arrayList which contains the sell orders with auction price. */
public ArrayList sellOrderWithAuctionPriceStorehouse=new ArrayList();
/** The reference price of each days. */
public double referencePrice=1;
/** The theoretical price. */
public double theoreticalPrice=0;
/** The opening price. */
public double openingPrice=1;
/** The number of buy order with auction price. */
public int nOfBuyOrderWithAuctionPrice;
/** The number of sell order with auction price. */
public int nOfSellOrderWithAuctionPrice;
/** The number of shares with auction price in buy side. */
public int nOfBuySharesWithAuctionPrice;
/** The number of shares with auction price in sell side. */
public int nOfSellSharesWithAuctionPrice;

/** The length of the mean price history. */
public int meanPriceHistoryLength=1000;
/** The length of the local price history. */
public int localHistoryLength=1000;
/** The vector containing the history of mean prices. */
public double[] meanPriceHistory = new double[meanPriceHistoryLength];
/** The vector containing the local price history. */
public double[] localHistory = new double[localHistoryLength];

/** Constructor for a new book. */
public Book(Zone aZone){
    super(aZone);
}

/** To inform the agent of executed price.
 * @see BasicSumAgent
 */
private void message(double n, double p, double q){
    BasicSumAgent anAgent;
    indexAgentListIndex.setOffset((int)n-1);
    anAgent = (BasicSumAgent)indexAgentListIndex.get();
    double[]confirmationOfExecutedPrice={p,q};
    anAgent.setConfirmationOfExecutedPrice(confirmationOfExecutedPrice);
}

/** To standardize the order. */
private double[] standardizedOrder(double[] orderFromAgent){
    double [] order=new double [5];
    if(orderFromAgent.length==2){
        double[] orderStandard={orderFromAgent[0], orderFromAgent[1], 1, 0, 0};
        order=orderStandard;
    }
    if(orderFromAgent.length==3){
        double [] orderStandard={ orderFromAgent[0], orderFromAgent[1],
            orderFromAgent[2], 0, 0};
        order=orderStandard;
    }
    if(orderFromAgent.length==4){
        double [] orderStandard={ orderFromAgent[0], orderFromAgent[1],
            orderFromAgent[2], orderFromAgent[3], 0};
        order=orderStandard;
    }
    return(order);
}

```

Appendice C

```
/** To extract data from the orders received from agents. */
private double extract( ArrayList nameList, int indexOrder, int positionDatum ){
    Object[] array=nameList.toArray();
    double datum=((double []) array[indexOrder])[positionDatum];
    return(datum);
}

/** To fill sellOrderStorehouse in increasing order */
private int fillStorehouseIncreasingP(double newPrice){
    int index=0;
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        if(newPrice<extract(sellOrderStorehouse, i, 0)){
            index=i;
            break;
        }
    }
    return(index);
}

/** To fill buyOrderStorehouse in decreasing order. */
private int fillStorehouseDecreasingP (double newPrice){
    int index=0;
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        if(newPrice>extract(buyOrderStorehouse, i, 0)){
            index=i;
            break;
        }
    }
    return(index);
}

/** This is the SetLocal method, used to shift the rows of the localHistory vector.
 * @author Mencarelli
 */
public void setLocal( double[] vect,double p ){
    int i;
    if(p==0) return;
    for(i=vect.length-2;i>=0;i--){
        vect[i+1]=vect[i];
    }
    vect[0]=p;
}

/** This is the getLocal method, that counts for the difference
 * between the buying and selling orders sent by the agents.
 * @author Mencarelli
 */
public int getLocal( double[] vect ){
    int i,tot=0;
    for(i=0;i<vect.length;i++){
        if(vect[i]>0) tot++;
        if(vect[i]<0) tot--;
    }
    return tot;
}

/** To set the value of option printing. */
public void setPrinting(int p){
    printing=p;
}

/** To set the value of total number of istant */
public void setIstantNumber(int n){
    istantNumber=n;
}

/** To know the adress of memory of the agent. */
public void setAgentListIndex(ListIndex i){
    indexAgentListIndex=i;
}
```

```

    }

    /** To set the value of actual istant */
    public void setNumberOfIstant(){
        numberOfIstant++;
    }

    /** At the end of each day to calculate the mean price. */
    public void setMeanPrice(){
        if(count>0)
            meanPrice=currentMeanPrice/count;
        // otherwise we keep previous value
    }

    /** At the end of each day to calculate the reference price. */
    public void setReferencePrice(){
        if(previousClosingPrice!=0)
            referencePrice=previousClosingPrice;
        if(previousClosingPrice==0){
            // The price's weighed average of last 10% quantity
            double quantity=count*0.1;
            double q=0;
            double weighedAverage=0;
            Object[] matrix=closingContractLog.toArray();
            int index=matrix.length;
            while(q<quantity){
                index--;
                weighedAverage+=(((double[])matrix[index])[0]*((double[])matrix[index])[1]);
                q+=((double[])matrix[index])[0];
            }
            weighedAverage=(q-quantity)*((double[])matrix[index])[1];
            referencePrice=weighedAverage/quantity;
            if(printing==1){
                System.out.println("The log of closing contrancts is:");
                for(int i=0; i<closingContractLog.size(); i++){
                    for(int y=0; y<2; y++){
                        System.out.print(extract(closingContractLog, i, y) + " ");
                    }
                    System.out.println();
                }
                System.out.println();
            }
            System.out.println("count = " + count );
            System.out.println("The quantity of last 10% is " + quantity );
            System.out.println("weighedAverage = " + weighedAverage );
            System.out.println("ReferencePrice = " + referencePrice);
        }
    }
}

/** At the beginning of each day to clean the book. */
public void setClean(){
    sellOrderStorehouse.clear();
    buyOrderStorehouse.clear();
    closingContractLog.clear();
    tableForTheoreticalPrice.clear();
    buyOrderWithAuctionPriceStorehouse.clear();
    sellOrderWithAuctionPriceStorehouse.clear();
    previousClosingPrice=openingPrice; // the closing price of "yesterday"
    orderNumber=0;
    for(int i=meanPriceHistoryLength-2;i>=0;i--){
        meanPriceHistory[i+1]=meanPriceHistory[i];
    }
    meanPriceHistory[0]=meanPrice;
    currentMeanPrice=0;
    count=0;
    sharesInSellSide=0;
    sharesInBuySide=0;
    nOfBuyOrderWithAuctionPrice=0;
}

```

Appendice C

```
nOfSellOrderWithAuctionPrice=0;
nOfBuySharesWithAuctionPrice=0;
nOfSellSharesWithAuctionPrice=0;
numberOfIstant=0;
if(printing==1)
    System.out.println("Book: clean");
}

/** To set rows in tableForTheoreticalPrice. */
private double[] setRows(double[] order){
    double[] row=new double[5];
    if(order[3]==0){
        if(order[0]>0){
            row[0]=order[0];
            row[1]=order[2];
            row[3]=Math.min(row[1],row[2]);
            row[4]=Math.abs(row[1]-row[2]);
        }
        else if(order[0]<0){
            row[0]=-order[0];
            row[2]=order[2];
            row[3]=Math.min(row[1],row[2]);
            row[4]=Math.abs(row[1]-row[2]);
        }
    }
    if(order[3]==1){
        row[0]=0;
        row[1]=order[2];
        row[3]=Math.min(row[1],row[2]);
        row[4]=Math.abs(row[1]-row[2]);
    }
    else if(order[3]==2){
        row[0]=0;
        row[2]=order[2];
        row[3]=Math.min(row[1],row[2]);
        row[4]=Math.abs(row[1]-row[2]);
    }
    return(row);
}

/** To set the prices in decreasing order for evaluate the theoretical price. */
private void setTableForTheoreticalPrice(double[] order){
    double[] row;
    row=setRows(order);
    Object[] matrix=tableForTheoreticalPrice.toArray();

    if(tableForTheoreticalPrice.size()==0){
        tableForTheoreticalPrice.add(row);
        if(row[1]>0 && row[0]==0)
            nOfBuySharesWithAuctionPrice+=row[1];
        if(row[2]>0 && row[0]==0)
            nOfSellSharesWithAuctionPrice+=row[2];
    }

    else if(row[0]==0){
        if(row[1]>0)
            nOfBuySharesWithAuctionPrice+=row[1];
        if(row[2]>0)
            nOfSellSharesWithAuctionPrice+=row[2];
        for(int i=0; i<matrix.length; i++){
            ((double[])matrix[i])[1]+=row[1];
            ((double[])matrix[i])[2]+=row[2];
            ((double[])matrix[i])[3]=Math.min(((double[])matrix[i])[1], ((double[])matrix[i])[2]);
            ((double[])matrix[i])[4]=Math.abs(((double[])matrix[i])[1]-((double[])matrix[i])[2]);
            tableForTheoreticalPrice.set(i, matrix[i]);
        }
    }
}
```

```

else if(row[0]>0){
    boolean thereIs=false;
    for(int i=0; i<matrix.length; i++){
        if(row[0]==((double[])matrix[i])[0]){
            thereIs=true;
        }
    }
    if(thereIs==false){
        if(row[0]>((double[])matrix[0])[0] && ((double[])matrix[0])[0]!=0){
            double[] newRow=new double[5];
            newRow[0]=row[0];
            newRow[1]=nOfBuySharesWithAuctionPrice;
            newRow[2]=((double[])matrix[0])[2];
            tableForTheoreticalPrice.add(0, newRow);
        }
        if(row[0]>((double[])matrix[0])[0] && ((double[])matrix[0])[0]==0){
            ((double[])matrix[0])[0]=row[0];
            tableForTheoreticalPrice.set(0, matrix[0]);
        }
        else if(row[0]<((double[])matrix[0])[0]){
            if(row[0]<((double[])matrix[matrix.length-1])[0]){
                double[] newRow=new double[5];
                newRow[0]=row[0];
                newRow[1]=((double[])matrix[0])[1];
                newRow[2]=nOfSellSharesWithAuctionPrice;
                tableForTheoreticalPrice.add(newRow);
            }
            else{
                for(int i=0; i<matrix.length; i++){
                    if(row[0]>((double[])matrix[i])[0]){
                        double[] newRow=new double[5];
                        newRow[0]=row[0];
                        newRow[1]=((double[])matrix[i-1])[1];
                        newRow[2]=((double[])matrix[i])[2];
                        tableForTheoreticalPrice.add(i, newRow);
                        break;
                    }
                }
            }
        }
    }

    matrix=tableForTheoreticalPrice.toArray();
    if(row[1]>0){
        for(int i=0; i<matrix.length; i++){
            if(row[0]>=((double[])matrix[i])[0]){
                ((double[])matrix[i])[1]+=row[1];
                ((double[])matrix[i])[2]+=row[2];
                ((double[])matrix[i])[3]=Math.min(((double[])matrix[i])[1],
                    ((double[])matrix[i])[2]);
                ((double[])matrix[i])[4]=Math.abs(((double[])matrix[i])[1]-
                    ((double[])matrix[i])[2]);
                tableForTheoreticalPrice.set(i, matrix[i]);
            }
        }
    }

    else if(row[2]>0){
        for(int i=0; i<matrix.length; i++){
            if(row[0]<=((double[])matrix[i])[0]){
                ((double[])matrix[i])[1]+=row[1];
                ((double[])matrix[i])[2]+=row[2];
                ((double[])matrix[i])[3]=Math.min(((double[])matrix[i])[1],
                    ((double[])matrix[i])[2]);
                ((double[])matrix[i])[4]=Math.abs(((double[])matrix[i])[1]-
                    ((double[])matrix[i])[2]);
                tableForTheoreticalPrice.set(i, matrix[i]);
            }
        }
    }

```

Appendice C

```
    }
  }
}

/** To set the theoretical price. */
public void setTheoreticalPrice(){
  int numberOfEqualMaxNegotiableQuantity=0;
  int numberOfEqualMinDifferenceBetweenVolume=0;
  double maxQuantityNegotiable=0;
  double minDifferenceBetweenVolume=0;
  int index=0;
  int index1=0;
  int index2=0;
  if(printing==1){
    System.out.println();
    System.out.println("The table for theoretical price is:");
    for(int i=0; i<tableForTheoreticalPrice.size(); i++){
      for(int y=0; y<5; y++){
        System.out.print(extract(tableForTheoreticalPrice, i, y) + " ");
      }
      System.out.println();
    };
    System.out.println();
    System.out.println("The reference price is: " + referencePrice);
    System.out.println();
  }
  Object[] matrix=tableForTheoreticalPrice.toArray();
  //first rule
  for(int i=0;i<matrix.length;i++){
    if(((double[])matrix[i])[3]>maxQuantityNegotiable){
      numberOfEqualMaxNegotiableQuantity=1;
      maxQuantityNegotiable=((double[])matrix[i])[3];
      index=i;
    }
    else if(((double[])matrix[i])[3]==maxQuantityNegotiable)
      numberOfEqualMaxNegotiableQuantity++;
  }
  if(matrix.length==0 || maxQuantityNegotiable==0 )
    theoreticalPrice=0;
  else if(numberOfEqualMaxNegotiableQuantity==1)
    theoreticalPrice=((double[])matrix[index])[0];
  //second rule
  else if(numberOfEqualMaxNegotiableQuantity>1){
    index1=index;
    minDifferenceBetweenVolume=((double[])matrix[index])[4];
    numberOfEqualMinDifferenceBetweenVolume=1;
    for(int i=index+1;i<index+numberOfEqualMaxNegotiableQuantity;i++){
      if(((double[])matrix[i])[4]<minDifferenceBetweenVolume){
        numberOfEqualMinDifferenceBetweenVolume=1;
        minDifferenceBetweenVolume=((double[])matrix[i])[4];
        index1=i;
      }
    }
    else if(((double[])matrix[i])[4]==minDifferenceBetweenVolume)
      numberOfEqualMinDifferenceBetweenVolume++;
  }
  if(numberOfEqualMinDifferenceBetweenVolume==1)
    theoreticalPrice=((double[])matrix[index1])[0];
  //third rule
  if(numberOfEqualMinDifferenceBetweenVolume>1){
    double[] array=new double[numberOfEqualMinDifferenceBetweenVolume];
    for(int i=0;i<array.length;i++){
      array[i]=Math.abs(((double[])matrix[i+index1])[0]-referencePrice);
    }
    double min=array[0];
    int numberOfEqualMin=1;
    index2=index1;
    for(int i=0;i<array.length-1;i++){
```

```

        if(array[i]>array[i+1]){
            numberOfEqualMin=1;
            min=array[i+1];
            index2=index1+i+1;
        }
        if(array[i]<array[i+1]){
            numberOfEqualMin=1;
            min=array[i];
            index2=index1+i;
        }
        else if(array[i]==array[i+1])
            numberOfEqualMin=2;
    }
    if(numberOfEqualMin==1){
        if(((double[])matrix[index2])[0]<referencePrice)
            theoreticalPrice=referencePrice-min;
        else
            theoreticalPrice=referencePrice+min;
    }
    //fourth rule
    else if(numberOfEqualMin==2)
        theoreticalPrice=referencePrice+min;
}
}
if(buyOrderStorehouse.size()==0 && buyOrderWithAuctionPriceStorehouse.size()!=0 &&
sellOrderStorehouse.size()==0 && sellOrderWithAuctionPriceStorehouse.size()!=0){
    theoreticalPrice=referencePrice;
}
if(printing==1){
    System.out.println("The theoretical price is: " + theoreticalPrice);
    System.out.println();
}
}

/** To set the opening price. */
public void setOpeningPrice(){
    if(theoreticalPrice==0)
        openingPrice=0;
    else if(theoreticalPrice>0){
        //validation
        if(theoreticalPrice<=referencePrice*1.1 && theoreticalPrice>=referencePrice*0.9)
            openingPrice=theoreticalPrice;
        else
            openingPrice=0;
    }
    if(printing==1){
        if(numberOfIstant==0)
            System.out.println("The opening price is: " + openingPrice);
        else if(numberOfIstant==istantNumber)
            System.out.println("The closing price is: " + openingPrice);
        System.out.println();
        System.out.println("Buy orders with auction Price: ");
        for(int i =0; i<buyOrderWithAuctionPriceStorehouse.size(); i++){
            for(int y=0; y<5; y++){
                System.out.print(extract(buyOrderWithAuctionPriceStorehouse, i, y) + " ");
            }
            System.out.println();
        };
        System.out.println();
        System.out.println("Sell orders with auction Price: ");
        for(int i =0; i<sellOrderWithAuctionPriceStorehouse.size(); i++){
            for(int y=0; y<5; y++){
                System.out.print(extract(sellOrderWithAuctionPriceStorehouse, i, y) + " ");
            }
            System.out.println();
        };
        System.out.println();
    }
}
}

```

Appendice C

```
}

/** To conclude contracts (if opening price > 0) or to fill the order with auction price. */
public void opening(){
    Object[] matrix=buyOrderWithAuctionPriceStorehouse.toArray();
    Object[] matrix1=sellOrderWithAuctionPriceStorehouse.toArray();
    if(openingPrice!=0){
        referencePrice=openingPrice;
        for(int i=0;i<matrix.length;i++){
            ((double[])matrix[i])[0]=openingPrice;
            ((double[])matrix[i])[3]=0;
        }
        for(int i=0;i<matrix1.length;i++){
            ((double[])matrix1[i])[0]=openingPrice;
            ((double[])matrix1[i])[3]=0;
        }
    }
    //If the auction failed, there is not an other auction
    else if(openingPrice==0 && theoreticalPrice>0 &&
        (theoreticalPrice>referencePrice*1.1 || theoreticalPrice<referencePrice*0.9)){
        if(buyOrderStorehouse.size()!=0 && sellOrderStorehouse.size()!=0){
            for(int i=0;i<matrix.length;i++){
                ((double[])matrix[i])[0]=extract(buyOrderStorehouse,0,0);
                ((double[])matrix[i])[3]=0;
            }
            for(int i=0;i<matrix1.length;i++){
                ((double[])matrix1[i])[0]=extract(sellOrderStorehouse,0,0);
                ((double[])matrix1[i])[3]=0;
            }
        }
        else if(buyOrderStorehouse.size()!=0 && sellOrderStorehouse.size()==0){
            for(int i=0;i<matrix.length;i++){
                ((double[])matrix[i])[0]=extract(buyOrderStorehouse,0,0);
                ((double[])matrix[i])[3]=0;
            }
            for(int i=0;i<matrix1.length;i++){
                ((double[])matrix1[i])[0]=referencePrice;
                ((double[])matrix1[i])[3]=0;
            }
        }
        if(buyOrderStorehouse.size()==0 && sellOrderStorehouse.size()!=0){
            for(int i=0;i<matrix.length;i++){
                ((double[])matrix[i])[0]=referencePrice;
                ((double[])matrix[i])[3]=0;
            }
            for(int i=0;i<matrix1.length;i++){
                ((double[])matrix1[i])[0]=extract(sellOrderStorehouse,0,0);
                ((double[])matrix1[i])[3]=0;
            }
        }
    }
    //If one side is empty or there are only orders with auction price
    else if(openingPrice==0 && buyOrderStorehouse.size()==0 && matrix.length!=0 &&
        sellOrderStorehouse.size()==0 && matrix1.length==0){
        for(int i=0;i<matrix.length;i++){
            ((double[])matrix[i])[0]=referencePrice;
            ((double[])matrix[i])[3]=0;
        }
    }
    else if(openingPrice==0 && buyOrderStorehouse.size()==0 && matrix.length==0 &&
        sellOrderStorehouse.size()==0 && matrix1.length!=0){
        for(int i=0;i<matrix1.length;i++){
            ((double[])matrix1[i])[0]=referencePrice;
            ((double[])matrix1[i])[3]=0;
        }
    }
    else if(openingPrice==0 && buyOrderStorehouse.size()!=0 && matrix.length!=0 &&
        sellOrderStorehouse.size()==0 && matrix1.length==0){
```

```

for(int i=0;i<matrix.length;i++){
    ((double[])matrix[i])[0]=extract(buyOrderStorehouse,0,0);
    ((double[])matrix[i])[3]=0;
}
}
else if(openingPrice==0 && buyOrderStorehouse.size()==0 && matrix.length==0 &&
    sellOrderStorehouse.size()!=0 && matrix1.length!=0){
for(int i=0;i<matrix1.length;i++){
    ((double[])matrix1[i])[0]=extract(sellOrderStorehouse,0,0);
    ((double[])matrix1[i])[3]=0;
}
}
if(printing==1){
System.out.println("Correct buy orders with auction Price: ");
for(int i =0; i<buyOrderWithAuctionPriceStorehouse.size(); i++){
    for(int y=0; y<5; y++){
        System.out.print(extract(buyOrderWithAuctionPriceStorehouse, i, y) + " ");
    }
    System.out.println();
};
System.out.println();
System.out.println("Correct sell orders with auction Price: ");
for(int i =0; i<sellOrderWithAuctionPriceStorehouse.size(); i++){
    for(int y=0; y<5; y++){
        System.out.print(extract(sellOrderWithAuctionPriceStorehouse, i, y) + " ");
    }
    System.out.println();
};
System.out.println();
}
//To fill the orders in buy side
double price=0;
if(matrix.length!=0)
    price=((double[])matrix[0])[0];
Object[] matrix2=buyOrderStorehouse.toArray();
int index=0;
for(int i=0; i<matrix2.length; i++){
    if(price>=((double[])matrix2[i])[0]){
        index=i;
        break;
    }
}
if(matrix2.length==0){
for(int i=0;i<matrix.length;i++){
    buyOrderStorehouse.add((double[])matrix[i]);
}
}
else if(matrix2.length!=0){
if(index==0 && price<((double[])matrix2[matrix2.length-1])[0]){
for(int i=0;i<matrix.length;i++){
    buyOrderStorehouse.add((double[])matrix[i]);
}
}
else if(((double[])matrix2[index])[0]==price){
for(int i=0;i<matrix.length;i++){
    if(((double[])matrix[i])[4]<extract(buyOrderStorehouse,index,4)){
        buyOrderStorehouse.add(index,(double[])matrix[i]);
        index++;
    }
    else if(((double[])matrix[i])[4]>extract(buyOrderStorehouse,index,4)){
        buyOrderStorehouse.add(index+1,(double[])matrix[i]);
        index++;
    }
}
}
else if(matrix2.length!=0 && ((double[])matrix2[index])[0]>price){
for(int i=0;i<matrix.length;i++){
    buyOrderStorehouse.add(index,(double[])matrix[i]);
}
}
}

```

Appendice C

```
        index++;
    }
}
else if(matrix2.length!=0 && ((double[])matrix2[index])[0]<price){
    for(int i=0;i<matrix.length;i++){
        buyOrderStorehouse.add(index,(double[])matrix[i]);
        index++;
    }
}
}
}
if(printing==1) {
    System.out.println();
    System.out.println("All buy side: ");
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        for(int y=0; y<5; y++){
            System.out.print(extract(buyOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
//To fill the orders in sell side
if(matrix1.length!=0)
    price=((double[])matrix1[0])[0];
Object[] matrix3=sellOrderStorehouse.toArray();
int index1=0;
for(int i=0; i<matrix3.length; i++){
    if(price<=((double[])matrix3[i])[0]){
        index1=i;
        break;
    }
}
if(matrix3.length==0){
    for(int i=0;i<matrix1.length;i++){
        sellOrderStorehouse.add((double[])matrix1[i]);
    }
}
else if(matrix3.length!=0){
    if(index1==0 && price>((double[])matrix3[matrix3.length-1])[0]){
        for(int i=0;i<matrix1.length;i++){
            sellOrderStorehouse.add((double[])matrix1[i]);
        }
    }
    else if(((double[])matrix3[index1])[0]==price){
        for(int i=0;i<matrix1.length;i++){
            if(((double[])matrix1[i])[4]<extract(sellOrderStorehouse,index1,4)){
                sellOrderStorehouse.add(index1,(double[])matrix1[i]);
                index1++;
            }
            else if(((double[])matrix1[i])[4]>extract(sellOrderStorehouse,index1,4)){
                sellOrderStorehouse.add(index1+1,(double[])matrix1[i]);
                index1++;
            }
        }
    }
}
else if(matrix3.length!=0 && ((double[])matrix3[index1])[0]<price){
    for(int i=0;i<matrix1.length;i++){
        sellOrderStorehouse.add(index1,(double[])matrix1[i]);
        index1++;
    }
}
else if(matrix3.length!=0 && ((double[])matrix3[index1])[0]>price){
    for(int i=0;i<matrix1.length;i++){
        sellOrderStorehouse.add(index1,(double[])matrix1[i]);
        index1++;
    }
}
}
```

```

if(printing==1) {
System.out.println();
System.out.println("All sell side: ");
for(int i=0; i<sellOrderStorehouse.size(); i++){
for(int y=0; y<5; y++){
System.out.print(extract(sellOrderStorehouse, i, y) + " ");
}
System.out.println();
};
System.out.println();
}

//To close the contracts
if(openingPrice!=0){
while(buyOrderStorehouse.size()!=0 && sellOrderStorehouse.size()!=0 &&
extract(buyOrderStorehouse, 0, 0)>=openingPrice &&
extract(sellOrderStorehouse, 0, 0)<=openingPrice){
if(extract(sellOrderStorehouse, 0, 2)==extract(buyOrderStorehouse, 0, 2)){
count+=extract(buyOrderStorehouse, 0, 2);
executedPrice=openingPrice;
message(extract(sellOrderStorehouse, 0, 1), -executedPrice,
extract(sellOrderStorehouse, 0, 2));
message(extract(buyOrderStorehouse, 0, 1), executedPrice,
extract(buyOrderStorehouse, 0, 2));
double[] log=new double[2];
log[0]=extract(buyOrderStorehouse, 0, 2);
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*extract(buyOrderStorehouse, 0, 2);
sharesInBuySide-=extract(buyOrderStorehouse, 0, 2);
sharesInSellSide-=extract(sellOrderStorehouse, 0, 2);
buyOrderStorehouse.remove(0);
sellOrderStorehouse.remove(0);
if(printing==1) {
System.out.println("Buy order storehouse:");
for(int i=0; i<buyOrderStorehouse.size(); i++){
for(int y=0; y<5; y++){
System.out.print(extract(buyOrderStorehouse, i, y) + " ");
}
System.out.println();
};
System.out.println();
System.out.println("Sell order storehouse:");
for(int i=0; i<sellOrderStorehouse.size(); i++){
for(int y=0; y<5; y++){
System.out.print(extract(sellOrderStorehouse, i, y) + " ");
}
System.out.println();
};
System.out.println();
}
}
else if(extract(sellOrderStorehouse, 0, 2)<extract(buyOrderStorehouse, 0, 2)){
count+=extract(sellOrderStorehouse, 0, 2);
executedPrice=openingPrice;
message(extract(sellOrderStorehouse, 0, 1), -executedPrice,
extract(sellOrderStorehouse, 0, 2));
message(extract(buyOrderStorehouse, 0, 1), executedPrice,
extract(sellOrderStorehouse, 0, 2));
double[] log=new double[2];
log[0]=extract(sellOrderStorehouse, 0, 2);
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*extract(sellOrderStorehouse, 0, 2);
((double[]) buyOrderStorehouse.get(0))[2]=extract(buyOrderStorehouse, 0, 2)
-extract(sellOrderStorehouse, 0, 2);
sharesInBuySide-=extract(sellOrderStorehouse, 0, 2);
sharesInSellSide+=extract(sellOrderStorehouse, 0, 2);
}
}

```

Appendice C

```
sellOrderStorehouse.remove(0);
if(printing==1) {
    System.out.println("Buy order storehouse:");
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        for(int y=0; y<5; y++){
            System.out.print(extract(buyOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
    System.out.println("Sell order storehouse:");
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        for(int y=0; y<5; y++){
            System.out.print(extract(sellOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
else if(extract(sellOrderStorehouse, 0, 2)>extract(buyOrderStorehouse, 0, 2)){
count+=extract(buyOrderStorehouse, 0, 2);
executedPrice=openingPrice;
message(extract(sellOrderStorehouse, 0, 1), -executedPrice,
        extract(buyOrderStorehouse, 0, 2));
message(extract(buyOrderStorehouse, 0, 1), executedPrice,
        extract(buyOrderStorehouse, 0, 2));
double[] log=new double[2];
log[0]=extract(buyOrderStorehouse, 0, 2);
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*extract(buyOrderStorehouse, 0, 2);
((double[]) sellOrderStorehouse.get(0))[2]=
    ((double[]) sellOrderStorehouse.get(0))[2]-extract(buyOrderStorehouse, 0, 2);
sharesInBuySide-=extract(buyOrderStorehouse, 0, 2);
sharesInSellSide+=extract(buyOrderStorehouse, 0, 2);
buyOrderStorehouse.remove(0);
if(printing==1) {
    System.out.println("Buy order storehouse:");
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        for(int y=0; y<5; y++){
            System.out.print(extract(buyOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
    System.out.println("Sell order storehouse:");
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        for(int y=0; y<5; y++){
            System.out.print(extract(sellOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
}
}

/** To recreate the table for theoretical price. */
public void recreateTableForTheoreticalPrice(){
    tableForTheoreticalPrice.clear();
    buyOrderWithAuctionPriceStorehouse.clear();
    sellOrderWithAuctionPriceStorehouse.clear();
    nOfBuyOrderWithAuctionPrice=0;
    nOfSellOrderWithAuctionPrice=0;
    nOfBuySharesWithAuctionPrice=0;
}
```

```

nOfSellSharesWithAuctionPrice=0;
Object[] matrix=sellOrderStorehouse.toArray();
Object[] matrix1=buyOrderStorehouse.toArray();
for(int i=0; i<matrix.length; i++){
  ((double[])matrix[i])[0]=-(double[])matrix[i][0];
  setTableForTheoreticalPrice(((double[])matrix[i]));
  ((double[])matrix[i])[0]=-(double[])matrix[i][0];
}
for(int i=0; i<matrix1.length; i++){
  setTableForTheoreticalPrice(((double[])matrix1[i]));
}
if(printing==1){
  System.out.println();
  System.out.println("The table for theoretical price is:");
  for(int i=0; i<tableForTheoreticalPrice.size(); i++){
    for(int y=0; y<5; y++){
      System.out.print(extract(tableForTheoreticalPrice, i, y) + " ");
    }
    System.out.println();
  };
  System.out.println();
}
}

/** Receiving an order in opening or closing Auctions from an agent. */
public void setOrderInOpeningOrClosingAuctionsFromAgent(double[] orderFromAgent) {
  double [] order = standardizedOrder(orderFromAgent);
  orderNumber++;
  order[4]=orderNumber;
  order[0] = new BigDecimal(order[0]).setScale(3, BigDecimal.ROUND_HALF_DOWN).doubleValue();

  /* local History. */
  setLocal(localHistory,order[0]);

  if(printing==1){
    if(numberOfIstant==0){
      System.out.println("The book received a before opening order from agent " + order[1] +
        " at price " + order[0] + " for " + order[2]+ " share(s), with option = " + order[3] +
        " number's order " + order[4]);
    }
    else if(numberOfIstant==istantNumber){
      System.out.println("The book received a before closing order from agent " + order[1] +
        " at price " + order[0] + " for " + order[2]+ " share(s), with option = " + order[3] +
        " number's order " + order[4]);
    }
  }

  /* if price=0 and it is not an order with auction price or quantity==0 no action required,
  * but sending a message to the agent with price=0 */
  if(order[0]==0 && order[3]==0 || order[2]==0)
    message(order[1], 0.0, 0.0);

  // the agent is selling at minimum price if price<0
  else if(order[0]<0 && order[3]==0) {
    message(order[1], 0.0, 0.0);
    setTableForTheoreticalPrice(order);
    order[0]=order[0];
  }

  /* to insert the order in the book in increasing order.
  * With add() the arrayList updates the index */
  if(sellOrderStorehouse.size()==0 ||
    order[0]>=extract(sellOrderStorehouse, (sellOrderStorehouse.size()-1), 0))
    sellOrderStorehouse.add(order);
  else
    sellOrderStorehouse.add(fillStorehouseIncreasingP(order[0]), order);
  sharesInSellSide+=order[2];
  sharesInSellSideInOpeningOrClosingAuctions=sharesInSellSide;
  if(printing==1) {
    System.out.println("Sell order storehouse:");
  }
}

```

Appendice C

```
        for(int i =0; i<sellOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(sellOrderStorehouse, i, y) + " ");
            }
            System.out.println();
        };
        System.out.println();
    }
}

// the agent is buying at the maximum price if price>0
else if(order[0]>0 && order[3]==0) {
    message(order[1], 0.0, 0.0);
    setTableForTheoreticalPrice(order);
    /* to insert the order in the book in decreasing order.
    * With add()the arrayList updates the index */
    if(buyOrderStorehouse.size()==0 ||
        order[0]<=extract(buyOrderStorehouse, (buyOrderStorehouse.size()-1), 0))
        buyOrderStorehouse.add(order);
    else
        buyOrderStorehouse.add(fillStorehouseDecreasingP(order[0],order));
    sharesInBuySide+=order[2];
    sharesInBuySideInOpeningOrClosingAuctions=sharesInBuySide;
    if(printing==1) {
        System.out.println("Buy order storehouse:");
        for(int i=0; i<buyOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(buyOrderStorehouse, i, y) + " ");
            }
            System.out.println();
        };
        System.out.println();
    }
}

// the agent is buying at the auction price
else if(order[3]==1 && order[0]==0){
    nOfBuyOrderWithAuctionPrice++;
    message(order[1], 0.0, 0.0);
    setTableForTheoreticalPrice(order);
    buyOrderWithAuctionPriceStorehouse.add(order);
    sharesInBuySide+=order[2];
    sharesInBuySideInOpeningOrClosingAuctions=sharesInBuySide;
}

// the agent is selling at the auction price
else if(order[3]==2 && order[0]==0){
    nOfSellOrderWithAuctionPrice++;
    message(order[1], 0.0, 0.0);
    setTableForTheoreticalPrice(order);
    sellOrderWithAuctionPriceStorehouse.add(order);
    sharesInSellSide+=order[2];
    sharesInSellSideInOpeningOrClosingAuctions=sharesInSellSide;
}
}

/** Receiving an order when the market is open. */
public void setOrderFromAgent(double[] orderFromAgent) {
    double [] order = standardizedOrder(orderFromAgent);
    orderNumber++;
    order[4]=orderNumber;
    order[0] = new BigDecimal(order[0]).setScale(3, BigDecimal.ROUND_HALF_DOWN).doubleValue();

    /* local History. */
    if(order[0]!=0)
        setLocal(localHistory,order[0]);

    if(printing==1)
```

```

System.out.println("The book received an order from agent " + order[1] +
" at price " + order[0] + " for " + order[2]+ " share(s), with option = " + order[3]+
" number's order " + order[4]);

/* if price==0 or quantity==0 no action required, but sending a message to the agent
 * with price=0 */
if(order[0]==0 && order[3]==0 || order[2]==0)
    message(order[1], 0.0, 0.0);

// the agent is selling at minimum price if price<0
else if(order[0]<0 && order[3]==0) {
    /* if there is a compatible order, this one is removed and the order received
    * it's not inserted in the book. It's not necessary to shift rows because
    * with remove() the arrayList updates the index */
    while(buyOrderStorehouse.size()-1 >= order[2] && extract(buyOrderStorehouse, 0, 0)>=order[0]
    && order[2]!=0) {
        if(order[2]==extract(buyOrderStorehouse, 0, 2)){
            count+=order[2];
            executedPrice=extract(buyOrderStorehouse, 0, 0);
            message(order[1], -executedPrice, order[2]);
            message(extract(buyOrderStorehouse, 0, 1), executedPrice, order[2]);
            double[] log=new double[2];
            log[0]=order[2];
            log[1]=executedPrice;
            closingContractLog.add(log);
            currentMeanPrice+=executedPrice*order[2];
            buyOrderStorehouse.remove(0);
            sharesInBuySide-=order[2];
            order[2]=0;
            if(printing==1) {
                System.out.println("Buy order storehouse:");
                for(int i=0; i<buyOrderStorehouse.size(); i++){
                    for(int y=0; y<order.length; y++){
                        System.out.print(extract(buyOrderStorehouse, i, y) + " ");
                    }
                    System.out.println();
                };
                System.out.println();
            }
            else if(order[2]<extract(buyOrderStorehouse, 0, 2)){
                count+=order[2];
                executedPrice=extract(buyOrderStorehouse, 0, 0);
                message(order[1], -executedPrice, order[2]);
                message(extract(buyOrderStorehouse, 0, 1), executedPrice, order[2]);
                double[] log=new double[2];
                log[0]=order[2];
                log[1]=executedPrice;
                closingContractLog.add(log);
                currentMeanPrice+=executedPrice*order[2];
                ((double[]) buyOrderStorehouse.get(0))[2]=extract(buyOrderStorehouse, 0, 2)
                -order[2];
                sharesInBuySide-=order[2];
                order[2]=0;
            if(printing==1) {
                System.out.println("Buy order storehouse:");
                for(int i=0; i<buyOrderStorehouse.size(); i++){
                    for(int y=0; y<order.length; y++){
                        System.out.print(extract(buyOrderStorehouse, i, y) + " ");
                    }
                    System.out.println();
                };
                System.out.println();
            }
            else if(order[2]>extract(buyOrderStorehouse, 0, 2)){
                count+=extract(buyOrderStorehouse, 0, 2);
                executedPrice=extract(buyOrderStorehouse, 0, 0);
            }
        }
    }
}

```

Appendice C

```
        message(order[1], -executedPrice, extract(buyOrderStorehouse, 0, 2));
        message(extract(buyOrderStorehouse, 0, 1), executedPrice,
        extract(buyOrderStorehouse, 0, 2));
        double[] log=new double[2];
        log[0]=extract(buyOrderStorehouse, 0, 2);
        log[1]=executedPrice;
        closingContractLog.add(log);
        currentMeanPrice+=executedPrice*extract(buyOrderStorehouse, 0, 2);
        order[2]=order[2]-extract(buyOrderStorehouse, 0, 2);
        sharesInBuySide-=extract(buyOrderStorehouse, 0, 2);
        buyOrderStorehouse.remove(0);
        if(printing==1) {
            System.out.println("Buy order storehouse:");
            for(int i=0; i<buyOrderStorehouse.size(); i++){
                for(int y=0; y<order.length; y++){
                    System.out.print(extract(buyOrderStorehouse, i, y) + " ");
                }
                System.out.println();
            };
            System.out.println();
        }
    }
    if(order[2]!=0) {
        message(order[1], 0.0, 0.0);
        order[0]=order[0];
        /* to insert the order in the book in increasing order.
        * With add( )the arrayList updates the index */
        if(sellOrderStorehouse.size()==0 ||
        order[0]>=extract(sellOrderStorehouse, (sellOrderStorehouse.size()-1), 0))
            sellOrderStorehouse.add(order);
        else
            sellOrderStorehouse.add(fillStorehouseIncreasingP(order[0]), order);
        sharesInSellSide+=order[2];
        if(printing==1) {
            System.out.println("Sell order storehouse:");
            for(int i =0; i<sellOrderStorehouse.size(); i++){
                for(int y=0; y<order.length; y++){
                    System.out.print(extract(sellOrderStorehouse, i, y) + " ");
                }
                System.out.println();
            };
            System.out.println();
        }
    }
}

// the agent is buying at the maximum price if price>0
else if(order[0]>0 && order[3]==0){
    /* if there is a compatible order, this one is removed and the order received
    * it's inserted in the book. It's not necessary shift rows because
    * with remove() the arrayList updates the index */
    while(sellOrderStorehouse.size()>0 && extract(sellOrderStorehouse, 0, 0)<=order[0]
    && order[2]!=0) {
        if(order[2]==extract(sellOrderStorehouse, 0, 2)){
            count+=order[2];
            executedPrice=extract(sellOrderStorehouse, 0, 0);
            message(order[1], executedPrice, order[2]);
            message(extract(sellOrderStorehouse, 0, 1), -executedPrice, order[2]);
            double[] log=new double[2];
            log[0]=order[2];
            log[1]=executedPrice;
            closingContractLog.add(log);
            currentMeanPrice+=executedPrice*order[2];
            sellOrderStorehouse.remove(0);
            sharesInSellSide-=order[2];
            order[2]=0;
            if(printing==1) {
```

```

System.out.println("Sell order storehouse:");
for(int i=0; i<sellOrderStorehouse.size(); i++){
    for(int y=0; y<order.length; y++){
        System.out.print(extract(sellOrderStorehouse, i, y) + " ");
    }
    System.out.println();
};
System.out.println();
}
}
else if(order[2]<extract(sellOrderStorehouse, 0, 2)){
count+=order[2];
executedPrice=extract(sellOrderStorehouse, 0, 0);
message(order[1], executedPrice, order[2]);
message(extract(sellOrderStorehouse, 0, 1), -executedPrice, order[2]);
double[] log=new double[2];
log[0]=order[2];
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*order[2];
((double[]) sellOrderStorehouse.get(0))[2]=extract(sellOrderStorehouse, 0, 2)
-order[2];
sharesInSellSide-=order[2];
order[2]=0;
if(printing==1) {
System.out.println("Sell order storehouse:");
for(int i=0; i<sellOrderStorehouse.size(); i++){
    for(int y=0; y<order.length; y++){
        System.out.print(extract(sellOrderStorehouse, i, y) + " ");
    }
    System.out.println();
};
System.out.println();
}
}
else if(order[2]>extract(sellOrderStorehouse, 0, 2)){
count+=extract(sellOrderStorehouse, 0, 2);
executedPrice=extract(sellOrderStorehouse, 0, 0);
message(order[1], executedPrice, extract(sellOrderStorehouse, 0, 2));
message(extract(sellOrderStorehouse, 0, 1), -executedPrice,
extract(sellOrderStorehouse, 0, 2));
double[] log=new double[2];
log[0]=extract(sellOrderStorehouse, 0, 2);
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*extract(sellOrderStorehouse, 0, 2);
order[2]=order[2]-extract(sellOrderStorehouse, 0, 2);
sharesInSellSide-=extract(sellOrderStorehouse, 0, 2);
sellOrderStorehouse.remove(0);
if(printing==1) {
System.out.println("Sell order storehouse:");
for(int i=0; i<sellOrderStorehouse.size(); i++){
    for(int y=0; y<order.length; y++){
        System.out.print(extract(sellOrderStorehouse, i, y) + " ");
    }
    System.out.println();
};
System.out.println();
}
}
}
}
if(order[2]!=0) {
    message(order[1], 0.0, 0.0);
}
/* to insert the order in the book in decreasing order.
* With add()the arrayList updates the index */
if(buyOrderStorehouse.size()==0 ||
order[0]<=extract(buyOrderStorehouse, (buyOrderStorehouse.size()-1),0))
buyOrderStorehouse.add(order);

```

Appendice C

```
        else
            buyOrderStorehouse.add(fillStorehouseDecreasingP(order[0],order);
sharesInBuySide+=order[2];
if(printing==1) {
    System.out.println("Buy order storhouse:");
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(buyOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
}

// If the agent is selling at the market price
else if(order[0]==0 && order[3]==2) {
    if(buyOrderStorehouse.size()==0)
        message(order[1], 0, 0);
    else{
        while(buyOrderStorehouse.size(>0 && order[2]!=0) {
            if(order[2]==extract(buyOrderStorehouse, 0, 2)){
                count+=order[2];
                executedPrice=extract(buyOrderStorehouse, 0, 0);
                message(order[1], -executedPrice, order[2]);
                message(extract(buyOrderStorehouse, 0, 1), executedPrice, order[2]);
                double[] log=new double[2];
                log[0]=order[2];
                log[1]=executedPrice;
                closingContractLog.add(log);
                currentMeanPrice+=executedPrice*order[2];
                buyOrderStorehouse.remove(0);
                sharesInBuySide-=order[2];
                order[2]=0;
            }
            if(printing==1) {
                System.out.println("Buy order storehouse:");
                for(int i=0; i<buyOrderStorehouse.size(); i++){
                    for(int y=0; y<order.length; y++){
                        System.out.print(extract(buyOrderStorehouse, i, y) + " ");
                    }
                    System.out.println();
                };
                System.out.println();
            }
            }
            else if(order[2]<extract(buyOrderStorehouse, 0, 2)){
                count+=order[2];
                executedPrice=extract(buyOrderStorehouse, 0, 0);
                message(order[1], -executedPrice, order[2]);
                message(extract(buyOrderStorehouse, 0, 1), executedPrice, order[2]);
                double[] log=new double[2];
                log[0]=order[2];
                log[1]=executedPrice;
                closingContractLog.add(log);
                currentMeanPrice+=executedPrice*order[2];
                ((double[]) buyOrderStorehouse.get(0))[2]=extract(buyOrderStorehouse, 0, 2)
                    -order[2];
                sharesInBuySide-=order[2];
                order[2]=0;
            }
            if(printing==1) {
                System.out.println("Buy order storehouse:");
                for(int i=0; i<buyOrderStorehouse.size(); i++){
                    for(int y=0; y<order.length; y++){
                        System.out.print(extract(buyOrderStorehouse, i, y) + " ");
                    }
                    System.out.println();
                };
            }
        }
    }
}
```

```

        System.out.println();
    }
    }
    else if(order[2]>extract(buyOrderStorehouse, 0, 2)){
count+=extract(buyOrderStorehouse, 0, 2);
executedPrice=extract(buyOrderStorehouse, 0, 0);
    message(order[1], -executedPrice, extract(buyOrderStorehouse, 0, 2));
    message(extract(buyOrderStorehouse, 0, 1), executedPrice,
extract(buyOrderStorehouse, 0, 2));
    double[] log=new double[2];
    log[0]=extract(buyOrderStorehouse, 0, 2);
    log[1]=executedPrice;
    closingContractLog.add(log);
    currentMeanPrice+=executedPrice*extract(buyOrderStorehouse, 0, 2);
    order[2]=order[2]-extract(buyOrderStorehouse, 0, 2);
sharesInBuySide-=extract(buyOrderStorehouse, 0, 2);
buyOrderStorehouse.remove(0);
if(printing==1) {
    System.out.println("Buy order storehouse:");
    for(int i=0; i<buyOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(buyOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
}
}
}

// the agent is buying at the market price
else if(order[0]==0 && order[3]==1){
if(sellOrderStorehouse.size()==0)
    message(order[1], 0, 0);
else{
while(sellOrderStorehouse.size()>0 && order[2]!=0) {
if(order[2]==extract(sellOrderStorehouse, 0, 2)){
count+=order[2];
executedPrice=extract(sellOrderStorehouse, 0, 0);
message(order[1], executedPrice, order[2]);
message(extract(sellOrderStorehouse, 0, 1), -executedPrice, order[2]);
double[] log=new double[2];
log[0]=order[2];
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*order[2];
sellOrderStorehouse.remove(0);
sharesInSellSide-=order[2];
order[2]=0;
if(printing==1) {
    System.out.println("Sell order storehouse:");
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(sellOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
}
else if(order[2]<extract(sellOrderStorehouse, 0, 2)){
count+=order[2];
executedPrice=extract(sellOrderStorehouse, 0, 0);
message(order[1], executedPrice, order[2]);
message(extract(sellOrderStorehouse, 0, 1), -executedPrice, order[2]);
double[] log=new double[2];

```

Appendice C

```
log[0]=order[2];
log[1]=executedPrice;
closingContractLog.add(log);
currentMeanPrice+=executedPrice*order[2];
((double[]) sellOrderStorehouse.get(0))[2]=extract(sellOrderStorehouse, 0, 2)
    -order[2];
sharesInSellSide-=order[2];
order[2]=0;
if(printing==1) {
    System.out.println("Sell order storehouse:");
    for(int i=0; i<sellOrderStorehouse.size(); i++){
        for(int y=0; y<order.length; y++){
            System.out.print(extract(sellOrderStorehouse, i, y) + " ");
        }
        System.out.println();
    };
    System.out.println();
}
}
else if(order[2]>extract(sellOrderStorehouse, 0, 2)){
    count+=extract(sellOrderStorehouse, 0, 2);
    executedPrice=extract(sellOrderStorehouse, 0, 0);
    message(order[1], executedPrice, extract(sellOrderStorehouse, 0, 2));
    message(extract(sellOrderStorehouse, 0, 1), -executedPrice,
        extract(sellOrderStorehouse, 0, 2));
    double[] log=new double[2];
    log[0]=extract(sellOrderStorehouse, 0, 2);
    log[1]=executedPrice;
    closingContractLog.add(log);
    currentMeanPrice+=executedPrice*extract(sellOrderStorehouse, 0, 2);
    order[2]=order[2]-extract(sellOrderStorehouse, 0, 2);
    sharesInSellSide-=extract(sellOrderStorehouse, 0, 2);
    sellOrderStorehouse.remove(0);
    if(printing==1) {
        System.out.println("Sell order storehouse:");
        for(int i=0; i<sellOrderStorehouse.size(); i++){
            for(int y=0; y<order.length; y++){
                System.out.print(extract(sellOrderStorehouse, i, y) + " ");
            }
            System.out.println();
        };
        System.out.println();
    }
}
}
}
}

/** To get the last executed price. */
public double getPrice(){
    return(executedPrice);
}

/** To get the mean price of yesterday. */
public double getMeanPrice(){
    return(meanPrice);
}

/** To get the number of shares filled in the sell side of book. */
public int getSharesInSellSide(){
    return(sharesInSellSide);
}

/** To get the number of shares filled in the buy side of book. */
public int getSharesInBuySide(){
    return(sharesInBuySide);
}
```

```
/** To get the closing price of yesterday. */
public double getPreviousClosingPrice(){
    return(previousClosingPrice);
}

/** To get the spread between the first and the last price in the sell side. */
public double getSellFirstLastSpread(){
    double spread=0;
    if(sellOrderStorehouse.size()==0)
        spread=0;
    else
        spread=extract(sellOrderStorehouse, 0, 0)-
            extract(sellOrderStorehouse, sellOrderStorehouse.size()-1, 0);
    return(spread);
}

/** To get the spread between the first and the last price in the buy side. */
public double getBuyFirstLastSpread(){
    double spread=0;
    if(buyOrderStorehouse.size()==0)
        spread=0;
    else
        spread=extract(buyOrderStorehouse, 0, 0)-
            extract(buyOrderStorehouse, buyOrderStorehouse.size()-1, 0);
    return(spread);
}

/** To get the spread between first bid and first ask price. */
public double getSpread(){
    double spread=0;
    if(sellOrderStorehouse.size()==0 && buyOrderStorehouse.size(>0)
        spread=0-extract(buyOrderStorehouse, 0, 0);
    else if(sellOrderStorehouse.size()==0 && buyOrderStorehouse.size()==0)
        spread=0;
    else if(sellOrderStorehouse.size(>0 && buyOrderStorehouse.size()==0)
        spread=extract(sellOrderStorehouse, 0, 0)-0;
    else
        spread=extract(sellOrderStorehouse, 0, 0)-extract(buyOrderStorehouse, 0, 0);
    return(spread);
}

/** To get the spread between quantities in sell and buy side. */
public int getBuySellQuantitySpread(){
    return(sharesInBuySide-sharesInSellSide);
}

/** To get the number of shares in sell side in opening or closing Auctions. */
public int getSharesInSellSideInOpeningOrClosingAuctions(){
    return(sharesInSellSideInOpeningOrClosingAuctions);
}

/** To get the number of shares in buy side in opening or closing Auctions. */
public int getSharesInBuySideInOpeningOrClosingAuctions(){
    return(sharesInBuySideInOpeningOrClosingAuctions);
}

/** to get the lagged Mean Price. */
public double getLaggedMeanPrice(int lag){
    return meanPriceHistory[lag-1];
}

/** to get the local History. */
public int getLocalHistory(){
    return getLocal(localHistory);
}
}
```

Appendice C

BasicSumAgent.java

```
// BasicSumAgent.java

import swarm.Globals;
import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

import swarm.objectbase.SwarmImpl;

import swarm.objectbase.EmptyProbeMap;
import swarm.objectbase.EmptyProbeMapImpl;

import java.util.*;

/**
 * This is the basic class of agents, that is inherited from all the others.
 * It contains the common characteristics of the agents.
 *
 *
 * @author Marco Agagliate, Antonio de Ruvo
 *
 */
public class BasicSumAgent extends SwarmObjectImpl
{
    /** This is the number of the agent. */
    public int number;
    /** The quantity of orders sending out by the agent.
     * There is one share per order:
     * the quantity of the order is the number of orders.
     */
    public int iMax;
    /** This is the probability of placing an order with market price. */
    public double agentProbToActWithMarketPrice;
    /** If printing=1 many objects print data on the terminal
     * window; If printing=2 BasicSumAgents print data on
     * the terminal window.
     */

    public int printing;
    /** The price of the order. */
    public double price;
    /** The asymmetry of buy and sell probability. */
    public double asymmetricBuySellProb;
    /** The distribution of buy and sell order (fixed to 0.5). */
    public double buySellSwitch;
    /** The total agent's quantity of shares*/
    public double shareQuantity;
    /** The maximum number of order per agent. */
    public int maxOrderQuantity;
    /** The value of the shares at mean daily price. */
    public double shareValueAtMeanDailyPrice;
    /** The liquidity of the agent. */
    public double liquidityQuantity;
    /** The agent's wealth at mean daily price. */
    public double agentWealthAtMeanDailyPrice;
    /** The average price of the satisfied order. */
    public double meanOperatingPrice;
    /** The matrix of orders' prices. */
    public ArrayList executedPrices;
    /** The book of the model. */
    public Book theBook;

    /** Constructor for a new BasicSumAgent. */
    public BasicSumAgent(Zone aZone, int maxOrderQuantity)
    {
        super(aZone);
        executedPrices = new ArrayList();
    }
}
```

```

        theBook = new Book(getZone());
        shareQuantity=0;
        shareValueAtMeanDailyPrice=0;
        liquidityQuantity=0;
        agentWealthAtMeanDailyPrice=0;
    }

    /** To extract data from the orders received from agents. */
    public double extract( ArrayList nameList, int indexOrder, int positionDatum ){
        double a=((double []) nameList.get(indexOrder))[positionDatum];
        return(a);
    }

    /** This method sets the number of the agent. */
    public void setNumber(int n)
    {
        number = n;
    }

    /** This method sets the probability of placing an order with market price. . */
    public void setAgentProbToActWithMarketPrice(double p)
    {
        agentProbToActWithMarketPrice=p;
    }

    /** This method sets the asymmetricBuySellProb. */
    public void setAsymmetricBuySellProb(double p)
    {
        asymmetricBuySellProb=p;
    }

    /** This method sets the book. */
    public void setBook(Book b)
    {
        theBook=b;
    }

    /** This method sets the maximum number of order per agent. */
    public void setMaxOrderQuantity(int m)
    {
        maxOrderQuantity=m;
    }

    /** Option about the agent print capability. */
    public void setPrinting(int p)
    {
        printing=p;
    }

    /** The number of executed prices is increased */
    public void setConfirmationOfExecutedPrice(double[] p)
    {
        if(p[0]!=0 || p[1]!=0)
            executedPrices.add(p);
    }

    /**
     * This method exists only for the CurrentAgent.
     * Its implementation is defined into specify agent class.
     */
    public void act0()
    {
    }

    /**
     * This method exists only for the CurrentAgent.
     * Its implementation is defined into specify agent class.
     */

```

Appendice C

```
public void act1()
{
}

/**
 * This is the method that the agent calls at the end of the day.
 * The agent makes daily accounting.
 */
public void act2()
{
    double mP, q, TodayNumberOfShares;
    int i;

    if (printing==2)
        for(i=0;i<executedPrices.size();i++)
            System.out.println("I'm agent " + number
                + " and the book told me: "
                + extract(executedPrices, i, 0)+ " for "
                + extract(executedPrices, i, 1)+ " share(s).");
    mP=theBook.getMeanPrice();
    q=0;
    for (i=0;i<executedPrices.size();i++)
        {
            if (extract(executedPrices, i, 0) > 0)
                q+=extract(executedPrices, i, 1);
            if (extract(executedPrices, i, 0) < 0)
                q-=extract(executedPrices, i, 1);
        }
    shareQuantity += q;
    shareValueAtMeanDailyPrice=shareQuantity*mP;
    meanOperatingPrice=0;
    for (i=0;i<executedPrices.size();i++)
        {
            liquidityQuantity -= extract(executedPrices, i, 0)*
                extract(executedPrices, i, 1);
            meanOperatingPrice +=Math.abs(extract(executedPrices, i, 0)*
                extract(executedPrices, i, 1));
        }
    TodayNumberOfShares=0;
    if(executedPrices.size() != 0){
        for (i=0;i<executedPrices.size();i++)
            TodayNumberOfShares+=extract(executedPrices, i, 1);
        meanOperatingPrice/=TodayNumberOfShares;
    }
    else
        meanOperatingPrice=mP;
    if (printing==2)
        System.out.println("I'm agent " + number
            + " and the meanOperatingPrice is: "
            + meanOperatingPrice);
        System.out.println();
    agentWealthAtMeanDailyPrice
        =shareValueAtMeanDailyPrice+liquidityQuantity;
        executedPrices.clear();
    }

    /** Return agent's wealth at mean daily price. */
    public double getWealthAtMeanDailyPrice()
    {
        return agentWealthAtMeanDailyPrice;
    }
}
```

BasicSumRuleMaster.java

```
// BasicSumRuleMaster.java
```

```
import swarm.Globals;
```

```

import swarm.defobj.Zone;
import swarm.objectbase.SwarmObjectImpl;

/**
 * This is the basic class of rule master, which is inherited from others.
 * A rule master contains the rule for the agent's acts.
 *
 *
 * @author Marco Agagliate
 *
 */
public class BasicSumRuleMaster extends SwarmObjectImpl
{
    /**
     * The probability of placing an order in the opening or closing phase.
     * So a day starts and finish with an auction, with a realistic effect.
     */
    public double agentProbToActInOpeningOrClosingAuctions;
    /**
     * The coefficient that RandomAgents multiplies
     * by the last price for the order.
     */
    public double minCorrectingCoeff;
    /**
     * The coefficient that RandomAgents multiplies
     * by the last price for the order.
     */
    public double maxCorrectingCoeff;
    /** The asymmetry of max/minCorrectingCoeff. */
    public double asymmetricRange;
    /** This is the floor price. */
    public double floorP;
    /**
     * The probability that an agent would buy
     * if the price is below floorP.
     */
    public double agentProbToActBelowFloorP;

    /** Constructor for a new BasicSumRuleMaster. */
    public BasicSumRuleMaster(Zone aZone)
    {
        super(aZone);
    }

    /** This method sets the agentProbToActInOpeningOrClosingAuctions. */
    public void setAgentProbToActInOpeningOrClosingAuctions(double p)
    {
        agentProbToActInOpeningOrClosingAuctions=p;
    }

    /** This method sets the minCorrectingCoeff. */
    public void setMinCorrectingCoeff(double min)
    {
        minCorrectingCoeff=min;
    }

    /** This method sets the setMaxCorrectingCoeff. */
    public void setMaxCorrectingCoeff(double max)
    {
        maxCorrectingCoeff=max;
    }

    /** This method sets the asymmetricRange. */
    public void setAsymmetricRange(double a)
    {
        asymmetricRange=a;
    }
}

```

Appendice C

```
/** This method sets the floorP and AgentProbToActBelowFloorP. */  
public void setFloorP$andAgentProbToActBelowFloorP(double f, double p)  
{  
    floorP=f;  
    agentProbToActBelowFloorP=p;  
}  
}
```

APPENDICE D

JAVADOC: DOCUMENTAZIONE DEL CODICE

Class StartJavaSum

```
java.lang.Object
├─StartJavaSum
```

```
public class StartJavaSum
extends java.lang.Object
```

The StartJavaSum class contains main(). We follow here the typical Swarm structure with main() (in Start... as a convention) generating the Observer and the Observer generating the Model.

Author:

Marco Agagliate

Constructor Summary

StartJavaSum()	
--------------------------------	--

Method Summary

static void	main (java.lang.String[] args)	The main() function is the top-level place where everything starts.
-------------	--	---

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

StartJavaSum

```
public StartJavaSum()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
    The main() function is the top-level place where everything starts.
```

Class ObserverSwarm

```
java.lang.Object
├─GUISwarmImpl
├─ObserverSwarm
```

Appendice D

```
public class ObserverSwarm
extends GUISwarmImpl
```

ObserverSwarm.java The observer swarm is collection of objects that are used to run and observe the ModelSwarm that actually comprises the simulation.

Author: Marco Agagliate, Antonio de Ruvo

See Also: [Serialized Form](#)

Field Summary	
EZGraphImpl	agentWealth our graphics or EZGraph output to files
EZGraphImpl	agentWealthGraph our graphics or EZGraph output to files
EZGraphImpl	bidAskSpreadGraph our graphics or EZGraph output to files
EZGraphImpl	bookLogGraph our graphics or EZGraph output to files
EZGraphImpl	currentBidAskSpreadFile our graphics or EZGraph output to files
EZGraphImpl	currentBuyFirstLastSpreadFile our graphics or EZGraph output to files
EZGraphImpl	currentQuantitySpreadFile our graphics or EZGraph output to files
EZGraphImpl	currentSellFirstLastSpreadFile our graphics or EZGraph output to files
ActionGroupImpl	displayActions two ActionGroup for sequence of GUI events
int	displayFrequency Update frequency.
int	displayPreviousDayMean To create file and represent previous day mean price.
ScheduleImpl	displaySchedule the single Schedule instance
EZGraphImpl	firstLastSpreadGraph our graphics or EZGraph output to files
EZGraphImpl	maxWealthAllFile our graphics or EZGraph output to files
EZGraphImpl	maxWealthLocallyImitatingFile our graphics or EZGraph output to files
EZGraphImpl	maxWealthMarketImitatingFile our graphics or EZGraph output to files
EZGraphImpl	maxWealthRandomFile our graphics or EZGraph output to files
EZGraphImpl	maxWealthStopLossFile our graphics or EZGraph output to files
EZGraphImpl	meanPriceFile our graphics or EZGraph output to files
EZGraphImpl	meanWealthAllFile our graphics or EZGraph output to files

EZGraphImpl	<u>meanWealthLocallyImitatingFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>meanWealthMarketImitatingFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>meanWealthRandomFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>meanWealthStopLossFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>minWealthAllFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>minWealthLocallyImitatingFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>minWealthMarketImitatingFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>minWealthRandomFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>minWealthStopLossFile</u> our graphics or EZGraph output to files
<u>ModelSwarm</u>	<u>modelSwarm</u> the ModelSwarm we are observing
EZGraphImpl	<u>priceFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>priceGraph</u> our graphics or EZGraph output to files
EZGraphImpl	<u>quantitySpreadGraph</u> our graphics or EZGraph output to files
int	<u>saveAgentWealthData</u> To save agent's wealth data.
int	<u>saveBidAskSpreadData</u> To save spread data between first bid and first ask price.
int	<u>saveBookLogData</u> To save book data.
int	<u>saveBuySellFirstLastSpreadData</u> To save spread data between the first and the last price in the two side.
int	<u>saveBuySellQuantitySpreadData</u> To save spread data between quantities in sell and buy side.
int	<u>savePriceData</u> To save on a file the price data.
EZGraphImpl	<u>sharesInBuySideFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>sharesInBuySideInOpeningOrClosingAuctionsFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>sharesInSellSideFile</u> our graphics or EZGraph output to files
EZGraphImpl	<u>sharesInSellSideInOpeningOrClosingAuctionsFile</u> our graphics or EZGraph output to files
int	<u>showAgentWealthGraph</u> To show agent's wealth graph.
int	<u>showBidAskSpreadGraph</u> To show spread graph between first bid and first ask price.
int	<u>showBookLogGraph</u>

Appendice D

	To show book graph.
int	showBuySellFirstLastSpreadGraph To show spread graph between the first and the last price in the two side.
int	showBuySellQuantitySpreadGraph To show spread graph between quantities in sell and buy side.
int	showBuySellSharesNumberInOpeningOrClosingAuctions To show the number of shares in the two side before opening or closing.
int	stopAtDayNumber To stop simulation at the end of a day.
Book	theBook The book of the simulation.

Constructor Summary	
ObserverSwarm (Zone aZone)	Constructor for a new ObserverSwarm.

Method Summary	
Activity	activateIn (Swarm swarmContext) Activate the schedules so that they are ready to run.
java.lang.Object	buildActions () Create the actions necessary for the simulation.
java.lang.Object	buildObjects () Create the objects used to display the model.
void	checkToStop () To check for the stopping conditions.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail
displayFrequency
public int displayFrequency Update frequency.
stopAtDayNumber
public int stopAtDayNumber To stop simulation at the end of a day.
savePriceData
public int savePriceData

To save on a file the price data.

displayPreviousDayMean

public int **displayPreviousDayMean**
To create file and represent previous day mean price.

showBookLogGraph

public int **showBookLogGraph**
To show book graph.

saveBookLogData

public int **saveBookLogData**
To save book data.

showAgentWealthGraph

public int **showAgentWealthGraph**
To show agent's wealth graph.

saveAgentWealthData

public int **saveAgentWealthData**
To save agent's wealth data.

showBidAskSpreadGraph

public int **showBidAskSpreadGraph**
To show spread graph between first bid and first ask price.

saveBidAskSpreadData

public int **saveBidAskSpreadData**
To save spread data between first bid and first ask price.

showBuySellQuantitySpreadGraph

public int **showBuySellQuantitySpreadGraph**
To show spread graph between quantities in sell and buy side.

saveBuySellQuantitySpreadData

public int **saveBuySellQuantitySpreadData**
To save spread data between quantities in sell and buy side.

showBuySellFirstLastSpreadGraph

public int **showBuySellFirstLastSpreadGraph**
To show spread graph between the first and the last price in the two side.

saveBuySellFirstLastSpreadData

public int **saveBuySellFirstLastSpreadData**
To save spread data between the first and the last price in the two side.

showBuySellSharesNumberInOpeningOrClosingAuctions

public int **showBuySellSharesNumberInOpeningOrClosingAuctions**
To show the number of shares in the two side before opening or closing.

modelSwarm

public [ModelSwarm](#) **modelSwarm**

Appendice D

the ModelSwarm we are observing

displaySchedule

```
public ScheduleImpl displaySchedule
    the single Schedule instance
```

theBook

```
public Book theBook
    The book of the simulation.
```

displayActions

```
public ActionGroupImpl displayActions
    two ActionGroup for sequence of GUI events
```

priceGraph

```
public EZGraphImpl priceGraph
    our graphics or EZGraph output to files
```

priceFile

```
public EZGraphImpl priceFile
    our graphics or EZGraph output to files
```

bookLogGraph

```
public EZGraphImpl bookLogGraph
    our graphics or EZGraph output to files
```

agentWealth

```
public EZGraphImpl agentWealth
    our graphics or EZGraph output to files
```

agentWealthGraph

```
public EZGraphImpl agentWealthGraph
    our graphics or EZGraph output to files
```

sharesInSellSideFile

```
public EZGraphImpl sharesInSellSideFile
    our graphics or EZGraph output to files
```

sharesInBuySideFile

```
public EZGraphImpl sharesInBuySideFile
    our graphics or EZGraph output to files
```

minWealthAllFile

```
public EZGraphImpl minWealthAllFile
    our graphics or EZGraph output to files
```

meanWealthAllFile

```
public EZGraphImpl meanWealthAllFile
    our graphics or EZGraph output to files
```

maxWealthAllFile

```
public EZGraphImpl maxWealthAllFile
```

our graphics or EZGraph output to files

minWealthRandomFile

```
public EZGraphImpl minWealthRandomFile
    our graphics or EZGraph output to files
```

meanWealthRandomFile

```
public EZGraphImpl meanWealthRandomFile
    our graphics or EZGraph output to files
```

maxWealthRandomFile

```
public EZGraphImpl maxWealthRandomFile
    our graphics or EZGraph output to files
```

minWealthMarketImitatingFile

```
public EZGraphImpl minWealthMarketImitatingFile
    our graphics or EZGraph output to files
```

meanWealthMarketImitatingFile

```
public EZGraphImpl meanWealthMarketImitatingFile
    our graphics or EZGraph output to files
```

maxWealthMarketImitatingFile

```
public EZGraphImpl maxWealthMarketImitatingFile
    our graphics or EZGraph output to files
```

minWealthLocallyImitatingFile

```
public EZGraphImpl minWealthLocallyImitatingFile
    our graphics or EZGraph output to files
```

meanWealthLocallyImitatingFile

```
public EZGraphImpl meanWealthLocallyImitatingFile
    our graphics or EZGraph output to files
```

maxWealthLocallyImitatingFile

```
public EZGraphImpl maxWealthLocallyImitatingFile
    our graphics or EZGraph output to files
```

minWealthStopLossFile

```
public EZGraphImpl minWealthStopLossFile
    our graphics or EZGraph output to files
```

meanWealthStopLossFile

```
public EZGraphImpl meanWealthStopLossFile
    our graphics or EZGraph output to files
```

maxWealthStopLossFile

```
public EZGraphImpl maxWealthStopLossFile
    our graphics or EZGraph output to files
```

meanPriceFile

```
public EZGraphImpl meanPriceFile
```

Appendice D

our graphics or EZGraph output to files

bidAskSpreadGraph

```
public EZGraphImpl bidAskSpreadGraph
    our graphics or EZGraph output to files
```

currentBidAskSpreadFile

```
public EZGraphImpl currentBidAskSpreadFile
    our graphics or EZGraph output to files
```

sharesInBuySideInOpeningOrClosingAuctionsFile

```
public EZGraphImpl sharesInBuySideInOpeningOrClosingAuctionsFile
    our graphics or EZGraph output to files
```

sharesInSellSideInOpeningOrClosingAuctionsFile

```
public EZGraphImpl sharesInSellSideInOpeningOrClosingAuctionsFile
    our graphics or EZGraph output to files
```

quantitySpreadGraph

```
public EZGraphImpl quantitySpreadGraph
    our graphics or EZGraph output to files
```

currentQuantitySpreadFile

```
public EZGraphImpl currentQuantitySpreadFile
    our graphics or EZGraph output to files
```

firstLastSpreadGraph

```
public EZGraphImpl firstLastSpreadGraph
    our graphics or EZGraph output to files
```

currentBuyFirstLastSpreadFile

```
public EZGraphImpl currentBuyFirstLastSpreadFile
    our graphics or EZGraph output to files
```

currentSellFirstLastSpreadFile

```
public EZGraphImpl currentSellFirstLastSpreadFile
    our graphics or EZGraph output to files
```

Constructor Detail

ObserverSwarm

```
public ObserverSwarm(Zone aZone)
    Constructor for a new ObserverSwarm.
```

Method Detail

buildObjects

```
public java.lang.Object buildObjects()
    Create the objects used to display the model.
```

buildActions

```
public java.lang.Object buildActions()
    Create the actions necessary for the simulation. This is where the schedule is built (but not run!) Here we
    create a display schedule - this is used to display the state of the world and check for user input.
```

activateIn

```
public Activity activateIn(Swarm swarmContext)
    Activate the schedules so that they are ready to run. The swarmContext argument is the zone in which the
    ObserverSwarm is activated. Typically the ObserverSwarm is the top-level swarm, so it is activated in
    "null". The other (sub)swarms and schedules will be activated inside of the ObserverSwarm context.
```

checkToStop

```
public void checkToStop()
    To check for the stopping conditions.
```

Class ModelSwarm

```
java.lang.Object
├─SwarmImpl
└─ModelSwarm
```

```
public class ModelSwarm
    extends SwarmImpl
```

The Model of JavaSum. The Model contains the Units and all the related tools.

Author:

Marco Agagliate

See Also:

[Serialized Form](#)

Field Summary	
ListImpl	agentList The list of all the agents.
int	agentNumber The total number of agents.
double	agentProbToActBelowFloorP The probability to act below floor price.
double	agentProbToActInOpeningOrClosingAuctions The probability of placing an order in the opening or closing phase.
double	agentProbToActWithMarketPrice The probability of placing an order with market price.
RandomAgent	anAgent1 The agents of the simulation.
MarketImitatingAgent	anAgent2
LocallyImitatingAgent	anAgent3
StopLossAgent	anAgent4
double	asymmetricBuySellProb The asymmetry of buy and sell probability.
double	asymmetricRange The asymmetry of max/minCorrectingCoeff.
int	checkingIfShortOrLong To check the position(sharequantity) of the agent.
int	dayNumber The number of current day.
double	floorP The minimum price at which the agents acts.

Appendice D

ListImpl	<u>indexAgentList</u> The list of the agents in create order.
ListIndex	<u>indexAgentListIndex</u> Its iterator.
int	<u>istantNumber</u> The total number of istants per day.
ListImpl	<u>locallyImitatingAgentList</u> The list of the LocallyImitatingAgents.
int	<u>locallyImitatingAgentNumber</u>
ListImpl	<u>marketImitatingAgentList</u> The list of the MarketImitatingAgents.
int	<u>marketImitatingAgentNumber</u>
double	<u>maxCorrectingCoeff</u> The coefficient that RandomAgents multiplays by the last price for the order.
double	<u>maxLossRate</u> The max rate of loss that stop loss agents effort.
int	<u>maxNumberOfOperatingAgents</u> The maximum number of operating agents.
int	<u>maxOrderQuantity</u> The maximum number of order per agent.
double	<u>minCorrectingCoeff</u> The coefficient that RandomAgents multiplays by the last price for the order.
ActionGroupImpl	<u>modelActions1</u> ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	<u>modelActions2</u> ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	<u>modelActions3</u> ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	<u>modelActions4</u> ActionGroup for holding an ordered sequence of action.
ActionGroupImpl	<u>modelActionsLS</u> ActionGroup for holding an ordered sequence of action.
ScheduleImpl	<u>modelSchedule</u> The Schedule operating in the Model.
int	<u>percentageOfOperatingAgents</u> The percentage quota of agents which act in the market.
int	<u>printing</u> If 1 book prints data on the terminal window, 2 for Agent.
ListImpl	<u>randomAgentList</u> The list of the RandomAgents.
int	<u>randomAgentNumber</u>
<u>RandomRuleMaster</u>	<u>randomRuleMaster</u> The randomRuleMaster manages RandomAgents.
ListImpl	<u>stopLossAgentList</u> The list of the StopLossAgents.
int	<u>stopLossAgentNumber</u>
int	<u>stopLossInterval</u>

	The interval that stop loss agents consider for their strategy.
StopLossRuleMaster	stopLossRuleMaster The stopLossRuleMaster manages StopLossAgents.
Book	theBook The book of the simulation.
CurrentAgent	theCurrentAgent This is a "ghost agent".
CurrentIstant	theCurrentIstant The object for the calls of the agents.
boolean	typeOfPercentage The type of percentage of agents.

Constructor Summary	
ModelSwarm (Zone aZone)	Constructor for a new ModelSwarm.

Method Summary	
Activity	activateIn (Swarm swarmContext) Now set up the model's activation. swarmContext indicates where we're being started in - typically, this model is run as a subswarm of an observer swarm.
java.lang.Object	buildActions () Here is where the model schedule is built, the data structures that define the simulation of time in the model.
java.lang.Object	buildObjects () Build the model objects.
ListImpl	getAgentList () The method returns the list of the agents.
ListIndex	getAgentListIndex () The method returns the index of indexAgentList.
Book	getBook () The method returns the book.
int	getCurrentDay () The method returns the number of the day.
ListImpl	getLocallyImitatingAgentList () The method returns the list of the LocallyImitatingAgents.
ListImpl	getMarketImitatingAgentList () The method returns the list of the MarketImitatingAgents.
ListImpl	getRandomAgentList () The method returns the list of the RandomAgents.
ListImpl	getStopLossAgentList () The method returns the list of the StopLossAgents.
void	increaseCurrentDayNumber () The method increases the number of the day.
void	openProbeTo (int n) The method opens the probe on an agent.
void	setAgentNumber () The method set the number of the agents.

Appendice D

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

randomAgentNumber

```
public int randomAgentNumber
```

marketImitatingAgentNumber

```
public int marketImitatingAgentNumber
```

locallyImitatingAgentNumber

```
public int locallyImitatingAgentNumber
```

stopLossAgentNumber

```
public int stopLossAgentNumber
```

agentNumber

```
public int agentNumber  
    The total number of agents.
```

istantNumber

```
public int istantNumber  
    The total number of istants per day.
```

dayNumber

```
public int dayNumber  
    The number of current day.
```

asymmetricBuySellProb

```
public double asymmetricBuySellProb  
    The asimmetry of buy and sell probability.
```

minCorrectingCoeff

```
public double minCorrectingCoeff  
    The coefficient that RandomAgents multiplays by the last price for the order.
```

maxCorrectingCoeff

```
public double maxCorrectingCoeff  
    The coefficient that RandomAgents multiplays by the last price for the order.
```

asymmetricRange

```
public double asymmetricRange  
    The asimmetry of max/minCorrectingCoeff.
```

agentProbToActWithMarketPrice

```
public double agentProbToActWithMarketPrice
    The probability of placing an order with market price.
```

```
agentProbToActInOpeningOrClosingAuctions
```

```
public double agentProbToActInOpeningOrClosingAuctions
    The probability of placing an order in the opening or closing phase. So a day starts and finish with an
    action, with a realistic effect.
```

```
floorP
```

```
public double floorP
    The minimum price at which the agents acts.
```

```
agentProbToActBelowFloorP
```

```
public double agentProbToActBelowFloorP
    The probability to act below floor price.
```

```
maxOrderQuantity
```

```
public int maxOrderQuantity
    The maximum number of order per agent.
```

```
maxLossRate
```

```
public double maxLossRate
    The max rate of loss that stop loss agents effort.
```

```
stopLossInterval
```

```
public int stopLossInterval
    The interval that stop loss agents consider for their strategy.
```

```
checkingIfShortOrLong
```

```
public int checkingIfShortOrLong
    To check the position(sharequantity) of the agent.
```

```
percentageOfOperatingAgents
```

```
public int percentageOfOperatingAgents
    The percentage quota of agents which act in the market.
```

```
maxNumberOfOperatingAgents
```

```
public int maxNumberOfOperatingAgents
    The maximum number of operating agents.
```

```
typeOfPercentage
```

```
public boolean typeOfPercentage
    The type of percentage of agents. If it is true percentage is a maximum value; if it is false percentage is a
    fixed value.
```

```
printing
```

```
public int printing
    If 1 book prints data on the terminal window, 2 for Agent.
```

```
agentList
```

```
public ListImpl agentList
    The list of all the agents.
```

Appendice D

randomAgentList

```
public ListImpl randomAgentList  
    The list of the RandomAgents.
```

marketImitatingAgentList

```
public ListImpl marketImitatingAgentList  
    The list of the MarketImitatingAgents.
```

locallyImitatingAgentList

```
public ListImpl locallyImitatingAgentList  
    The list of the LocallyImitatingAgents.
```

stopLossAgentList

```
public ListImpl stopLossAgentList  
    The list of the StopLossAgents.
```

indexAgentList

```
public ListImpl indexAgentList  
    The list of the agents in create order.
```

indexAgentListIndex

```
public ListIndex indexAgentListIndex  
    Its iterator.
```

modelActions1

```
public ActionGroupImpl modelActions1  
    ActionGroup for holding an ordered sequence of action.
```

modelActionsLS

```
public ActionGroupImpl modelActionsLS  
    ActionGroup for holding an ordered sequence of action.
```

modelActions2

```
public ActionGroupImpl modelActions2  
    ActionGroup for holding an ordered sequence of action.
```

modelActions3

```
public ActionGroupImpl modelActions3  
    ActionGroup for holding an ordered sequence of action.
```

modelActions4

```
public ActionGroupImpl modelActions4  
    ActionGroup for holding an ordered sequence of action.
```

modelSchedule

```
public ScheduleImpl modelSchedule  
    The Schedule operating in the Model.
```

theBook

```
public Book theBook  
    The book of the simulation.
```

anAgent1

public [RandomAgent](#) **anAgent1**
The agents of the simulation.

anAgent2

public [MarketImitatingAgent](#) **anAgent2**

anAgent3

public [LocallyImitatingAgent](#) **anAgent3**

anAgent4

public [StopLossAgent](#) **anAgent4**

randomRuleMaster

public [RandomRuleMaster](#) **randomRuleMaster**
The randomRuleMaster manages RandomAgents.

stopLossRuleMaster

public [StopLossRuleMaster](#) **stopLossRuleMaster**
The stopLossRuleMaster manages StopLossAgents.

theCurrentAgent

public [CurrentAgent](#) **theCurrentAgent**
This is a "ghost agent". See the comments of CurrentAgent class.

theCurrentIstant

public [CurrentIstant](#) **theCurrentIstant**
The object for the calls of the agents.

Constructor Detail

ModelSwarm

public **ModelSwarm**(Zone aZone)
Constructor for a new ModelSwarm.

Method Detail

buildObjects

public java.lang.Object **buildObjects**()
Build the model objects.

buildActions

public java.lang.Object **buildActions**()
Here is where the model schedule is built, the data structures that define the simulation of time in the model. The core is an actionGroup that has a list of actions. Then that's put in a Schedule.

activateIn

public Activity **activateIn**(Swarm swarmContext)
Now set up the model's activation. swarmContext indicates where we're being started in - typically, this model is run as a subswarm of an observer swarm.

increaseCurrentDayNumber

public void **increaseCurrentDayNumber**()
The method increases the number of the day.

Appendice D

setAgentNumber

```
public void setAgentNumber()
```

The method set the number of the agents.

getCurrentDay

```
public int getCurrentDay()
```

The method returns the number of the day.

getAgentList

```
public ListImpl getAgentList()
```

The method returns the list of the agents.

getRandomAgentList

```
public ListImpl getRandomAgentList()
```

The method returns the list of the RandomAgents.

getMarketImitatingAgentList

```
public ListImpl getMarketImitatingAgentList()
```

The method returns the list of the MarketImitatingAgents.

getLocallyImitatingAgentList

```
public ListImpl getLocallyImitatingAgentList()
```

The method returns the list of the LocallyImitatingAgents.

getStopLossAgentList

```
public ListImpl getStopLossAgentList()
```

The method returns the list of the StopLossAgents.

getAgentListIndex

```
public ListIndex getAgentListIndex()
```

The method returns the index of indexAgentList.

getBook

```
public Book getBook()
```

The method returns the book.

openProbeTo

```
public void openProbeTo(int n)
```

The method opens the probe on an agent.

Class Book

```
java.lang.Object  
├─SwarmObjectImpl  
└─Book
```

```
public class Book  
extends SwarmObjectImpl
```

This is the book. The book works on the basis of two arrayList containing sell order in increasing order or buy order in decreasing order. The orders are arrays which have in the first position the price, in the second the number of agent who places the order and in the third the quantity. If an order obtains an immediate matching, it is not filed

Author: Antonio de Ruvo e-mail:antderu@libero.it, Integrations of Bruno Mencarelli

See Also: [Serialized Form](#)

Field Summary	
java.util.ArrayList	buyOrderStorehouse The arrayList which contains the buy orders.
java.util.ArrayList	buyOrderWithAuctionPriceStorehouse The arrayList which contains the buy orders with auction price.
java.util.ArrayList	closingContractLog The arrayList to log the closing contracts.
int	count The number of share negotiated
double	currentMeanPrice The addition of prices*quantity to calculate the mean price.
double	executedPrice The price of the last contract closing.
ListIndex	indexAgentListIndex The index used for send message to agents.
int	istantNumber The total number of istant.
double[]	localHistory The vector containing the local price history.
int	localHistoryLength The length of the local price history.
double	meanPrice The mean price of yesterday.
double[]	meanPriceHistory The vector containing the history of mean prices.
int	meanPriceHistoryLength The length of the mean price history.
int	nOfBuyOrderWithAuctionPrice The number of buy order with auction price.
int	nOfBuySharesWithAuctionPrice The number ofshares with auction price in buy side.
int	nOfSellOrderWithAuctionPrice The number of sell order with auction price.
int	nOfSellSharesWithAuctionPrice The number ofshares with auction price in sell side.
int	numberOfIstant The number of actual istant.
double	openingPrice The opening price.
int	orderNumber The number of orders received from agents.
double	previousClosingPrice The closing price of yesterday.
int	printing If 1 many objects print data on the terminal window.
double	referencePrice The reference price of each days.

Appendice D

java.util.ArrayList	<u>sellOrderStorehouse</u> The arrayList which contains the sell orders.
java.util.ArrayList	<u>sellOrderWithAuctionPriceStorehouse</u> The arrayList which contains the sell orders with auction price.
int	<u>sharesInBuySide</u> The number of shares in buy side of book.
int	<u>sharesInBuySideInOpeningOrClosingAuctions</u> The number of shares in buy side of book in opening or closing Auctions.
int	<u>sharesInSellSide</u> The number of shares in sell side of book.
int	<u>sharesInSellSideInOpeningOrClosingAuctions</u> The number of shares in sell side of book in opening or closing Auctions.
java.util.ArrayList	<u>tableForTheoreticalPrice</u> The arrayList as a table to evaluate the teoric price.
double	<u>theoreticalPrice</u> The theoretical price.

Constructor Summary

<u>Book</u> (Zone aZone) Constructor for a new book.	
---	--

Method Summary

private double	<u>extract</u> (java.util.ArrayList nameList, int indexOrder, int positionDatum) To extract data from the orders received from agents.
private int	<u>fillStorehouseDecreasingP</u> (double newPrice) To fill buyOrderStorehouse in decreasing order.
private int	<u>fillStorehouseIncreasingP</u> (double newPrice) To fill sellOrderStorehouse in increasing order
double	<u>getBuyFirstLastSpread</u> () To get the spread between the first and the last price in the buy side.
int	<u>getBuySellQuantitySpread</u> () To get the spread between quantities in sell and buy side.
double	<u>getLaggedMeanPrice</u> (int lag) to get the lagged Mean Price.
int	<u>getLocal</u> (double[] vect) This is the getLocal method, that counts for the difference between the buying and selling orders sent by the agents.
int	<u>getLocalHistory</u> () to get the local History.
double	<u>getMeanPrice</u> () To get the mean price of yesterday.
double	<u>getPreviousClosingPrice</u> () To get the closing price of yesterday.
double	<u>getPrice</u> () To get the last executed price.
double	<u>getSellFirstLastSpread</u> ()

	To get the spread between the first and the last price in the sell side.
int	getSharesInBuySide() To get the number of shares filled in the buy side of book.
int	getSharesInBuySideInOpeningOrClosingAuctions() To get the number of shares in buy side in opening or closing Auctions.
int	getSharesInSellSide() To get the number of shares filled in the sell side of book.
int	getSharesInSellSideInOpeningOrClosingAuctions() To get the number of shares in sell side in opening or closing Auctions.
double	getSpread() To get the spread between first bid and first ask price.
private void	message(double n, double p, double q) To inform the agent of executed price.
void	opening() To conclude contracts (if opening price > 0) or to fill the order with auction price.
void	recreateTableForTheoreticalPrice()
void	setAgentListIndex(ListIndex i) To know the adress of memory of the agent.
void	setClean() At the beginning of each day to clean the book.
void	setIstantNumber(int n) To set the value of total number of istant
void	setLocal(double[] vect, double p) This is the SetLocal method, used to shift the rows of the localHistory vector.
void	setMeanPrice() At the end of each day to calculate the mean price.
void	setNumberOfIstant() To set the value of actual istant
void	setOpeningPrice() To set the opening price.
void	setOrderFromAgent(double[] orderFromAgent) Receiving an order when the market is open.
void	setOrderInOpeningOrClosingAuctionsFromAgent(double[] orderFromAgent) Receiving an order in opening or closing Auctions from an agent.
void	setPrinting(int p) To set the value of option printing.
void	setRefencePrice() At the end of each day to calculate the reference price.
private double[]	setRows(double[] order) To set rows in tableForTheoreticalPrice.
private void	setTableForTheoreticalPrice(double[] order) To set the prices in decreasing order for evaluete the theoretical price.
void	setTheoreticalPrice() To set the theoretical price.
private double[]	standardizedOrder(double[] orderFromAgent) To standardize the order.

Methods inherited from class java.lang.Object

Appendice D

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait,
wait, wait
```

Field Detail

indexAgentListIndex

```
public ListIndex indexAgentListIndex
    The index used for send message to agents.
```

printing

```
public int printing
    If 1 many objects print data on the terminal window.
```

istantNumber

```
public int istantNumber
    The total number of istant.
```

numberOfIstant

```
public int numberOfIstant
    The number of actual istant.
```

orderNumber

```
public int orderNumber
    The number of orders received from agents.
```

count

```
public int count
    The number of share negotiated
```

sharesInBuySide

```
public int sharesInBuySide
    The number of shares in buy side of book.
```

sharesInSellSide

```
public int sharesInSellSide
    The number of shares in sell side of book.
```

sharesInBuySideInOpeningOrClosingAuctions

```
public int sharesInBuySideInOpeningOrClosingAuctions
    The number of shares in buy side of book in opening or closing Auctions.
```

sharesInSellSideInOpeningOrClosingAuctions

```
public int sharesInSellSideInOpeningOrClosingAuctions
    The number of shares in sell side of book in opening or closing Auctions.
```

executedPrice

```
public double executedPrice
    The price of the last contract closing.
```

meanPrice

public double **meanPrice**
The mean price of yesterday.

previousClosingPrice

public double **previousClosingPrice**
The closing price of yesterday.

currentMeanPrice

public double **currentMeanPrice**
The addition of prices*quantity to calculate the mean price.

buyOrderStorehouse

public java.util.ArrayList **buyOrderStorehouse**
The arrayList which contains the buy orders.

sellOrderStorehouse

public java.util.ArrayList **sellOrderStorehouse**
The arrayList which contains the sell orders.

closingContractLog

public java.util.ArrayList **closingContractLog**
The arrayList to log the closing contracts.

tableForTheoreticalPrice

public java.util.ArrayList **tableForTheoreticalPrice**
The arrayList as a table to evaluate the teoric price.

buyOrderWithAuctionPriceStorehouse

public java.util.ArrayList **buyOrderWithAuctionPriceStorehouse**
The arrayList which contains the buy orders with auction price.

sellOrderWithAuctionPriceStorehouse

public java.util.ArrayList **sellOrderWithAuctionPriceStorehouse**
The arrayList which contains the sell orders with auction price.

referencePrice

public double **referencePrice**
The reference price of each days.

theoreticalPrice

public double **theoreticalPrice**
The theoretical price.

openingPrice

public double **openingPrice**
The opening price.

nOfBuyOrderWithAuctionPrice

public int **nOfBuyOrderWithAuctionPrice**
The number of buy order with auction price.

nOfSellOrderWithAuctionPrice

Appendice D

public int **nOfSellOrderWithAuctionPrice**
The number of sell order with auction price.

nOfBuySharesWithAuctionPrice

public int **nOfBuySharesWithAuctionPrice**
The number of shares with auction price in buy side.

nOfSellSharesWithAuctionPrice

public int **nOfSellSharesWithAuctionPrice**
The number of shares with auction price in sell side.

meanPriceHistoryLength

public int **meanPriceHistoryLength**
The length of the mean price history.

localHistoryLength

public int **localHistoryLength**
The length of the local price history.

meanPriceHistory

public double[] **meanPriceHistory**
The vector containing the history of mean prices.

localHistory

public double[] **localHistory**
The vector containing the local price history.

Constructor Detail

Book

public **Book**(Zone aZone)
Constructor for a new book.

Method Detail

message

private void **message**(double n,
double p,
double q)
To inform the agent of executed price.
See Also:
[BasicSumAgent](#)

standardizedOrder

private double[] **standardizedOrder**(double[] orderFromAgent)
To standardize the order.

extract

private double **extract**(java.util.ArrayList nameList,
int indexOrder,
int positionDatum)
To extract data from the orders received from agents.

fillStorehouseIncreasingP

private int **fillStorehouseIncreasingP**(double newPrice)
To fill sellOrderStorehouse in increasing order

fillStorehouseDecreasingP

```
private int fillStorehouseDecreasingP(double newPrice)
    To fill buyOrderStorehouse in decreasing order.
```

setLocal

```
public void setLocal(double[] vect,
                    double p)
    This is the SetLocal method, used to shift the rows of the localHistory vector.
```

getLocal

```
public int getLocal(double[] vect)
    This is the getLocal method, that counts for the difference between the buying and selling orders sent by the agents.
```

setPrinting

```
public void setPrinting(int p)
    To set the value of option printing.
```

setIstantNumber

```
public void setIstantNumber(int n)
    To set the value of total number of istant
```

setAgentListIndex

```
public void setAgentListIndex(ListIndex i)
    To know the adress of memory of the agent.
```

setNumberOfIstant

```
public void setNumberOfIstant()
    To set the value of actual istant
```

setMeanPrice

```
public void setMeanPrice()
    At the end of each day to calculate the mean price.
```

setRefencePrice

```
public void setRefencePrice()
    At the end of each day to calculate the reference price.
```

setClean

```
public void setClean()
    At the beginning of each day to clean the book.
```

setRows

```
private double[] setRows(double[] order)
    To set rows in tableForTheoreticalPrice.
```

setTableForTheoreticalPrice

```
private void setTableForTheoreticalPrice(double[] order)
    To set the prices in decreasing order for evaluete the theoretical price.
```

setTheoreticalPrice

```
public void setTheoreticalPrice()
```

Appendice D

To set the theoretical price.

setOpeningPrice

```
public void setOpeningPrice()  
    To set the opening price.
```

opening

```
public void opening()  
    To conclude contracts (if opening price > 0) or to fill the order with auction price.
```

recreateTableForTheoreticalPrice

```
public void recreateTableForTheoreticalPrice()
```

setOrderInOpeningOrClosingAuctionsFromAgent

```
public void  
setOrderInOpeningOrClosingAuctionsFromAgent(double[] orderFromAgent)  
    Receiving an order in opening or closing Auctions from an agent.
```

setOrderFromAgent

```
public void setOrderFromAgent(double[] orderFromAgent)  
    Receiving an order when the market is open.
```

getPrice

```
public double getPrice()  
    To get the last executed price.
```

getMeanPrice

```
public double getMeanPrice()  
    To get the mean price of yesterday.
```

getSharesInSellSide

```
public int getSharesInSellSide()  
    To get the number of shares filled in the sell side of book.
```

getSharesInBuySide

```
public int getSharesInBuySide()  
    To get the number of shares filled in the buy side of book.
```

getPreviousClosingPrice

```
public double getPreviousClosingPrice()  
    To get the closing price of yesterday.
```

getSellFirstLastSpread

```
public double getSellFirstLastSpread()  
    To get the spread between the first and the last price in the sell side.
```

getBuyFirstLastSpread

```
public double getBuyFirstLastSpread()  
    To get the spread between the first and the last price in the buy side.
```

getSpread

```
public double getSpread()
```

To get the spread between first bid and first ask price.

getBuySellQuantitySpread

```
public int getBuySellQuantitySpread()
    To get the spread between quantities in sell and buy side.
```

getSharesInSellSideInOpeningOrClosingAuctions

```
public int getSharesInSellSideInOpeningOrClosingAuctions()
    To get the number of shares in sell side in opening or closing Auctions.
```

getSharesInBuySideInOpeningOrClosingAuctions

```
public int getSharesInBuySideInOpeningOrClosingAuctions()
    To get the number of shares in buy side in opening or closing Auctions.
```

getLaggedMeanPrice

```
public double getLaggedMeanPrice(int lag)
    to get the lagged Mean Price.
```

getLocalHistory

```
public int getLocalHistory()
    to get the local History.
```

Class BasicSumAgent

```
java.lang.Object
├─SwarmObjectImpl
└─BasicSumAgent
```

Direct Known Subclasses:

[LocallyImitatingAgent](#), [MarketImitatingAgent](#), [RandomAgent](#), [StopLossAgent](#)

```
public class BasicSumAgent
    extends SwarmObjectImpl
```

This is the basic class of agents, that is inherited from all the others. It contains the common characteristics of the agents.

Author:

Marco Agagliate, Antonio de Ruvo

See Also:

[Serialized Form](#)

Field Summary	
double	agentProbToActWithMarketPrice This is the probability of placing an order with market price.
double	agentWealthAtMeanDailyPrice The agent's wealth at mean daily price.
double	asymmetricBuySellProb The asymmetry of buy and sell probability.
double	buySellSwitch The distribution of buy and sell order (fixed to 0.5).
java.util.ArrayList	executedPrices The matrix of orders' prices.
int	iMax The quantity of orders sending out by the agent.

Appendice D

double	liquidityQuantity The liquidity of the agent.
int	maxOrderQuantity The maximum number of order per agent.
double	meanOperatingPrice The average price of the satisfied order.
int	number This is the number of the agent.
double	price The price of the order.
int	printing If printing=1 many objects print data on the terminal window; If printing=2 BasicSumAgents print data on the terminal window.
double	shareQuantity The total agent's quantity of shares
double	shareValueAtMeanDailyPrice The value of the shares at mean daily price.
Book	theBook The book of the model.

Constructor Summary

[BasicSumAgent](#) (Zone aZone, int maxOrderQuantity)
Constructor for a new BasicSumAgent.

Method Summary

void	act0 () This method exists only for the CurrentAgent.
void	act1 () This method exists only for the CurrentAgent.
void	act2 () This is the method that the agent calls at the end of the day.
double	extract (java.util.ArrayList nameList, int indexOrder, int positionDatum) To extract data from the orders received from agents.
double	getWealthAtMeanDailyPrice () Return agent's wealth at mean daily price.
void	setAgentProbToActWithMarketPrice (double p) This method sets the probability of placing an order with market price. .
void	setAsymmetricBuySellProb (double p) This method sets the asymmetricBuySellProb.
void	setBook (Book b) This method sets the book.
void	setConfirmationOfExecutedPrice (double[] p) The number of executed prices is increased
void	setMaxOrderQuantity (int m) This method sets the maximum number of order per agent.
void	setNumber (int n)

	This method sets the number of the agent.
void	setPrinting (int p) Option about the agent print capability.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

number

```
public int number
    This is the number of the agent.
```

iMax

```
public int iMax
    The quantity of orders sending out by the agent. There is one share per order: the quantity of the order is the number of orders.
```

agentProbToActWithMarketPrice

```
public double agentProbToActWithMarketPrice
    This is the probability of placing an order with market price.
```

printing

```
public int printing
    If printing=1 many objects print data on the terminal window; If printing=2 BasicSumAgents print data on the terminal window.
```

price

```
public double price
    The price of the order.
```

asymmetricBuySellProb

```
public double asymmetricBuySellProb
    The asymmetry of buy and sell probability.
```

buySellSwitch

```
public double buySellSwitch
    The distribution of buy and sell order (fixed to 0.5).
```

shareQuantity

```
public double shareQuantity
    The total agent's quantity of shares
```

maxOrderQuantity

```
public int maxOrderQuantity
    The maximum number of order per agent.
```

Appendice D

shareValueAtMeanDailyPrice

```
public double shareValueAtMeanDailyPrice
    The value of the shares at mean daily price.
```

liquidityQuantity

```
public double liquidityQuantity
    The liquidity of the agent.
```

agentWealthAtMeanDailyPrice

```
public double agentWealthAtMeanDailyPrice
    The agent's wealth at mean daily price.
```

meanOperatingPrice

```
public double meanOperatingPrice
    The average price of the satisfied order.
```

executedPrices

```
public java.util.ArrayList executedPrices
    The matrix of orders' prices.
```

theBook

```
public Book theBook
    The book of the model.
```

Constructor Detail

BasicSumAgent

```
public BasicSumAgent(Zone aZone,
                    int maxOrderQuantity)
    Constructor for a new BasicSumAgent.
```

Method Detail

extract

```
public double extract(java.util.ArrayList nameList,
                    int indexOrder,
                    int positionDatum)
    To extract data from the orders received from agents.
```

setNumber

```
public void setNumber(int n)
    This method sets the number of the agent.
```

setAgentProbToActWithMarketPrice

```
public void setAgentProbToActWithMarketPrice(double p)
    This method sets the probability of placing an order with market price. .
```

setAsymmetricBuySellProb

```
public void setAsymmetricBuySellProb(double p)
    This method sets the asymmetricBuySellProb.
```

setBook

```
public void setBook(Book b)
    This method sets the book.
```

setMaxOrderQuantity

public void **setMaxOrderQuantity**(int m)
 This method sets the maximum number of order per agent.

setPrinting

public void **setPrinting**(int p)
 Option about the agent print capability.

setConfirmationOfExecutedPrice

public void **setConfirmationOfExecutedPrice**(double[] p)
 The number of executed prices is increased

act0

public void **act0**()
 This method exists only for the CurrentAgent. Its implementation is defined into specify agent class.

act1

public void **act1**()
 This method exists only for the CurrentAgent. Its implementation is defined into specify agent class.

act2

public void **act2**()
 This is the method that the agent calls at the end of the day. The agent makes daily accounting.

getWealthAtMeanDailyPrice

public double **getWealthAtMeanDailyPrice**()
 Return agent's wealth at mean daily price.

Class BasicSumRuleMaster

java.lang.Object
 ↳SwarmObjectImpl
 ↳**BasicSumRuleMaster**

Direct Known Subclasses:
[RandomRuleMaster](#), [StopLossRuleMaster](#)

public class **BasicSumRuleMaster**
 extends SwarmObjectImpl

This is the basic class of rule master, which is inherited from others. A rule master contains the rule for the agent's acts.

Author: Marco Agagliate

See Also: [Serialized Form](#)

Field Summary	
double	agentProbToActBelowFloorP The probability that an agent would buy if the price is below floorP.
double	agentProbToActInOpeningOrClosingAuctions The probability of placing an order in the opening or closing phase.
double	asymmetricRange The asymmetry of max/minCorrectingCoeff.
double	floorP This is the floor price.

Appendice D

double	maxCorrectingCoeff The coefficient that RandomAgents multiplays by the last price for the order.
double	minCorrectingCoeff The coefficient that RandomAgents multiplays by the last price for the order.

Constructor Summary

[BasicSumRuleMaster](#)(Zone aZone)
Constructor for a new BasicSumRuleMaster.

Method Summary

void	setAgentProbToActInOpeningOrClosingAuctions (double p) This method sets the agentProbToActInOpeningOrClosingAuctions.
void	setAsymmetricRange (double a) This method sets the asymmetricRange.
void	setFloorP\$andAgentProbToActBelowFloorP (double f, double p) This method sets the floorP and AgentProbToActBelowFloorP.
void	setMaxCorrectingCoeff (double max) This method sets the setMaxCorrectingCoeff.
void	setMinCorrectingCoeff (double min) This method sets the minCorrectingCoeff.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

agentProbToActInOpeningOrClosingAuctions

public double **agentProbToActInOpeningOrClosingAuctions**
The probability of placing an order in the opening or closing phase. So a day starts and finish with an auction, with a realistic effect.

minCorrectingCoeff

public double **minCorrectingCoeff**
The coefficient that RandomAgents multiplays by the last price for the order.

maxCorrectingCoeff

public double **maxCorrectingCoeff**
The coefficient that RandomAgents multiplays by the last price for the order.

asymmetricRange

public double **asymmetricRange**
The asimmetry of max/minCorrectingCoeff.

floorP

```
public double floorP
    This is the floor price.
```

agentProbToActBelowFloorP

```
public double agentProbToActBelowFloorP
    The probability that an agent would buy if the price is below floorP.
```

Constructor Detail

BasicSumRuleMaster

```
public BasicSumRuleMaster(Zone aZone)
    Constructor for a new BasicSumRuleMaster.
```

Method Detail

setAgentProbToActInOpeningOrClosingAuctions

```
public void setAgentProbToActInOpeningOrClosingAuctions(double p)
    This method sets the agentProbToActInOpeningOrClosingAuctions.
```

setMinCorrectingCoeff

```
public void setMinCorrectingCoeff(double min)
    This method sets the minCorrectingCoeff.
```

setMaxCorrectingCoeff

```
public void setMaxCorrectingCoeff(double max)
    This method sets the setMaxCorrectingCoeff.
```

setAsymmetricRange

```
public void setAsymmetricRange(double a)
    This method sets the asymmetricRange.
```

setFloorP\$andAgentProbToActBelowFloorP

```
public void setFloorP$andAgentProbToActBelowFloorP(double f,
    double p)
    This method sets the floorP and AgentProbToActBelowFloorP.
```

Class RandomAgent

```
java.lang.Object
    ↳ SwarmObjectImpl
        ↳ BasicSumAgent
            ↳ RandomAgent
```

```
public class RandomAgent
    extends BasicSumAgent
```

This is the RandomAgent. RandomAgent acts at random.

Author: Marco Agagliate integrations of Antonio de Ruvo

See Also: [Serialized Form](#)

Field Summary

double []	order
-----------	-----------------------

Appendice D

	This variable is for send order to the book.
RandomRuleMaster	ruleMaster This is the agent's rule master.

Fields inherited from class [BasicSumAgent](#)

[agentProbToActWithMarketPrice](#), [agentWealthAtMeanDailyPrice](#),
[asymmetricBuySellProb](#), [buySellSwitch](#), [executedPrices](#), [iMax](#), [liquidityQuantity](#),
[maxOrderQuantity](#), [meanOperatingPrice](#), [number](#), [price](#), [printing](#), [shareQuantity](#),
[shareValueAtMeanDailyPrice](#), [theBook](#)

Constructor Summary

[RandomAgent](#)(Zone aZone, int maxOrderQuantity)
Constructor for a new RandomAgent.

Method Summary

void	act0 () This method is called in the first phase of the day.
void	act1 () This method is called in the first phase of the day.
void	setRuleMaster (RandomRuleMaster r) This method sets the rule master.

Methods inherited from class [BasicSumAgent](#)

[act2](#), [extract](#), [getWealthAtMeanDailyPrice](#), [setAgentProbToActWithMarketPrice](#),
[setAsymmetricBuySellProb](#), [setBook](#), [setConfirmationOfExecutedPrice](#),
[setMaxOrderQuantity](#), [setNumber](#), [setPrinting](#)

Methods inherited from class java.lang.Object

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#),
[wait](#), [wait](#)

Field Detail

order

public double[] **order**
This variable is for send order to the book.

ruleMaster

public [RandomRuleMaster](#) **ruleMaster**
 This is the agent's rule master.

Constructor Detail

RandomAgent

public **RandomAgent**(Zone aZone,
 int maxOrderQuantity)
 Constructor for a new RandomAgent.

Method Detail

setRuleMaster

public void **setRuleMaster**([RandomRuleMaster](#) r)
 This method sets the rule master.

act0

public void **act0**()
 This method is called in the first phase of the day.
Overrides:
[act0](#) in class [BasicSumAgent](#)

act1

public void **act1**()
 This method is called in the first phase of the day. The agent acts in the market.
Overrides:
[act1](#) in class [BasicSumAgent](#)

Class RandomRuleMaster

```
java.lang.Object
├─SwarmObjectImpl
│   └─BasicSumRuleMaster
│       └─RandomRuleMaster
```

public class **RandomRuleMaster**
 extends [BasicSumRuleMaster](#)

This is the RandomRuleMaster. A rule master contains the rule for the agent's acts.

Author: Marco Agagliate

See Also: [Serialized Form](#)

Field Summary

Fields inherited from class [BasicSumRuleMaster](#)

[agentProbToActBelowFloorP](#), [agentProbToActInOpeningOrClosingAuctions](#),
[asymmetricRange](#), [floorP](#), [maxCorrectingCoeff](#), [minCorrectingCoeff](#)

Constructor Summary

[RandomRuleMaster](#)(Zone aZone)

Appendice D

Constructor for a new RandomRuleMaster.

Method Summary

double	getPrice (double lp, double p) This method is called when the agent acts in the market.
double	getPriceInOpeningOrClosingAuctions (double lp, double p) This method is called before the market is open or before closing auction.

Methods inherited from class [BasicSumRuleMaster](#)

[setAgentProbToActInOpeningOrClosingAuctions](#), [setAsymmetricRange](#),
[setFloorP\\$andAgentProbToActBelowFloorP](#), [setMaxCorrectingCoeff](#),
[setMinCorrectingCoeff](#)

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#),
[wait](#), [wait](#)

Constructor Detail

RandomRuleMaster

```
public RandomRuleMaster(Zone aZone)  
    Constructor for a new RandomRuleMaster.
```

Method Detail

[getPriceInOpeningOrClosingAuctions](#)

```
public double getPriceInOpeningOrClosingAuctions(double lp,  
                                                double p)  
    This method is called before the market is open or before closing auction.
```

[getPrice](#)

```
public double getPrice(double lp,  
                      double p)  
    This method is called when the agent acts in the market.
```

Class [MarketImitatingAgent](#)

```
java.lang.Object  
├─SwarmObjectImpl  
├─BasicSumAgent  
└─MarketImitatingAgent
```

```
public class MarketImitatingAgent  
    extends BasicSumAgent
```

[MarketImitatingAgent](#) inherits from [BasicSumAgent](#) and operates in an imitative way, following market tendency.

Author: Bruno Mencarelli integrations of Antonio de Ruvo

See Also: [Serialized Form](#)

Field Summary	
double[]	order This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.
RandomRuleMaster	ruleMaster This is the Rule Master of the market imitating agent.

Fields inherited from class BasicSumAgent
agentProbToActWithMarketPrice , agentWealthAtMeanDailyPrice , asymmetricBuySellProb , buySellSwitch , executedPrices , iMax , liquidityQuantity , maxOrderQuantity , meanOperatingPrice , number , price , printing , shareQuantity , shareValueAtMeanDailyPrice , theBook

Constructor Summary
MarketImitatingAgent (Zone aZone, int maxOrderQuantity) Constructor for a new market imitating agent.

Method Summary	
void	act0 () This method is called in the first phase of the day.
void	act1 () This method is called during the simulated day.
double	chooseBuySellSwitch (Book theBook, double asymmetricBuySellProb) This method allows the agent to operate following the market tendency.
void	setRuleMaster (RandomRuleMaster r) This method sets the Rule Master.

Methods inherited from class BasicSumAgent
act2 , extract , getWealthAtMeanDailyPrice , setAgentProbToActWithMarketPrice , setAsymmetricBuySellProb , setBook , setConfirmationOfExecutedPrice , setMaxOrderQuantity , setNumber , setPrinting

Methods inherited from class java.lang.Object
clone , equals , finalize , getClass , hashCode , notify , notifyAll , toString , wait , wait

Appendice D

Field Detail

order

```
public double[] order
```

This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.

ruleMaster

```
public RandomRuleMaster ruleMaster
```

This is the Rule Master of the market imitating agent.

Constructor Detail

MarketImitatingAgent

```
public MarketImitatingAgent (Zone aZone,  
                             int maxOrderQuantity)
```

Constructor for a new market imitating agent.

Method Detail

chooseBuySellSwitch

```
public double chooseBuySellSwitch (Book theBook,  
                                   double asymmetricBuySellProb)
```

This method allows the agent to operate following the market tendency.

setRuleMaster

```
public void setRuleMaster (RandomRuleMaster r)
```

This method sets the Rule Master.

act0

```
public void act0 ()
```

This method is called in the first phase of the day.

Overrides:

[act0](#) in class [BasicSumAgent](#)

act1

```
public void act1 ()
```

This method is called during the simulated day. The agent acts in the market.

Overrides:

[act1](#) in class [BasicSumAgent](#)

Class LocallyImitatingAgent

```
java.lang.Object  
  ↳ SwarmObjectImpl  
    ↳ BasicSumAgent  
      ↳ LocallyImitatingAgent
```

```
public class LocallyImitatingAgent  
    extends BasicSumAgent
```

LocallyImitatingAgent inherits from BasicSumAgent and operates in an imitative way, following other agents actions (last operating agents or "local" imitation).

Author:

Bruno Mencarelli integrations of Antonio de Ruvo

See Also:

[Serialized Form](#)

Field Summary	
double[]	order This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.
(package private) RandomRuleMaster	ruleMaster This is the RuleMaster of the locally imitating agent.

Fields inherited from class BasicSumAgent
agentProbToActWithMarketPrice , agentWealthAtMeanDailyPrice , asymmetricBuySellProb , buySellSwitch , executedPrices , iMax , liquidityQuantity , maxOrderQuantity , meanOperatingPrice , number , price , printing , shareQuantity , shareValueAtMeanDailyPrice , theBook

Constructor Summary
LocallyImitatingAgent (Zone aZone, int maxOrderQuantity) Constructor for a new locally imitating agent.

Method Summary	
void	act0 () This method is called in the first phase of the day.
void	act1 () This method is called during the simulated day.
double	chooseBuySellSwitch (Book theBook, double asymmetricBuySellProb) This method allows the agent to operate following the other agents' actions.
void	setRuleMaster (RandomRuleMaster r) This method sets the Rule Master.

Methods inherited from class BasicSumAgent
act2 , extract , getWealthAtMeanDailyPrice , setAgentProbToActWithMarketPrice , setAsymmetricBuySellProb , setBook , setConfirmationOfExecutedPrice , setMaxOrderQuantity , setNumber , setPrinting

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Appendice D

Field Detail

order

```
public double[] order
```

This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.

ruleMaster

```
RandomRuleMaster ruleMaster
```

This is the RuleMaster of the locally imitating agent.

Constructor Detail

LocallyImitatingAgent

```
public LocallyImitatingAgent(Zone aZone,  
                             int maxOrderQuantity)
```

Constructor for a new locally imitating agent.

Method Detail

chooseBuySellSwitch

```
public double chooseBuySellSwitch(Book theBook,  
                                   double asymmetricBuySellProb)
```

This method allows the agent to operate following the other agents' actions.

setRuleMaster

```
public void setRuleMaster(RandomRuleMaster r)
```

This method sets the Rule Master.

act0

```
public void act0()
```

This method is called in the first phase of the day.

Overrides:

[act0](#) in class [BasicSumAgent](#)

act1

```
public void act1()
```

This method is called during the simulated day. The agent acts in the market.

Overrides:

[act1](#) in class [BasicSumAgent](#)

Class StopLossAgent

```
java.lang.Object  
  ↳ SwarmObjectImpl  
    ↳ BasicSumAgent  
      ↳ StopLossAgent
```

```
public class StopLossAgent
```

```
extends BasicSumAgent
```

StopLossAgent inherits from BasicSumAgent and operates in random way, like a random agent. After that, its behavior can be modified by a "stop loss" decision, operating in two possible ways (i) without memory: if the current price, i.e. the last executedPrice (at day t) is \geq to (mean price at day t-stopLossInterval) \cdot (1+maxLossRate) \leq to (mean price at day t-stopLossInterval) \cdot (1-maxLossRate) and checkingIfShortOrLong=0, the agent buys/sells, at the current price, a quantity of shares between 1 and maxOrderQuantity; (ii) with memory (the short or long position of the agent is taken in account; if the agent has no position, doesn't operate here): if the current price, i.e. the last executedPrice (at day t) is \geq to (mean price at day t-stopLossInterval) \cdot (1+maxLossRate) \leq to (mean price at day t-stopLossInterval) \cdot (1-maxLossRate) and checkingIfShortOrLong=1, the agent buys if short (shareQuantity0) at the current price, a quantity of shares between 1 and maxOrderQuantity (regardless its shareQuantity value).

Author: Bruno Mencarelli integration of Antonio de Ruvo

See Also: [Serialized Form](#)

Field Summary	
(package private) int	checkingIfShortOrLong These are the variables that the agent checks for the stop-loss strategy.
(package private) double	maxLossRate This is the maximum rate of loss that the agent will accept before applying the strategy.
double []	order This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.
StopLossRuleMaster	ruleMaster This is the RuleMaster of the stop loss agent.
(package private) int	stopLossInterval These are the variables that the agent checks for the stop-loss strategy.

Fields inherited from class BasicSumAgent
agentProbToActWithMarketPrice , agentWealthAtMeanDailyPrice , asymmetricBuySellProb , buySellSwitch , executedPrices , iMax , liquidityQuantity , maxOrderQuantity , meanOperatingPrice , number , price , printing , shareQuantity , shareValueAtMeanDailyPrice , theBook

Constructor Summary
StopLossAgent (Zone aZone, int maxOrderQuantity) Constructor for a new stop loss agent.

Method Summary	
void	act0 () This method is called in the first phase of the day.
void	act1 () This method is called during the simulated day.
void	setMaxLossRate\$andCheckingIfShortOrLong (double r, int sl) This method sets the maxlossrate and the option checking if short or long.
void	setRuleMaster (StopLossRuleMaster r) This method sets the Rule Master.
void	setStopLossInterval (int i) This method sets the stoplossinterval.

Methods inherited from class BasicSumAgent
act2 , extract , getWealthAtMeanDailyPrice , setAgentProbToActWithMarketPrice .

Appendice D

[setAsymmetricBuySellProb](#), [setBook](#), [setConfirmationOfExecutedPrice](#),
[setMaxOrderQuantity](#), [setNumber](#), [setPrinting](#)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait,
wait, wait

Field Detail

order

public double[] **order**

This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.

stopLossInterval

int **stopLossInterval**

These are the variables that the agent checks for the stop-loss strategy.

checkingIfShortOrLong

int **checkingIfShortOrLong**

These are the variables that the agent checks for the stop-loss strategy.

maxLossRate

double **maxLossRate**

This is the maximum rate of loss that the agent will accept before applying the strategy.

ruleMaster

public [StopLossRuleMaster](#) **ruleMaster**

This is the RuleMaster of the stop loss agent.

Constructor Detail

StopLossAgent

public **StopLossAgent**(Zone aZone,
int maxOrderQuantity)

Constructor for a new stop loss agent.

Method Detail

setRuleMaster

public void **setRuleMaster**([StopLossRuleMaster](#) r)

This method sets the Rule Master.

setStopLossInterval

public void **setStopLossInterval**(int i)

This method sets the stoplossinterval.

setMaxLossRate\$andCheckingIfShortOrLong

public void **setMaxLossRate\$andCheckingIfShortOrLong**(double r,
int sl)

This method sets the maxlossrate and the option checking if short or long.

act0

```
public void act0()
    This method is called in the first phase of the day.
    Overrides:
    act0 in class BasicSumAgent
```

act1

```
public void act1()
    This method is called during the simulated day. The agent acts in the market.
    Overrides:
    act1 in class BasicSumAgent
```

Class StopLossAgent

```
java.lang.Object
├─SwarmObjectImpl
│   └─BasicSumAgent
│       └─StopLossAgent
```

```
public class StopLossAgent
    extends BasicSumAgent
```

StopLossAgent inherits from BasicSumAgent and operates in random way, like a random agent. After that, its behavior can be modified by a "stop loss" decision, operating in two possible ways (i) without memory: if the current price, i.e. the last executedPrice (at day t) is \geq to (mean price at day t-stopLossInterval) \cdot (1+maxLossRate) \leq to (mean price at day t-stopLossInterval) \cdot (1-maxLossRate) and checkingIfShortOrLong=0, the agent buys/sells, at the current price, a quantity of shares between 1 and maxOrderQuantity; (ii) with memory (the short or long position of the agent is taken in account; if the agent has no position, doesn't operate here): if the current price, i.e. the last executedPrice (at day t) is \geq to (mean price at day t-stopLossInterval) \cdot (1+maxLossRate) \leq to (mean price at day t-stopLossInterval) \cdot (1-maxLossRate) and checkingIfShortOrLong=1, the agent buys if short (shareQuantity0) at the current price, a quantity of shares between 1 and maxOrderQuantity (regardless its shareQuantity value).

Author: Bruno Mencarelli integration of Antonio de Ruvo

See Also: [Serialized Form](#)

Field Summary	
(package private) int	checkingIfShortOrLong These are the variables that the agent checks for the stop-loss strategy.
(package private) double	maxLossRate This is the maximum rate of loss that the agent will accept before applying the strategy.
double[]	order This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.
StopLossRuleMaster	ruleMaster This is the RuleMaster of the stop loss agent.
(package private) int	stopLossInterval These are the variables that the agent checks for the stop-loss strategy.

Fields inherited from class BasicSumAgent
agentProbToActWithMarketPrice , agentWealthAtMeanDailyPrice , asymmetricBuySellProb , buySellSwitch , executedPrices , iMax , liquidityQuantity , maxOrderQuantity , meanOperatingPrice , number , price , printing , shareQuantity .

Appendice D

[shareValueAtMeanDailyPrice](#), [theBook](#)

Constructor Summary

[StopLossAgent](#)(Zone aZone, int maxOrderQuantity)
Constructor for a new stop loss agent.

Method Summary

void	act0 () This method is called in the first phase of the day.
void	act1 () This method is called during the simulated day.
void	setMaxLossRate\$andCheckingIfShortOrLong (double r, int sl) This method sets the maxlossrate and the option checking if short or long.
void	setRuleMaster (StopLossRuleMaster r) This method stes the Rule Master.
void	setStopLossInterval (int i) This method sets the stoplossinterval.

Methods inherited from class [BasicSumAgent](#)

[act2](#), [extract](#), [getWealthAtMeanDailyPrice](#), [setAgentProbToActWithMarketPrice](#),
[setAsymmetricBuySellProb](#), [setBook](#), [setConfirmationOfExecutedPrice](#),
[setMaxOrderQuantity](#), [setNumber](#), [setPrinting](#)

Methods inherited from class java.lang.Object

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#),
[wait](#), [wait](#)

Field Detail

`order`

public double[] **order**
This is the vector containing the basic input of an order: the price of the offer, the quantity of shares and the identification number of the agent.

`stopLossInterval`

int **stopLossInterval**
These are the variables that the agent checks for the stop-loss strategy.

`checkingIfShortOrLong`

int **checkingIfShortOrLong**

These are the variables that the agent checks for the stop-loss strategy.

maxLossRate

double **maxLossRate**
This is the maximum rate of loss that the agent will accept before applying the strategy.

ruleMaster

public [StopLossRuleMaster](#) **ruleMaster**
This is the RuleMaster of the stop loss agent.

Constructor Detail

StopLossAgent

public **StopLossAgent**(Zone aZone,
int maxOrderQuantity)
Constructor for a new stop loss agent.

Method Detail

setRuleMaster

public void **setRuleMaster**([StopLossRuleMaster](#) r)
This method sets the Rule Master.

setStopLossInterval

public void **setStopLossInterval**(int i)
This method sets the stoplossinterval.

setMaxLossRate\$andCheckingIfShortOrLong

public void **setMaxLossRate\$andCheckingIfShortOrLong**(double r,
int sl)
This method sets the maxlossrate and the option checking if short or long.

act0

public void **act0**()
This method is called in the first phase of the day.
Overrides:
[act0](#) in class [BasicSumAgent](#)

act1

public void **act1**()
This method is called during the simulated day. The agent acts in the market.
Overrides:
[act1](#) in class [BasicSumAgent](#)

Class CurrentAgent

```
java.lang.Object
├─SwarmObjectImpl
└─CurrentAgent
```

public class **CurrentAgent**
extends [SwarmObjectImpl](#)

This is a ghost agent which simply transfers the 'act' message to the first agent in the list agentList and rotate it this trick is necessary to send the act message agentNumber times; so the observer can display the price of each acting step.

Author:

Marco Agagliate

Appendice D

See Also:

[Serialized Form](#)

Field Summary	
(package private) ListImpl	agentList This is the agent list.
int	printing If 1 many objects print data on the terminal window.

Constructor Summary	
CurrentAgent (Zone aZone)	Constructor for a new CurrentAgent.

Method Summary	
void	act1 () The method calls the act1 method from an agent, which is in a list.
void	setAgentList (ListImpl l) The method sets agentList.
void	setPrinting (int p) To set the value of option printing.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail
agentList
ListImpl agentList This is the agent list.
printing
public int printing If 1 many objects print data on the terminal window.

Constructor Detail
CurrentAgent
public CurrentAgent (Zone aZone) Constructor for a new CurrentAgent.

Method Detail
setPrinting
public void setPrinting (int p)

To set the value of option printing.

setAgentList

```
public void setAgentList(ListImpl l)
    The method sets agentList.
```

act1

```
public void act1()
    The method calls the act1 method from an agent, which is in a list.
```

Class CurrentIstant

```
java.lang.Object
├─SwarmObjectImpl
└─CurrentIstant
```

```
public class CurrentIstant
    extends SwarmObjectImpl
```

This is a class for calling a different number of agents for every istant of simulating day. The CurrentAgent is called a numberOfOperatinAgents times.

Author:

Marco Agagliate

See Also:

[Serialized Form](#)

Field Summary	
int	maxNumberOfOperatingAgents The maximum number of operating agents.
int	numberOfOperatingAgents The number of operating agents per istant.
int	percentageOfOperatingAgents The percentage quota of agents which act in the market.
int	printing If I many objects print data on the terminal window.
(package private) CurrentAgent	theCurrentAgent The Current Agent of the simulation.
boolean	typeOfPercentage The type of percentage of agents.

Constructor Summary	
CurrentIstant (Zone aZone)	Constructor for a new CurrentIstant.

Method Summary	
void	callAgents () The method calls the CurrentAgent numberOfOperatingAgents times.

Appendice D

void	setCurrentAgent (CurrentAgent c) The method sets theCurrentAgent.
void	setMaxNumberOfOperatingAgents (int n) This method sets the maximum value of operating agents.
void	setPercentageOfOperatingAgents (int p) This method sets the percentage of operating agents.
void	setPrinting (int p) To set the value of option printing.
void	setTypeOfPercentage (boolean t) This method sets the type of percentage.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

theCurrentAgent

[CurrentAgent](#) **theCurrentAgent**
The Current Agent of the simulation.

numberOfOperatingAgents

public int **numberOfOperatingAgents**
The number of operating agents per istant.

maxNumberOfOperatingAgents

public int **maxNumberOfOperatingAgents**
The maximum number of operating agents.

percentageOfOperatingAgents

public int **percentageOfOperatingAgents**
The percentage quota of agents which act in the market.

typeOfPercentage

public boolean **typeOfPercentage**
The type of percentage of agents. If it is true percentage is a maximum value; if it is false percentage is a fixed value.

printing

public int **printing**
If I many objects print data on the terminal window.

Constructor Detail

CurrentIstant

public **CurrentIstant** (Zone aZone)
Constructor for a new CurrentIstant.

Method Detail

setCurrentAgent

```
public void setCurrentAgent (CurrentAgent c)
    The method sets theCurrentAgent.
```

setMaxNumberOfOperatingAgents

```
public void setMaxNumberOfOperatingAgents (int n)
    This method sets the maximum value of operating agents.
```

setPercentageOfOperatingAgents

```
public void setPercentageOfOperatingAgents (int p)
    This method sets the percentage of operating agents.
```

setTypeOfPercentage

```
public void setTypeOfPercentage (boolean t)
    This method sets the type of percentage.
```

setPrinting

```
public void setPrinting (int p)
    To set the value of option printing.
```

callAgents

```
public void callAgents ()
    The method calls the CurrentAgent numberOfOperatingAgents times. The CurrentAgent calls act1 method from an agent, which is in a list.
```

Class SwarmUtils

```
java.lang.Object
├─SwarmUtils
```

```
public class SwarmUtils
    extends java.lang.Object
```

These two static methods create a selector. A selector is an object of the Selector class used by Swarm to encapsulate a "message" destined for an object, where the message is the name of a method defined for the class to which the object belongs. Because the method must indeed be defined for the class of the object and because this can be determined only at run time, there is a possibility that the creation of the selector will throw an exception if the class and the method do not match. Java requires that events that might throw exceptions be enclosed in try/catch blocks. If there is an error creating the new selector in the try block, the catch block can handle the resulting exception. Here we have taken a pretty crude approach to handling the exception: we simply call System.exit(1). (Note that the "return null" which ends the catch block is there only to tell the compiler that "return sel" will never be reached if an exception occurs and sel is undefined. We'll exit on an exception before ever returning sel to the calling method.)

Note that the getSelector method overloaded. It can be called with either a string containing the class name as its first argument, or with an object of the desired class. In the first case, the string is converted to a class identifier using the forName() method, while in the second case getClass() is used to find the class identifier for the object. The second argument to getSelector is always a string containing the method name. (The boolean "false" at the end of the Selector constructor is theobjCFlag. It allows one to use ObjectiveC-type key/value method syntax. Since we always use Java-style method names, for us the flag is always false.)

Author:

Charles P. Staelin

Constructor Summary	
SwarmUtils ()	

Appendice D

Method Summary	
static Selector	getSelector (java.lang.Object obj, java.lang.String method)
static Selector	getSelector (java.lang.String name, java.lang.String method)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SwarmUtils

```
public SwarmUtils()
```

Method Detail

getSelector

```
public static Selector getSelector(java.lang.String name,  
                                     java.lang.String method)
```

getSelector

```
public static Selector getSelector(java.lang.Object obj,  
                                     java.lang.String method)
```

Class Matrix

```
java.lang.Object  
├─SwarmObjectImpl  
└─Matrix
```

```
public class Matrix  
extends SwarmObjectImpl
```

This is the class for matrixes and vectors.

Author: Marco Agagliate

See Also: [Serialized Form](#)

Field Summary	
double [][]	matr This is a matrix of double
double []	vect This is a vector of double

Constructor Summary

Matrix (Zone aZone, int rows) Constructors for a new Matrix.	
Matrix (Zone aZone, int rows, int cols)	

Method Summary	
int	getLength ()
double	P (int rows) This method returns a double from a position of the vector.
void	P\$setFrom (int rows, double x) This method sets a double in a position of the vector.
double	R\$C (int rows, int cols) This method returns a double from a position of the matrix.
void	R\$C\$setFrom (int rows, int cols, double x) This method sets a double in a position of the matrix.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail
vect
public double[] vect This is a vector of double
matr
public double[][] matr This is a matrix of double

Constructor Detail
Matrix
public Matrix (Zone aZone, int rows) Constructors for a new Matrix.
Matrix
public Matrix (Zone aZone, int rows, int cols)

Method Detail
P\$setFrom
public void P\$setFrom (int rows, double x) This method sets a double in a position of the vector.

Appendice D

P

```
public double P(int rows)
    This method returns a double from a position of the vector.
```

getLength

```
public int getLength()
```

RSC\$setFrom

```
public void RSC$setFrom(int rows,
                        int cols,
                        double x)
    This method sets a double in a position of the matrix.
```

RSC

```
public double RSC(int rows,
                  int cols)
    This method returns a double from a position of the matrix.
```

RIFERIMENTI BIBLIOGRAFICI

AGAGLIATE, M. (2003), *La simulazione ad agenti e lo studio dell'economia: ricostruzione virtuale del mercato borsistico*, tesi di laurea, Facoltà di Economia, Torino.

BARCKLEY, J., ROSSER, JR. (1999), *On the Complexities of Complex Economic Dynamics*, Journal of Economic Perspectives, volume 13 numero 4 pages 169-192.

CAPPELLINI, A. N. (2003), *Esperimenti su mercati finanziari con agenti naturali e artificiali*, tesi di laurea, Facoltà di Economia, Torino.

CHINAGLIA, M. (2004), *Modelli di simulazione e mercati finanziari: la formazione di bolle e crash*, tesi di laurea, Facoltà di Economia, Torino.

DAY, R. H. (1994), *Complex economic Dynamics, Volume I: an Introduction to dynamical System and Market Mechanisms*, Cambridge, MA: MIT Press.

DAMILANO E ALTRI (2002), *Il mercato azionario*, Giappichelli editore.

ECKEL, B. (2002), *Thinking in Java*. Ed. Italiana Apogeo.

FERRARIS, G. (2000), *Algoritmi genetici e classifier system nei modelli di agenti*.

FORESTIERI G, MOTTURA P., *Il sistema finanziario: istituzioni, mercati e modelli finanziari*, Ed. Egea.

GILBERT, N., TERNA, P. (2000), *How to built and use agent-based models in social science*. Mind & Society, no. 1, pp.57-72.

LEBARON, B. (2002), *Building the Santa Fe Artificial Stock Market*, Brandeis University.

MEZZERA, P. (2003), *Aste a chiamata controllo dei prezzi ed esperimenti con agenti umani e artificiali in un modello di simulazione di borsa*, tesi di laurea, Facoltà di Economia, Torino.

PARISI, D. (1999), *Mente. Nuovi modelli della vita artificiale*. Universale Paparbacks, Il Mulino, Bologna.

PARISI, D. (2001), *Simulazioni: la realtà rifatta nel computer*. Universale Paparbacks, Il Mulino, Bologna.

ORMEROD, P. (1998), *L'economia della farfalla: società mercato e comportamento*.

OSTROM, T. (1988), *Computer simulation: the third symbol system*, Journal of Experimental Social Psychology, 24:381-392.

STAELIN, C. P. (2000), *jSIMPLEBUG: A swarm tutorial for Java*.

TERNA, P. (1995), *Reti neurali artificiali e modelli con agenti adattivi*, XXXV riunione scientifica annuale della società degli economisti 28-29 Ottobre 1994, Milano (versione giugno 1995).

TERNA, P. (2000c), *Hayek e il connessionismo: modelli con agenti che apprendono*, in G. Clerico e S. Rizzello (eds.), *Il pensiero di Friedrich von Hayek*. Torino, Utet.

TERNA, P. (2001), *Cognitive Agents Behaving in a Simple Stock Market Structure*, in F. Luna and A. Perrone (eds), *Agent-Based Methods in Economics and Finance: Simulations in Swarm*. Kluwer Academic, Dordrecht and London, pp.188-227.

TERNA, P. (2002c), *La simulazione come strumento di indagine per l'economia*. Workshop su "Scienze Cognitive ed Economia" organizzato dalla Associazione Italiana di Scienze Cognitive, 21 settembre 2002, Rovereto.

TERNA, P. (2003), *Another ASM (Artificial Stock Marker), so AASM: Why?*, in pubblicazione.